C++: устройство и применение

Лебедев П.А.

6 декабря 2016 г.

Замечание: это рабочая версия книги. Известно, что она не заверше, никарректна и содержит много ошибок форм атирования.

Оглавление

Предисловие

Данный текст предназначен для изучающих дисциплину «Языки программирования» и содержит весь материал, необходимый для успешной теоретической подготовки и написания практических работ. Если вы не согласны с данным утверждением, автор настоятельно просит уведомить его об этом, чтобы недочёты и упущения оперативно исправлялись.

Выделенный ***жирным курсивом*** текст обычно встречается при первом употреблении терминов вместе с их определением. Знание этих определений является основополагающим как для понимания дальнейшего материала, так и в практической работе. Для большинства терминов в скобках даётся англоязычный вариант, что поможет вам при чтении англоязычной литературы и сообщений, выдаваемых инструментальными средствами. Если таким образом выделен не термин, а целое предложение, значит оно несёт исключительную важность. Также следует обращать особое внимание на содержимое списков.

Краткие фрагменты и отдельные «слова» языков программирования, а также любой другой текст, предназначенный для ввода в компьютер, набран `моноширинным шрифтом`. При описании синтаксиса некоторых конструкций, среди фиксированных элементов, набранных таким образом, могут встречаться переменные части, которые набраны обычным шрифтом. Фиксированные части подлежат вводу в точности, как указано, переменные части следует при использовании заменять соответствующими им по смыслу элементами программы. Нижний индекс «опц» означает, что данный элемент является опциональным. Например:

`return` выражение$_{\text{опц}}$ `;`

Сначала следует записать слово `return`, являющееся фиксированным элементом. Вместо слова «выражение» нужно ввести некоторое выражение, которое может быть опущено, что показано нижним индексом «опц». Вслед за этим следует записать точку с запятой, также являющуюся фиксированным элементом.

> Текст, выделенный чертами на полях, как этот абзац, содержит расширенную, более сложную часть материала. При первом прочтении её можно пропустить, но вернуться к ней следует обязательно перед переходом к следующей главе, чтобы, если не помнить наизусть всех хитростей, то хотя бы иметь представление о деталях и возможных проблемах и уметь быстро находить в тексте нужный материал, когда в этом возникнет необходимость.

Крупные фрагменты программ имеют своё отдельное место в тексте. Они раскрашены для улучшения зрительного восприятия, а их строки пронумерованы на полях для удобства ссылок на них:

```
1  int example(int& a,int b,int c)
2  {
3      // Это фрагмент текста программы, также называемый листингом.
4      a = b+c;
5      return 0;
6  }
```

Номера строк, разумеется, к самому тексту программы не относятся. Нетривиальные по объёму фрагменты кода, не являющиеся самодостаточными программами, обычно отмечаются соответствующим комментарием.

В примерах взаимодействия программ с пользователем приводится весь текст, как он выглядит на экране, при этом текст, вводимый пользователем, подчёркивается. Нажатие клавиши Enter обозначается знаком ↵. Пример:

```
Input number: 234↵
You have entered 234.
```

Часть I

Общие сведения

Глава 1

Введение в языки программирования

Этот текст призван дать читателю начальные сведения о языках программирования, т.е. о том средстве, которое позволяет воспользоваться вычислительным устройством для решения требуемых задач. Речь идёт, разумеется, об участии в активной роли разработчика, который не только пользуется уже существующими инструментами, но и улучшает существующие, а также создаёт новые.

Автор преследовал следующие цели при написании *ещё одной* книги по программированию:

- ***Дать читателю понимание происходящего на всех уровнях***, начиная от архитектуры программных систем, и заканчивая спецификой работы центрального процессора. Это книга для тех, кто хочет *понимать*, что же на самом деле происходит внутри компьютера, и почему. Без подробного рассмотрения абстрактных и практических аспектов невозможно объяснить разницу между «работающей» (внешне, здесь и сейчас) и «корректной» программой, которая *действительно* работает во всех смыслах важных для разработчиков, служб поддержки и конечных пользователей в рамках тех или иных требований.
- ***Рассмотреть язык C++ в его современном варианте***, зачастую кардинально отличном от описанного в имеющейся, особенно русскоязычной, литературе. Само преподавание этого языка, особенно в качестве первого для начинающих специалистов, часто подвергается жёсткой критике, однако автор всё равно выбрал этот путь. Во-первых, этот язык достаточно низкоуровнёвый, чтобы описание происходящего в нём на всех уровнях было достаточно прозрачным и лаконичным для изложения начинающим программистам. Отсутствие такого понимания и породило армию «шаманов», занимающих значительную долю рынка вакансий разработчиков ПО. Во-вторых, за этим языком в настоящем (и обозримом будущем) остаётся огромное количество кода который необходимо использовать и дорабатывать. Поэтому серьёзному специалисту никак не избежать знакомства с этим языком, а если это так, то следует строить систему знаний начиная с самого низкого уровня. Овладение многими языками более высокого уровня, в первую очередь использующими те же парадигмы, что и C++, после знакомства с ним, является относительно лёгким процессом, поскольку останется только достроить ещё один этаж абстракций, чего не скажешь об обратном процессе, что многократно подтверждалось на практике.

  Практически во всех аспектах своего существования язык C++ связан с языком, от которого был произведён — языком C. Язык C имеет меньшее число средств, чем C++, и имеет другие многочисленные отличия. Поскольку программисту на C++ также не избежать и встречи с C, эти различия будут приведены в приложении.
- ***Ввести все требуемые определения в том виде, в котором они даны в стандарте***, в том числе и на английском языке. Охватить 100% возможностей даже чистого стандарта языка данная книга пытаться не будет, поэтому необходимо с самого начала готовить читателя к чтению других источников. Кроме этого на этом языке говорят все инструментальные средства, с которым программисту предстоит вести диалог.

Данный текст предполагает наличие знаний в областях математики, информатики и английского языка в рамках школьной программы. Мы будем останавливаться на необходимых понятиях из дисциплин «Математический анализ», «Линейная алгебра» и «Архитектура ЭВМ» только в необходимом для данного изложения объёме, поскольку их изучение предполагается как процесс, параллельный знакомству с рассматриваемым в этом тексте материалом. Другие

затрагиваемые области науки будут отмечены, как имеющие значимость, в соответствующих разделах, чтобы направить читателя на поиск дополнительного материала в нужном направлении. Перечислим основные из них здесь в виде целей, которые данная книга ***не*** ставит перед собой:

- Детальное изучение языков ассемблера.

  Примеры на одном их этих языков будут приводиться с целью демонстрации связи кода на языке C++ с инструкциями, исполняемыми процессором, но учебником по нему данная книга не претендует являться.

- Изложение теории языков и трансляторов.

  Мы познакомимся с практически полезными понятиями, необходимыми для понимания устройства конкретного языка, но рассмотрение теории, лежащей в основе построения машинных языков как таковых, не входит в задачи данной книги.

- Детальное рассмотрение большого количества алгоритмов и структур данных.

  Сколько-либо полное рассмотрение даже очень узких и специализированных вопросов в этой области легко может занять тысячи страниц. Вместо этого будет введено базовое понятие алгоритмической сложности и рассмотрены имеющиеся на уровне стандарта языка средства, из которых путём комбинирования можно построить удовлетворительные решения для многих практических задач.

Ограничив себя конечными рамками, приступим к изложению материала.

1.1. Определение языка программирования и его аспектов

Язык программирования (ЯП) (programming language) — формальная знаковая система для планирования поведения компьютеров.

Язык программирования — ***формальный язык (formal language)***, т.е. множество конечных строк над конечным алфавитом. Алфавит определяет множество неделимых единиц, последовательности которых применительно к языку программирования называются ***программами (program)***. Хотя в большинстве случаев алфавит программы — символы, а сама программа — текст, могут использоваться и другие представления этой формальной конструкции, к ней сводимые.

Не обязательно каждая последовательность элементов алфавита относится к конкретному языку — для этого она также должна быть ***согласованной (well-formed)***, а именно удовлетворять некоторым дополнительным правилам (например, не каждая последовательность букв естественного языка имеет в его рамках смысл). Основным способом задания формального языка является именно этот набор правил построения согласованных последовательностей из элементов алфавита. Чаще всего он предполагает многоэтапное построение более сложных конструкций из простых, начиная с элементов алфавита, и заканчивая целой программой. Эти правила чаще всего называют ***синтаксисом (syntax)*** языка, а согласованные последовательности элементов алфавита — синтаксически корректными программами.

Язык программирования — ***знаковая система (notation)***, т.е. система элементов, за каждым из которых и, следовательно, за программами из них составленными в соответствие с синтаксисом языка, закреплено некоторое значение или смысл. Правила, задающие связь между синтаксически корректными элементами программы и их значением, называются ***семантикой (semantics)*** языка. В отличие от естественных языков, языки программирования не допускают разночтений — смысл каждого элемента строг и однозначен. Придание программам однозначного смысла даёт возможность использовать их для хранения и передачи информации, необходимой для планирования поведения компьютеров.

Наконец, последний элемент языка программирования, который часто не обсуждается, несмотря на его огромную важность, — это прагматика. Синтаксических и семантических аспектов достаточно, чтобы установить взаимоотношения между одним человеком и компьютером, поскольку последний не имеет целевых установок, а является всего лишь исполнителем. Но язык программирования — не только средство взаимодействия человека с компьютером, но и средство взаимодействия людей друг с другом, когда речь идёт о планировании поведения компьютеров.

Приведём пример: математическая запись $a \times b$ относительно однозначна в том смысле, что соответствует умножению двух величин. Синтаксический её аспект — правила записи

знака операции между двумя операндами. Семантический аспект — правила, определяющие понятие «умножение». Но какой *проблеме* соответствует эта запись? Поиск площади прямоугольника по сторонам, нахождение мощности, зная ток и напряжение, расчёт общего числа предметов, зная число коробок и предметов в каждой из них — вот несколько принципиально разных, но допустимых интерпретаций данной записи. Более того, наше предположение о том, что знак × соответствует скалярному умножению, также является не беспочвенным, а следствием интуитивного поиска подходящей *целевой установки*. Таким образом, для установления взаимопонимания между людьми, имеющими дело с одной программой, необходим контекст, в первую очередь, одинаковые представления о сути решаемой задачи. Кроме того, даже отдельные элементы программы могут нести смысл, не описываемый его формальной семантикой.

Без рассмотрения синтаксиса и семантики языков книге по языкам программирования обойтись не удастся, но автор будет стараться уделить не меньшее внимание и последнему, наивысшему уровню — ***прагматике (pragmatics)***, т.е. языковым средствам установления контекста, ролей и передачи целевых установок между людьми. Переоценить важность этого аспекта невозможно, поскольку бурно развивающиеся технологии приводят к тому, что программа, которую невозможно понять для доработки и исправления ошибок, становится никому не нужной с момента её первого написания. В любом случае, программы гораздо чаще читают, нежели пишут.

Отметим, что для использования в вычислительной технике разработано множество искусственных языков, не каждый из которых является языком программирования — они отличаются от других тем, что предназначены для планирования поведения компьютеров с целью решения поставленных людьми задач.

Читателю, которого интересуют другие основополагающие теоретические принципы языков программирования можно порекомендовать книгу [6].

1.2. История языков программирования

Языки программирования принято разделять на несколько поколений. Точные границы и принадлежность языка программирования не всегда удаётся определить строго, особенно учитывая факт изменения принятой классификации со временем. Приведём примерный план исторического развития ЯП с указанием ключевых особенностей, на которых построена эта классификация.

1. ***Машинные коды (machine code)***. Данные языки программирования напрямую отражают систему команд того или иного вычислительного средства, т.е. являются специфичными для семейств или даже отдельных аппаратных средств. Это самый низкий уровень, который может иметь язык программирования, если не опускаться до внутреннего устройства соответствующего процессора. Поскольку, фактически, каждому программируемому вычислительному устройству соответствует такой язык, выбрать «первого» представителя можно по-разному в зависимости от того, что же считать первым компьютером и по каким критериям. Автор предлагает в качестве примера рассмотреть электромеханический компьютер Z3, разработанный Конрадом Цузе и завершённый в 1941-м году, который являлся первым работающим в действительности (а не на бумаге) программируемым устройством. В качестве современных представителей языков программирования первого поколения приведём две наиболее распространённые в настоящее время архитектуры центральных процессоров: x86 (почти наверняка используется в вашем персональном компьютере) и ARM (вероятнее всего используется в вашем смартфоне). Формально система машинных кодов описывает лишь соответствие входных данных, составляющих программу, тем или иным действиям исполнительного устройства, никак не описывая представление этих инструкций в пригодном для чтения человеком виде. Как следствие, вместо них даже на самом низком уровне программирования применяются языки ассемблера.
2. ***Языки ассемблера (assembly language)***. Специфической чертой этого поколения языков программирования является прямое соответствие один-в-один команд этого языка и соответствующих инструкций машинного кода. Таким образом, языки ассемблера являются формой записи машинного кода в форме, читаемой человеком. Для каждого машинного языка обычно существует минимум один язык ассемблера, а для популярных архитектур — несколько. Одна из ранних ЭВМ, называемая EDSAC (1949 г.), имела мнемоническую однобуквенную систему записи команд, считающуюся первым языком ассемблера. Архитектура современных ПК x86 имеет множество близких по синтаксису языков ассемблера,

большая часть которых является производной от одного из форматов: Intel или AT&T. Современные языки ассемблера могут включать дополнительные средства, нарушающие принцип соответствия 1-к-1 с машинным кодом, включая работу с макросами, но всё равно не теряют своего основного свойства — машинозависимости, то есть привязанности к конкретной аппаратной архитектуре.

3. ***Машинно-независимые языки (hardware-independent language)***. В 50–е годы XX века появились первые языки программирования, которые стали использовать более сложные конструкции в своём составе, не отражающие напрямую соответствующие машинные инструкции. В то время их классифицировали как языки «высокого уровня», хотя с современной точки зрения их свойств недостаточно для такого звания. Тем не менее, свойство независимости от используемой архитектуры вычислительной системы принципиально отличает их от машинных кодов и языков ассемблера. Ранними представителями таких языков были, например, Fortran, ALGOL и COBOL. Несмотря на значительное усложнение относительно своих предшественников, многие современные языки тоже относятся к этой группе. Среди них можно назвать языки C и C++, последнему из которых будет посвящена основная часть этой книги, а также Java, Python и многие другие.

4. ***Предметно-ориентированные языки и среды (domain-specific language)***. Возникновению языков «уровня более высокого, чем языки высокого уровня» способствовала идея сделать персональные компьютеры, которые только начинали получать распространение, инструментом не только программистов, но и специалистов других профессий, в первую очередь, речь шла о задачах бизнеса. Язык Mark IV, разработанный Informatics, Inc. в 1967 году позволил, по отзывам клиентов, многократно ускорить разработку решений задач инвентаризации и торговой аналитики по сравнению с языками третьего поколения. К языкам четвёртого поколения относят языки, нацеленные на решение задач конкретной предметной области, интегрированные с соответствующими средами и информационными системами. За счёт специализации, они способны предложить конструкции, детально отражающие конкретную предметную область, что обычно находится за пределами возможностей языков программирования общего назначения. В настоящее время языки специального назначения существуют практически во всех областях применения, как для решения прикладных задач в конкретных предметных областях, так и для решения проблем, возникающих в самой компьютерной среде. Назовём в качестве современных представителей этого поколения язык запросов к реляционным базам данных SQL, многочисленные языки оболочек командной строки операционных систем и язык платформы 1C.

5. ***Языки программирования пятого поколения***. Понятие языков пятого поколения как таковое не является устоявшимся — иногда к нему даже пытаются отнести по соображениям маркетинга продвинутые языки четвёртого уровня. Отличительная особенность, которая ожидается от языка, полноправно претендующего на принадлежность этому классу — самостоятельный поиск машиной способа решения задачи по её условию. Столь завораживающая цель казалось доступной на волне развития вычислительных мощностей в 1980–е годы, когда некоторыми государствами были вложены значительные средства в разработки в этом направлении. К сожалению, ожидаемого перехода количества в качество не произошло, и большая часть результатов проведённых исследований осталась вне области внимания большинства программистов. Тем не менее, созданные в рамках этой идеологии языки, хотя и не являются решением поставленной сверхзадачи, содержат важные достижения. Ослабив требования, скажем, что под языками пятого поколения в настоящее время принято понимать языки, осуществляющие поиск решения по заданным ограничениям в достаточно узкой области. Наиболее известным представителем языков такого типа является Пролог (1972 г.) — система логического программирования на языке предикатов, оперирующая фактами, запросами и правилами вывода.

В настоящее время наблюдается тенденция развития и создания новых языков программирования, состоящая во включении элементов, присущих языкам четвёртого и пятого поколения, в продвинутые машинно-независимые языки.

Классифицировать языки программирования можно по различным признакам. При рассмотрении одной из таких классификаций — по поколениям — мы уже столкнулись с понятием предметно-ориентированных языков в противовес к языкам программирования ***общего назначения (general purpose)***. Это классификация по ответу на вопрос «для решения каких задач предназначен язык?»

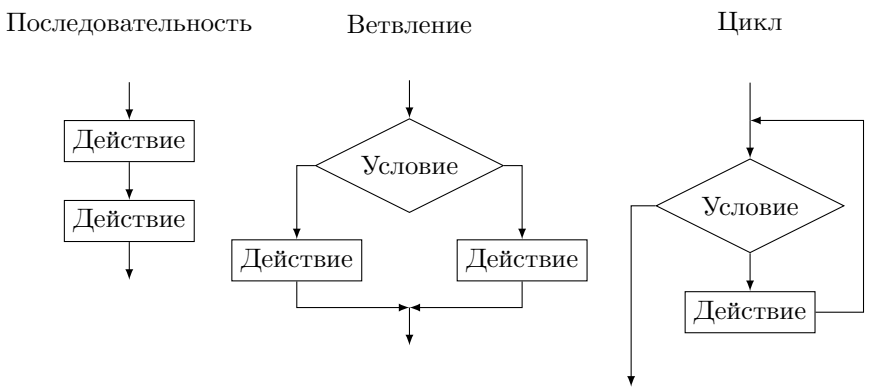


Рис. 1.1: Конструкции структурного программирования

1.3. Парадигмы программирования

По выбору основных понятий и их воплощению в элементах языка программирования последние относят к той или иной ***парадигме программирования (programming paradigm)***. Большинство языков представляют не одну, а сразу несколько парадигм программирования. Выделяют множество парадигм программирования, многие из которых являются уточнениями и частными случаями других. Рассмотрим основные парадигмы программирования.

Императивная (imperative) парадигма программирования подразумевает запись программы в виде последовательности явных инструкций, ведущих к цели, т.е. в соответствии с ***алгоритмом (algorithm)*** решения задачи. Конкретное формальное определение понятия «алгоритм» может варьироваться в зависимости от позиции той или иной научной области. Одним из наиболее общих, удовлетворяющих идее императивной парадигмы программирования и достаточным в рамках данной дисциплины, является «набор инструкций, описывающих порядок действий исполнителя для достижения результата за конечное число действий». Императивная парадигма программирования возникла первой и продолжает оставаться одной из основных, поскольку напрямую соответствует устройству подавляющего большинства аппаратных средств.

Первые вычислительные машины обладали очень малыми ресурсами, и разработчики предпринимали значительные усилия, чтобы выжать из них всё возможное. К сожалению, используемые при этом ухищрения негативно сказывались на читаемости программ. Основной проблемой являлась хаотичная передача управления в погоне за сокращением объёма программы, что часто приводило к эффекту, получившему название «макаронный код» — запутанное переплетение, в котором трудно разобраться. Однако в 1966 году математиками Коррадо Бёмом и Джузеппе Якопини была доказана теорема, показывающая, что любой алгоритм может быть преобразован к виду, содержащему только три различные структурные конструкции:

1. ***Последовательность (sequence)*** — исполнение инструкций по порядку.
2. ***Ветвление (selection (branching))*** — исполнение одного из двух наборов инструкций в зависимости от значения некоторого булевского выражения.
3. ***Цикл (iteration (loop))*** — исполнение набора инструкций до смены значения некоторого логического выражения.

Под «набором инструкций» выше понимается одна инструкция исполнителя или одна из трёх указанных конструкций. Путём комбинирования одинаковых конструкций можно получить последовательности из нескольких инструкций, ветвления с произвольным числом веток и вложенные циклы, а путём комбинирования разных — любые более сложные алгоритмы. Парадигма программирования, предписывающая запись программы в императивном стиле с использованием только структурных конструкций, называется ***структурной (structured)*** парадигмой программирования. Блок-схемы, соответствующие конструкциям структурного программирования, приведены на рис. 1.1.

Процедурная (procedural) парадигма программирования является дальнейшей попыткой внести упорядоченность в императивную. Языки, ей соответствующие, содержат возможность выделения групп инструкций в именованные сущности, называемые ***подпрограммами (subroutine)***. Фундаментальным понятием этой парадигмы является ***вызов подпрограммы (subroutine call)*** — специальная инструкция, приводящая к исполнению всех инструкций,

соответствующих указанной подпрограмме так, словно это одна инструкция исполнителя. В зависимости от языка подпрограммы могут носить имя процедур, методов и/или функций (не в математическом смысле). Языки процедурной парадигмы программирования также предусматривают механизм обмена данными между вызывающей и вызываемой процедурами с помощью входных и выходных параметров, это позволяет изменять поведение процедуры при каждом её вызове и получать от неё результаты выполнения. Подпрограммы могут вызывать другие подпрограммы, образуя сложные вложенные структуры передачи управления.

В процессе обработки данных программами процедурная парадигма рассматривает в первую очередь структуру алгоритма. ***Объектно-ориентированная (object-oriented)*** парадигма программирования связывает части алгоритма и обрабатываемые ими данные в понятие ***объекта (object)***. Программа в объектно-ориентированной парадигме есть набор взаимодействующих друг с другом объектов. Считается, что это позволяет уменьшить семантическое различие между сущностями предметной области решаемой задачи и языка, на котором разрабатывается её решение.

Модульная (modular) парадигма программирования подразумевает разбиение программы на максимально независимые составные части. Такое ***разделение ответственности (separation of concerns)*** упрощает разработку и поддержку сложных программных систем. В модульной программе имеются логические границы между модулями — частями программы, которые определяют ***интерфейсы (interface)*** — элементы программы, через которые происходит их взаимодействие с внешней по отношению к каждому из них средой. Что конкретно представляет собой интерфейс зависит от конкретного языка программирования и других реализуемых в нём парадигм, но в любом случае это является программной реализацией некоторого контракта, т.е. набора правил по взаимодействию модуля с пользователями предоставляемых им сервисов. В данной парадигме каждый модуль является ***чёрным ящиком (black box)***: известно его поведение, видимое снаружи и определяемое его интерфейсом, но неизвестно внутреннее устройство. Такая структура позволяет менять внутреннюю реализацию без потери совместимости с внешними пользователями интерфейса, если его контракт продолжает соблюдаться. Соответствующий принцип ***скрытия информации (information hiding)*** может проявляться в языке программирования не только в виде самостоятельной структуры-модуля, но и как часть более мелких элементов.

Декларативная (declarative) парадигма программирования радикально отличается от императивной, настолько, что часто определяется как парадигма программирования, свободная от описания ***порядка выполнения (control flow)*** — последовательности действий, задаваемых алгоритмом в императивной парадигме. В связи с отсутствием явного алгоритма, работающего над некоторым объёмом данных, языки этой парадигмы часто отстраняются частично или полностью от понятия ***побочного эффекта (side effect)*** — взаимодействия одного элемента программы с другими, за исключением создания результата для того элемента, который привёл к выполнению данного. Языки декларативной парадигмы часто опираются в своём построении на ту или иную математическую концепцию, и потому достаточно разнообразны, особенно по сравнению с относительно похожими друг на друга языками императивной парадигмы. Характерными для данных языков являются специализация и описание условий задачи в терминах предметной области, многие из них являются представителями языков четвёртого и пятого поколения.

Выделим в декларативной парадигме программирования две наиболее известные: функциональную и логическую. ***Функциональная (functional)*** парадигма программирования рассматривает программу как набор функций в математическом (в отличие от императивного подхода) смысле, а процесс выполнения программы — как вычисление некоторого значения. ***Чистая (pure)*** функция зависит только от своих аргументов и в процессе вычисления не вносит никаких изменений в среду выполнения программы. Функциональные языки в той или иной мере стремятся к максимальной чистоте. Данный подход имеет положительные характеристики в части удобства реализации в современных высокопроизводительных системах, но требует решения вопроса о представлении вычислений, которые нельзя избавить от побочных эффектов по определению, в первую очередь, задачи обмена информацией с внешней средой. ***Логическая (logic)*** парадигма программирования основана на аппарате математической логики. Выполнение программы, являющейся набором логических утверждений в рамках этой парадигмы, сводится к попытке доказательства некоторого утверждения на основе имеющихся фактов и правил вывода.

Мета-программирование (metaprogramming) является вспомогательной парадигмой, рассматривающей программы, оперирующие другими программами, в первую очередь в качестве результатов своей работы. В рамках этой парадигмы строятся основные инструментальные средства программирования — генераторы кода и трансляторы. Средства ***рефлексивной (re-***

flection) парадигмы включают языки, имеющие возможность исследования и модификации программой самой себя в процессе выполнения. Эти средства полезны в особо оптимизированных и/или высокоуровнёвых системах и в некоторых специальных применениях.

Из-за замедления темпов роста производительности отдельной вычислительной единицы в последние годы всё большую популярность набирает параллельное программирование. Языки, содержащие встроенные средства обеспечения параллельных вычислений, относятся к ***параллельной (parallel)*** парадигме программирования. Эти языки позволяют ускорить разработку высокопроизводительных приложений.

1.4. Инструментальные средства

Рассмотрим инструменты, имеющиеся в распоряжении программиста, позволяющие ему выполнять свою работу и делать это эффективно.

1.4.1. Трансляторы

Поскольку времена, когда программирование в машинных кодах было единственным возможным способом, прошли, прежде чем программа на том или ином языке станет исполняться аппаратурой, её необходимо перевести в соответствующий ей машинный код. Этот процесс перевода может различаться по времени, когда происходит, и по объёму фрагментов программы, подвергающихся переводу, в зависимости от устройства программной среды, в которой происходит выполнение программы. Кроме этого, перевод может осуществляться не только в машинный код, но и между другими видами языков. В данном разделе под «языком высокого уровня» (ЯВУ) будем понимать машинно-независимые языки и языки более высоких поколений. Будем считать, что среди них тоже возможно установление отношений по уровню развитости: более или менее относительно высокого уровня. Рассмотрим сначала процесс перевода программ, отдельный от процесса их выполнения.

Транслятор (translator) — инструментальное средство, осуществляющее перевод программы с одного языка программирования на другой с сохранением семантики. Это наиболее общий термин — в таблице 1.1 показаны употребляемые термины для трансляторов между определёнными уровнями языков. Под «ЯВУ↓» в данной таблице подразумевается язык высокого уровня, но относительно более низкого, чем рассматриваемый.

<table>
<tr><th>↗</th><th>ЯВУ</th><th>ЯВУ↓</th><th>Ассемблер</th><th>Машинный/байт-код</th></tr>
<tr><td>ЯВУ</td><td>транслятор</td><td colspan="3" rowspan="2">компилятор</td></tr>
<tr><td>ЯВУ↓</td><td></td><td>транслятор</td></tr>
<tr><td>Ассемблер</td><td colspan="3" rowspan="2"></td><td>ассемблер</td></tr>
<tr><td>(Байт-)код</td><td colspan="2">декомпилятор</td><td>дизассемблер</td></tr>
</table>

Таблица 1.1: Названия трансляторов

Когда речь идёт о переводе между языками примерно одного высокого уровня, пользуются общим термином «транслятор». Такая ситуация встречается редко.

Компилятором (compiler) называют транслятор, понижающий уровень представления программы. Частный случай компилятора, выполняющего преобразование языка ассемблера в соответствующий машинный код, называется ***ассемблером (assembler)***. Средство, осуществляющее обратное преобразование, называется ***дизассемблером (disassembler)***. Инструменты, пытающиеся приближённо восстановить текст на языке высокого уровня по соответствующему ему машинному коду, называют ***декомпиляторами (decompiler)***. Остальные направления преобразования используются редко.

Вместо получения из того или иного языка полного машинного кода, предназначенного для непосредственного исполнения на конкретном процессоре, возможна и другая стратегия выполнения: специальное средство считывает текст программы и исполняет его относительно небольшими частями, обычно по одной инструкции. Такое средство называется ***интерпретатором (interpreter)***. Сам процессор с некоторыми допущениями можно рассматривать как аппаратный интерпретатор машинного кода. Простые интерпретаторы обладают меньшей производительностью, но проще в реализации сами и облегчают реализацию некоторых динамических языковых средств. С другой стороны, скомпилированные программы не требуют наличия интерпретатора для своей работы.

Это историческое деление способов выполнения программы на компилируемые и интерпретируемые, а также связанные с этим достоинства и недостатки в настоящее время является весьма размытыми. Многие среды предполагают в качестве итога работы компилятора не машинный код для конкретной аппаратной платформы, а код для некоторой абстрактной машины. Его называют ***байт-кодом (bytecode)*** или ***переносимым кодом (p-code (portable code))*** — кодом в наборе инструкций, специально разработанном для удобства реализации его программного интерпретатора. Это позволяет использовать этот код не на конкретной аппаратной архитектуре, а на любой, где имеются требуемые для его исполнения средства. Эти средства могут иметь вид, опять же, компилятора в настоящий машинный код, или интерпретатора байт-кода, который в данном случае носит название ***виртуальной машины (virtual machine)***. С точки зрения применения к нему терминологии, байт-код может рассматриваться как один из машинных кодов, например, существуют ассемблеры байт-кодов.

Эти два способа исполнения программ имеют следующие преимущества и недостатки с точки зрения возможных оптимизаций. Компилятор обладает значительно бóльшим временем на принятие решений по оптимизации кода, поскольку его работа происходит отдельно от самого выполнения программы. Интерпретатор в свою очередь связан жёсткими временными рамками, поскольку его работа и есть выполнение программы, но он обладает информацией о характере выполнения программы, которая компилятору не доступна. Чтобы воспользоваться обоими преимуществами, современные программные среды выполнения программ могут являться комбинированными компиляторами-интерпретаторами. Одна из традиционно используемых схем такова:

- Программа на языке высокого уровня компилируется в байт-код. Соответственно, на разбор синтаксиса программы во время её выполнения время затрачиваться не будет. К байт-коду применяются простые и однозначно полезные оптимизации.
- Выполнение программы осуществляется в виртуальной машине. Параллельно с её выполнением собирается статистическая информация о «горячих» местах в программе, на выполнение которых затрачивается бóльшая часть времени.
- Выявленные критичные ко времени выполнения фрагменты компилируются параллельно с выполнением программы, учитывая характеристики их выполнения. В дальнейшем выполнение этих фрагментов осуществляется не интерпретацией, а запуском откомпилированных и оптимизированных фрагментов.
- Оптимизированные фрагменты могут быть сохранены, чтобы не тратить время на их компиляцию при следующих запусках программы.

Такие системы, осуществляющие компиляцию во время или непосредственно перед исполнением программы или её частей, называют ***JIT-компиляторами (JIT (Just In Time)-compiler)***.

Помимо рассмотренной возможны и другие гибридные схемы, размывающие границы между этапами трансляции и выполнения программ.

1.4.2. Другие инструментальные средства

Трансляторы являются одними из важнейших, но далеко не единственными средствами, обеспечивающими разработку программ.

Исторически ограниченными ресурсами компьютеров обусловлено возникновение компоновщиков. На заре развития вычислительной техники начали возникать ситуации, в которых для транслирования одной большой программы оказывалось недостаточно памяти. Проблему решили путём разбиения программы на несколько частей. Результат работы транслятора над отдельной частью не является готовой к выполнению программой. Сборку оттранслированных частей программы в единое целое и осуществляет ***компоновщик (linker)***, также называемый ***редактором связей***. Помимо решения указанной проблемы, использование компоновщика вписывается в модульную парадигму программирования и способствует повторному использованию кода — единожды оттранслированный модуль может быть затем включён в несколько программ без повторной трансляции. Использование компоновщика в том числе позволяет писать разные части программы на разных языках.

В общем случае процесс создания итоговой программы включает в себя множество вызовов различных трансляторов и генераторов кода. В программе, состоящий из множества модулей, встаёт задача построения минимального достаточного набора команд, необходимого для обновления готовой программы, по набору изменённых модулей, поскольку выполнение полной процедуры сборки при любом изменении в больших проектах слишком накладно. Решение

этой задачи исходя из описанных явно или вычисленных автоматически зависимостей каждого модуля, хранимого обычно в отдельном файле на диске, включая параллельное по возможности исполнение требуемых команд, осуществляют ***системы сборки (build system)***.

Для отслеживания состояния программы во время её работы с целью нахождения ошибок и других проблем предназначены ***отладчики (debugger)***. Одним из основных свойств отладчика является возможность «заморозить» выполнение программы при возникновении ошибки или по команде программиста, чтобы изучить её состояние в определённый момент времени. При этом часто возможен пошаговый режим выполнения программы, при котором программа останавливается после каждой выполненной строки или инструкции — это позволяет следить за работой программы в удобном человеку темпе выполнения.

Существуют различные средства ***инструментирования (instrumentation)*** кода, осуществляющие изменение программы при трансляции таким образом, что она в процессе своего выполнения, помимо решения заданной программистом задачи, осуществляет и другую деятельность. В качестве примера целей инструментирования можно привести сбор статистической информации о времени выполнения для более точной оптимизации программы трансляторами и обнаружение дополнительных ошибок и проблем безопасности в программе сверх того, что гарантируется самим языком.

Для удобства доступа программиста ко всем инструментам, находящимся в его распоряжении, имеются ***интегрированные среды разработки (integrated development environment (IDE))***. Это программные системы, основной которых являются специализированные текстовые редакторы, которые интегрируют в одном интерфейсе возможности всех перечисленных инструментальных средств. Используя полный объём информации о разрабатываемой программе, они способны обеспечить синтаксическую и семантическую подсветку исходного кода, средства автодополнения при наборе кода, визуализировать в графической форме структуры программы и её отдельных компонентов, объединить процессы отладки и редактирования кода и предоставить другие продвинутые возможности.

Перечисленными средствами инструментарий программиста, разумеется, не ограничен. В зависимости от масштабов разрабатываемого проекта программисты часто разрабатывают вспомогательные программы для упрощения разработки основных требуемых программ.

Глава 2

Основные понятия архитектуры ЭВМ

Хотя эта книга и посвящена машинно-независимому языку, его низкоуровнёвый характер позволяет детально рассмотреть, как те или иные конструкции соответствует машинному коду и данным в памяти ЭВМ. Более того, обойтись без этого попросту невозможно, потому что именно эта связь легла в основу этого языка — многие его понятия напрямую следуют из свойств аппаратных средств, хотя об этом не всегда и явно упоминают. Как уже говорилось, автор считает возможность рассмотрения языка на всех уровнях его существования весьма полезным.

Спускаясь на ступеньку ниже абстракций рассматриваемых языков, придётся выбрать конкретную аппаратную платформу для которой приводить примеры. В данной книге будут рассматриваться платформа x86 в 32-битном защищённом режиме (далее просто x86) как одна из наиболее популярных в настоящее время — её 64-битная версия лежит в основе всех современных ПК. В целях упрощения мы ограничимся рассмотрением её устаревающей, но пока не потерявшей полностью актуальность 32-битной версии. Мы остановимся только на важных для обсуждения языков программирования фактах в приближении, достаточном для передачи требуемой идеи, оставив точное и детальное рассмотрение соответствующим дисциплинам.

Начнём обзор архитектуры с устройства памяти. Как известно из школьного курса информатики, ***оперативная память (random access memory, RAM)*** является энергозависимой памятью, предназначенной для хранения программ и данных, с которыми в настоящее время выполняется работа. Поскольку современная аппаратная база использует двоичную логику, минимальной единицей хранимой информации является ***бит (bit)*** — единица информации, принимающее одну из двух значений, обычно обозначаемых 0 и 1. Бит является слишком мелкой единицей, чтобы к каждому из них предоставлялся непосредственный доступ, поэтому минимальной ***адресуемой (addressable)*** единицей памяти является ***байт (byte)***, состоящий в большинстве современных систем из 8 бит. Под «адресуемостью» подразумевается наличие у единицы памяти некоторого имени, позволяющего получить к ней доступ. В большинстве современных ОС, применяемых на ПК, каждая программа имеют собственное ***плоское адресное пространство (flat address space)***. Это означает, что все байты некоторого виртуального объёма памяти, потенциально доступного программе, пронумерованы по порядку, начиная с 0, и это число — ***адрес (address)*** — и является «именем», по которому к нему можно получить доступ.

Для большинства вычислений байт тоже слишком маленькая единица. Выполнение инструкций неспециализированных программ осуществляет ***центральный процессор (ЦП) (central processing unit (CPU))***, часто просто «процессор». У большинства процессоров существует некоторый фиксированный объём данных — машинное ***слово (word)***, с которым им наиболее удобно производить вычисления. Размер этого объёма данных в битах называется ***разрядностью (bitness)*** процессора. Помимо ограничений на числовые операции, этот параметр определяет количество возможных адресов памяти, с которыми работает данный процессор: 32 бита для рассматриваемой архитектуры x86. Таким образом, каждая программа на данной архитектуре имеет в своём распоряжении 2^{32} байт = 4 Гбайта адресного пространства. Благодаря механизмам виртуальной памяти операционной системы, этот объём никак не связан с реальным объёмом ОЗУ, установленном в машине, который может быть как меньше, так и больше. Именно поэтому этот объём называется ***пространством***, а не памятью как

таковой — далеко не всему ему соответствует какая-либо реальная память.

Помимо оперативной памяти, сам центральный процессор содержит очень малый объём памяти, называемый ***регистровым файлом (register file)***. Он состоит из ***регистров (register)***, большинство из которых имеет размер, соответствующий разрядности процессора. Регистры, в отличие от оперативной памяти, не имеют адресов, вместо этого для обращения к ним используются специальные имена. Наличие регистров обуславливается несколькими причинами:

- Многие регистры играют специальную роль, храня данные, относящиеся к текущему состоянию процессора и его режимам работы или играют особую роль в исполнение некоторых команд — эти данные не могут хранится в ОЗУ.
- Поскольку регистры физически расположены непосредственно в составе процессора, скорость доступа к ним значительно выше, чем к оперативной памяти. Более того, для современных процессоров эта разница в скорости настолько велика, что для обеспечения работы процессора без значительных простоев, требуется применение сложной многоуровневой дополнительной памяти, также обычно размещённой в самом процессоре — ***кеша (cache)***.
- Рассматриваемый набор инструкций x86 не позволяет использовать в качестве операндов более одного адреса памяти, а также имеет множество других ограничений по использованию оперативной памяти в качестве источника или приёмника данных. Это означает, что для многих операций, особенно использующих более одного аргумента, требуется предварительно загружать операнды в регистры, выполнять там над ними операции, и только после этого записывать результат назад в ОЗУ отдельной инструкцией. Это следствие исторического развития архитектуры, изначально имевшей очень ограниченный набор команд и определявшей специальную роль большинства регистров.

Обратим внимание, что все современные архитектуры являются архитектурами с ***хранимой программой (stored program)*** — программа, также как и данные, хранится в памяти компьютера и может быть изменена с той же лёгкостью, что и данные. В рамках архитектуры фон Неймана для хранения программ и данных используется одна и та же память, что справедливо и для архитектуры x86. Один из регистров процессора `EIP` (Extended Instruction Pointer) хранит адрес инструкции, следующей за исполняемой в настоящий момент. Инструкции архитектуры x86 могут иметь различную длину в байтах, после прочтения и декодирования очередной из них значение этого регистра увеличивается на её длину, чем обеспечивается последовательное выполнение команд. Специальные команды могут изменять значение этого регистра по другим правилам, обеспечивая возможность произвольной организации передачи управления.

Так же как и с памятью, современные многозадачные операционные системы создают для каждой программы иллюзию того, что она обладает собственным процессором, не зависящим от других выполняемых задач. Мы рассмотрим остальные регистры процессора и конкретные команды по мере надобности. Для желающих подробнее ознакомится с языком ассемблера автор предлагает воспользоваться книгой [9], где многие из вопросов, рассматриваемых в данном тексте, изложены с его применением. В этой книге мы тоже будем пользоваться синтаксисом ассемблера NASM, базирующемся на синтаксисе Intel.

Схема работы процессора с учётом наших текущих представлений показана на рис. 2.1: ***арифметическо-логическое устройство (АЛУ) (arithmetic logic unit (ALU))*** — часть центрального процессора, ответственная за выполнение инструкций и вычислений, — выполняет считанную из памяти команду. При этом происходит изменение значений регистров и осуществляются дополнительные операции чтения/записи памяти, когда это необходимо.

Вычислительные возможности центрального процессора и объём оперативной памяти являются двумя основными ресурсами, о которых идёт речь при разработке оптимальных алгоритмов и их реализаций на конкретном языке программирования. Работа с этими двумя сущностями достаточно прозрачно наблюдается с точки зрения самой программы, в то время как работа с остальными устройствами обычно делегируется средствам операционной системы.

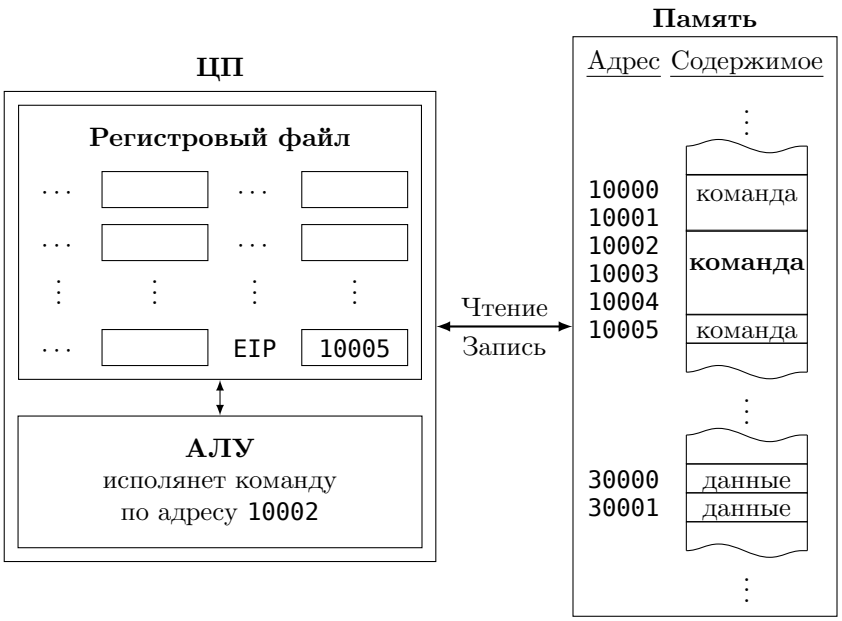


Рис. 2.1: Первое приближение схемы работы процессора x86

Часть II

Язык программирования C++

Глава 3

Обзор и основные понятия языка C++

Прежде чем говорить о языке C++, расскажем при каких обстоятельствах был создан его предшественник.

Язык C был создан в 1969–1973 годах Деннисом Ритчи, перед которым стояла задача создания языка более высокого уровня, чем машинозависимый ассемблер, чтобы обеспечить свойство ***переносимости (portability)*** для разрабатываемой в то время операционной системы UNIX. Под переносимостью подразумевается минимальность необходимых действий, чтобы обеспечить работу программной системы в новой среде. Эти цели удалось выполнить даже в большей мере, чем этого ожидал автор.

Язык C лежит в основе всего современного системного программирования в чистом виде или в лице производных от него языков. Огромная масса кода как системного, так и прикладного уровня написана на нём. Именно поэтому знание теоретических и практических основ работы данного языка является ключевым в понимания работы языков и систем любого более высокого уровня — мы познакомимся с ними в рамках языка C++, являющегося наиболее значимым из его последователей.

Перечислим основные характеристики языка C:

- Язык C относительно низкого уровня. Иногда его даже называют «переносимым ассемблером». Бóльшая часть конструкций языка соответствует относительно небольшому количеству инструкций процессора. Глядя на текст программы, довольно легко понять, что на самом деле происходит при её работе: число «сюрпризов» — неявных действий — совершаемых без указания программиста, мало по сравнению с языками более высокого уровня. С другой стороны, «КПД» программиста в смысле «полезной работы, совершаемой программой на единицу объёма исходного кода», ниже, чем у языков более высокого уровня.
- Парадигма программирования, напрямую поддерживаемая языком C — процедурная. Программист явно описывает последовательность действий, т.е. алгоритм, который по его мнению приведёт к необходимому результату. Все необходимые конструкции для поддержки структурной парадигмы программирования также присутствуют.
- Основной особенностью C с практической точки зрения является наличие т.н. ***сырых (raw)*** указателей, позволяющих напрямую манипулировать адресами памяти, в его системе типов. Прямая работа с памятью является очень мощным инструментом, который может эффективно использоваться как для решения, так и для создания наиболее сложных проблем.
- Язык C создавался в то время, когда считалось, что каждый программист знает, что делает. В отличие от современных высокоуровнёвых языков, осуществляющих контроль за действиями программиста, C не делает ничего, о чём его явно не попросили. Это позволяет достичь максимальной производительности — сам язык не делает ничего лишнего, но это также является источником труднообнаружимых ошибок, особенно, если программист переходит на C с другого языка или это его первый язык.

Все эти качества возникли не случайно, а в результате разработки языка для решения указанной задачи. С этой точки зрения язык C относится к языкам, который изначально разрабатывали практики, а не теоретики.

В 1979 году Бьёрн Страуструп начал разработку языка, тогда носившего имя «C with Classes» — «C с классами», как расширение C. Стимулом к этому послужило полученное

на собственном опыте программирования понимание того, что языку C требуется лучше поддерживать разработку крупных проектов. В то время существовали языки, более удобные для это, но они сильно уступали C по производительности, поэтому Страуструп решил собрать все преимущества в одном языке. Следующие принципы легли в основу построения нового языка:

- Совместимость с существующей программной инфраструктурой, в первую очередь языком C и библиотеками, написанными на нём.
- Наличие средств разделения и повторного использования кода (к сожалению, наследие языка C до сих пор не позволяет решить эту задачу полноценно).
- Более строгая система типов, чем в C, чтобы позволить компилятору находить большее число ошибок автоматически. Однако если программист знает что делает, у него должна быть возможность затребовать любые опасные действия.
- Язык должен оставаться по возможности прозрачным, в первую очередь, он не должен затрачивать ресурсы на возможности, не используемые программистом. Однако с этой точки зрения он всё же не столь прозрачен как C и требует большей внимательности программиста. В данном случае прозрачностью пожертвовано в пользу автоматизации.
- Поддержка множества парадигм и стилей программирования и возможность применять их вместе. Важнее предоставить программисту различные возможности на выбор, даже если это усложнит язык.

  Язык C++ поддерживает несколько парадигм.

- ***Структурное процедурное программирование***. Язык C++ включает в себя почти все возможности языка C, и потому поддерживает процедурную и структурную парадигмы в полной мере. Более того, они лежит в основе поддержки всех остальных парадигм.
- ***Объектно-ориентированное программирование (ООП)***. ООП является подходом к разделению ответственности, выбирающим в качестве отделяемых друг от друга единиц ***объекты*** (этот термин используется в абстрактном смысле а не в смысле, который будет введён в самом языке C++). Основной идеей ООП является создание в рамках программы системы типов объектов, отражающих предметную область решаемой задачи, что позволит приблизить запись алгоритма решения на языке программирования к естественному его восприятию специалистом в соответствующей области. Разумеется, не все реальные объекты осмысленно представлять в виде программных сущностей, кроме того в реальных программах часто имеется множество объектов с сущностями предметной области не связанными, которые выполняют различные служебные функции. В качестве примеров типов, удовлетворяющих этим идеям, можно привести «многочлен с целыми коэффициентами», «модель торгово-сервисного предприятия», «соединение с удалённой базой данных, позволяющее выполнять запросы к ней». Язык C++ обеспечивает взаимодействие объектов друг с другом за счёт установления специальной синтаксической и семантической связи функций (элементов процедурной парадигмы) и структур данных, которые они обрабатывают, в рамках классовых типов.
- ***Обобщённое программирование***. Обобщённое программирование состоит в отделении алгоритмов от конкретных типов данных, с которыми они работают. Синтаксическим средством реализации этой парадигмы в языке C++ являются шаблоны.

  Неожиданно для самих разработчиков языка шаблоны послужили основой отдельного направления ***шаблонного метапрограммирования (template metaprogramming, TMP)***. В современном C++ это мощное средство исполнения кода на этапе трансляции, которое может использоваться для управления процессом оптимизации программы компилятором, построения в рамках синтаксиса C++ доменно-специфических языков и других продвинутых возможностей.
- ***Функциональное-программирование***. Язык C++ не отстаёт от моды в контексте повышения интереса к функциональным языкам, что происходит в развитии языков программирования в последнее время. Современные версии языка имеют поддержку лямбда-выражений, позволяющих частично воспользоваться преимуществами этой парадигмы программирования.

Помимо упомянутых в идеологии языка особенностей и поддержке перечисленных выше парадигм, включая всё, что унаследовано от языка C, отметим две следующих характерных черты языка C++: поддержку сложных иерархий классовых типов, включая множественное

наследование, и механизм исключений как средство обработки ошибок времени выполнения в программах.

В 1983 году промежуточная версия языка получило новое имя «C++» — «следующий после C». В 1985-ом выходит первая книга по языку C++, «The C++ Programming Language», которая используется в качестве основного описания языка в отсутствие какого-либо стандарта. Впоследствии язык был принят в качестве нескольких международных стандартов:

1. ISO/IEC 14882:1998 (C++98) — первый стандарт языка.
2. ISO/IEC 14882:2003 (C++03) — работа над ошибками первого стандарта, без существенных изменений. Поддерживается всеми современными компиляторами в полном объёме.
3. ISO/IEC TR 19768:2007 (C++ TR1) — расширения стандартной библиотеки, которые планировалось внедрить в следующей версии языка.
4. ISO/IEC 14882:2011 (C++11) — предпоследний официальный международный стандарт языка на момент написания этого текста. За 8 лет с момента выпуска предыдущей версии язык был значительно расширен, что во многом повлияло на стиль написания программ. Этот стандарт поддерживает большинство современных компиляторов. Изложение языка C++ с новой точки зрения, которая стала формироваться, начиная с этого стандарта, является одной из основных причин создания данного текста. Этот стандарт иногда также называют C++0x — в процессе ожидания окончания работы над ним требовалось неформальное имя, и предполагалось, что он выйдет до 2010-го года.
5. ISO/IEC 14882:2014(E) (C++14) — последний официальный стандарт языка C++, содержащий в основном исправления и дополнения к стандарту 2011-го года, позволяющие в полной мере воспользоваться его возможностями. Неформальное рабочее обозначение — C++1y.
6. Следующая версия стандарта, работа над которой близится к завершению. Неформальное обозначение — C++1z, ожидается в 2017-ом году.

Мы будем рассматривать самую свежую версию языка, включая возможности, которые планируются к введению в будущем официальном стандарте. Вместо официального текста стандарта, являющегося платным, можно использовать последний рабочий черновик, доступный в виде документа N4618 [1]. Этот документ является рабочей версией нового стандарта, выход которого запланирован на 2017-ый год.

Помимо новой версии стандарта, в настоящее время параллельно разрабатываются множественные расширения языка отдельными рабочими группами комитета стандартизации, которые будут выходить независимо от базового стандарта. Переход к такой форме вызван расширением объёма стандартной библиотеки языка и необходимостью ускорить развитие стандартизированной базы языка.

3.1. Стандарт ISO/IEC 14882:2014(E)

Упомянутый в заглавии данного подраздела международный стандарт является основным документом, определяющим отношения между программистами и создателями сред, в которых могут транслироваться и исполнятся программы на языке C++.

Помимо описания конструкций языка, стандарт предписывает наличие ***стандартной библиотеки*** — набора средств, самих являющихся конструкциями языка C++, обеспечивающих базовую функциональность программ. Её наличие необходимо для соответствия стандарту, и её средства автоматически доступны для всех программ на языке C++. В её состав входят расширения базовых языковых конструкций, средства ввода/вывода, обеспечивающие взаимодействие программы с другими компонентами системы, математические и другие утилитарные функции, а также основные структуры данных и алгоритмы над ними. Почти вся стандартная библиотека языка C входит в стандартную библиотеку C++ в неизменном виде. Отличительной особенностью языка C++ является тот факт, что многие конструкции, которые обычно входят в описание самого языка программирования, в данном языке реализованы в виде части стандартной библиотеки.

Под ***реализацией (implementation)*** стандарта будем понимать программно-аппаратную систему, способную произвести выполнение программы, заданной на языке C++. С точки зрения стандарта реализация состоит из ***среды трансляции (translation environment)*** и ***среды выполнения (execution environment)***. Поскольку подавляющее большинство реализаций языка C++ содержат компиляторы, а не интерпретаторы, термин «компилятор» будет

в дальнейшем использоваться вместо более общего «транслятор». Помимо конкретных экземпляров компилятора, отвечающего за обработку языковых средств, и стандартной библиотеки языка C++, также являющейся частью стандарта, к реализации можно отнести операционную систему и архитектуру ЭВМ на которой работает компилятор, и на которых предполагается выполнение программы. В большинстве случаев результат работы компилятора предназначен для выполнения на той же системе, что и сам компилятор, так что характеристики сред трансляции и выполнения обычно совпадают.

> Компилятор, результат работы которого предназначен для среды выполнения, отличной от среды выполнения самого компилятора, называется ***кросс-компилятором (cross-compiler)***, а его использование — кросс-компиляцией. Кросс-компиляторы используются для первоначальной сборки окружения новой среды выполнения или больших проектов, предназначенных для слабых систем, на которых использование обычного для них компилятора слишком медленно.

Основной смысл стандарта — установить требования на соответствующую ему программу и реализацию языка C++, в которой она выполняется, таким образом, чтобы видимый (в специальном смысле) результат выполнения программы был предсказуем.

Основные конструкции языка при использовании их в соответствии со стандартом обычно имеют ***определённое поведение (defined behavior)*** — если применить в программе на языке C++ указанную конструкцию, результат на любой корректной реализации будет в точности таким, как это описано в стандарте. ***Неуточняемым поведением (unspecified behavior)*** называют поведение языковых средств, для которых стандарт определяет множество вариантов поведения, но не указывает способ выбора конкретного из них корректной реализацией. Этот тип поведения используется в случаях, когда конкретный вариант обычно не важен для программиста и позволяет упростить соответствующую стандарту реализацию. В случаях, когда стандарт требует от конкретной реализации однозначного поведения, не указанного явно в самом стандарте, поведение называют ***зависящим от реализации (implementation-defined behavior)***. Каждая реализация должна документировать своё поведение для всех пунктов стандарта, где встречается такое описание. Наконец, реальную угрозу несут действия программы, имеющие ***неопределённое поведение (undefined behavior, UB)*** — никаких ограничений на то, что при этом происходит, стандарт не накладывает.

Причин наличия нестрогих описаний поведения в языке C++ две. Во-первых, это позволяет стандарту не накладывать жёстких ограничений на реализации там, где в этом нет необходимости, чтобы сохранить их простоту. Например, реализация одного из вариантов может быть удобной и эффективной только на некоторых платформах, а другого — на других. В некоторых случаях неуточняемое поведение даётся для случаев, где все варианты равнозначны для большинства программ и нет смысла требовать одного конкретного поведения от всех реализаций. Во-вторых, как уже было сказано, язык C++ старается не ограничивать действия программиста, надеясь на его компетентность. В языке полностью отсутствуют средства контроля корректности всех совершаемых программой действий во время выполнения, а доказать, что для всех возможных случаев та или иная операция имеет осмысленный результат на этапе компиляции в большинстве случаев невозможно или слишком сложно. Поэтому стандарт содержит большое количество оговорок вида «если программа пытается совершить ту или иную некорректную операцию, то её поведение не определено». К неопределённому поведению также ведут нарушения требований стандарта, содержащие слова ***должен (shall)*** или ***не должен (shall not)***, а также любые другие действия, о которых в стандарте ничего не сказано явно.

Неопределённое поведение является одной из отличительных черт языка C++, унаследованной от C, и должно требовать к себе повышенного внимания, особенно у начинающих программистов. Отметим, что поскольку нет никаких ограничений на поведение программы при наличии в ней неопределённого поведения, не противоречат стандарту и реально встречаются на практике следующие варианты:

- Ошибочная операция не имеет никакого эффекта. Часто это является результатом оптимизаций компилятора, который строит программу так, что в случае возникновения неопределённого поведения она ничего не делает. В корректной программе его быть не может, и компилятор делает самое простое, что может — ничего.
- Программа «сходит с ума» и начинает выполнять необъяснимые действия, причём не сразу после выполнения ошибочной операции, а через полчаса работы (после одной операции с неопределённым поведением, всё дальнейшее *и даже прошлое)* поведение программы превращается в неопределённое поведение). Из-за оптимизаций компилятора случаи, когда

работа программы искажается ещё до формального выполнения конструкции, содержащей неопределённое поведение, могут наблюдаться в действительности. Формально, неопределённое поведение имеет право делать всё что угодно, в том числе изобретать машину времени и отправляться портить программу в прошлое до времени исполнения команды, её содержащей. Этот факт следует учитывать при анализе ошибочного поведения программ и не удивляться.

- Компьютер отращивает пасть и щупальца и съедает вашего котёнка. Масштаб неограниченных стандартом последствий зачастую превышает фантазию даже опытного программиста, поэтому автор не постесняется привести этот пункт в числе встречающихся в действительности.
- Программа делает всё в точности так, как этого изначально хотел программист, не заметивший свою ошибку... по крайней мере, первые несколько лет её эксплуатации. Ошибки отличаются везением, когда дело доходит до маскировки собственного существования.

Как видно из приведённых примеров, ошибки, связанные с неопределённым поведением, особенно трудны в отладке, поскольку их последствия могут обнаруживаться на большом расстоянии от их реального местоположения, как в тексте программы, так и по времени проявления, а могут и не проявляться вовсе. При этом в большинстве случаев никаких диагностических сообщений от компилятора программист не получает. Для успешной борьбы с этим классом ошибок, необходимо всегда быть в курсе и учитывать ограничения, накладываемые используемыми конструкциями языка, и вводить в программу проверки там, где это требуется. Язык C++ ничего не проверяет сам, то есть никогда не будет замедлять вашу программу не затребованными явно действиями, поэтому все необходимые проверки остаются на совести программиста. Это и есть основной аргумент в пользу того, что чёткое понимание происходящего критически важно для написания корректных программ, особенно на языках относительно низкого уровня. Также в процессе обучения следует морально подготовиться к факту, что внешне корректно работающие программы могут содержать фатальные ошибки.

3.2. Обзор процесса трансляции

Прежде чем приступать к изучению деталей языка, рассмотрим весь процесс трансляции программы в целом.

Программа на языке C++ состоит из одного или нескольких текстов, хранимых в ***файлах исходного текста (source files)***. Каждый из них проходит ***предварительную обработку (preprocessing)***, которая имеет конечной целью представление модифицированного исходного текста в виде последовательности ***токенов (token)*** — минимальных неделимых единиц языка. Полученная ***единица трансляции (translation unit)*** транслируется согласно описанию языка, результатом этого процесса является ***объектный файл (object file)***. Наконец, все объектные файлы, из которых состоит программа, а также необходимые ***библиотеки (library)*** (в первом приближении — заранее оттранслированные наборы объектных файлов) объединяются для получения ***образа программы (program image)*** — полного объёма информации, необходимого для выполнения программы в среде выполнения.

3.3. Лексический состав языка

Файлы исходного текста программы при считывании отображаются в ***базовый набор символов исходного текста (basic source character set)***:

- Латинские буквы: `A--Z`, `a--z`.
- Арабские цифры: `0--9`.
- Символы:
  `_ { } [ ] # ( ) < > % : ; . ? * + - / ^& | ~! = , \ " '`
- Пробельные символы: пробел (space), горизонтальная и вертикальная табуляции (horizontal/vertical tab), новая страница (form feed), перевод строки (new-line).

Все остальные символы преобразуются в специальные последовательности, состоящие только из символов базового набора. Как происходят эти отображения определяется реализацией. Вопросы, связанные с представлением текста в исходном коде программы и во время её работы, нетривиальны, мы рассмотрим их позднее в отдельном разделе. Пока автор

предлагает воздержаться от использование кириллицы и других символов, не входящих в перечисленные, в тексте программы, поскольку в этом случае могут возникать многие не объяснимые пока проблемы.

Минимальным набором символов среды выполнения является ***базовый набор символов среды выполнения (basic execution character set)***, который помимо символов базового набора символов исходного текста должен включать управляющие символы звонка (alert), возврата на один символ (backspace), возврата каретки (carriage return) и нулевой символ (null, не путать с символом цифры ноль!). Полный набор символов среды выполнения может быть шире указанного по усмотрению реализации.

Перед началом работы основной фазы трансляции единица трансляции претерпевает некоторые изменения, относящиеся к ***предварительной обработке (preprocessing)***. Часть транслятора (которая может быть и отдельной программой), осуществляющая эти изменения, так и называется — ***препроцессором (preprocessor)***. Её использование — ещё одна из особенностей, унаследованных C++ от языка C, и в современных программах следует минимизировать её использование. Мы рассмотрим связанные с ней вопросы далее по мере их возникновения, а пока можно не учитывать выполняемые на этом этапе действия.

После обработки препроцессором единица трансляции представляет собой последовательность токенов, относящихся к одной из пяти видов:

- ***Идентификаторы (identifier)***. Идентификаторы обозначают элементы программы. Это имена или части имён, которые программист даёт тем сущностям, с которыми работает. Идентификаторы синтаксически являются последовательностями букв, символов подчёркивания и арабских цифр, но не могут начинаться с цифр. Примеры: `i`, `calculateAverage`, `random_number_generator`, `_internal115`. Регистр в идентификаторах имеет значение: `man` и `MAN` — разные идентификаторы. Идентификаторы, содержащие два символа подчёркивания подряд или начинающиеся с символа подчёркивания, за которым следует заглавная буква, зарезервированы реализацией и в программах использоваться не должны.

- ***Ключевые слова (keyword)***. Ключевые слова синтаксически могли бы быть причислены к идентификаторам, но имеют фиксированный смысл и зарезервированы языком для своих нужд — использовать их в качестве имён элементов программ нельзя. В стандарте C++14 определены следующие ключевые слова:

```
alignas     do            new               this
alignof     double        noexcept          thread_local
asm         dynamic_cast  nullptr           throw
auto        else          operator          true
bool        enum          private           try
break       explicit      protected         typedef
case        export        public            typeid
catch       extern        register          typename
char        false         reinterpret_cast  union
char16_t    float         return            unsigned
char32_t    for           short             using
class       friend        signed            virtual
const       goto          sizeof            void
constexpr   if            static            volatile
const_cast  inline        static_assert     wchar_t
continue    int           static_cast       while
decltype    long          struct
default     mutable       switch
delete      namespace     template
```

- ***Операции и пунктуаторы (punctuator)***. Пунктуаторы используются в синтаксисе большинства конструкций как разделители. Операции обозначают требуемые в программах вычисления. Некоторые из них образованы не специальными знаками, а словами, таким образом, некоторые из ключевых слов, перечисленные выше, относят также и к операциям. Мы будем указывать случаи, когда речь идёт об операциях, особо, все остальные использования этих токенов являются пунктуаторами, при этом некоторые из них могут выступать в обоих ролях в зависимости от контекста. В языке определены следующие пунктуаторы и операции:

```
{     }       [       ]       #       ##      (       )
<:    :>      <%      %>      %:      %:%:    ;       :     ...
new   delete  ?       ::      .       .*
+     -       *       /       %       ^       &       |     ~
!     =       <       >       +=      -=      *=      /=    %=
^=    &=      |=      <<      >>      >>=     <<=     ==    !=
<=    >=      &&      ||      ++      --      ,       ->*   ->
and   and_eq  bitand  bitor   compl   not     not_eq
or    or_eq   xor     xor_eq
```

Хотя многие из этих токенов состоят из нескольких символов, с точки зрения языка они являются неделимыми элементами.

- ***Литералы (literals)***. Литералы используются в тексте программы для записи фиксированных значений. В языке C++ они делятся на целочисленные, символьные, с плавающей точкой, строковые, булевские, литерал указателя и пользовательские литералы. Примеры: `123`, `4.56e7`, `'x'`. Их синтаксис зависит от подтипа и будет рассмотрен отдельно.

> Два возможных идентификатора `override` и `final` в некоторых контекстах имеют специальный смысл и называются ***идентификаторами со специальным значением (identifiers with special meaning)***. В этих местах текста программы они имеют особый смысл, как и остальные ключевые слова, но в других могут использоваться как обычные идентификаторы. Делать это на практике не следует, чтобы не вносить путаницу.

Пробельные символы не входят в состав токенов (за исключением символьных и строковых литералов, внутри которых они имеют значение). Они являются разделителями и обязательны только там, где это влияет на разбиение на токены: оно происходит таким образом, что в очередной токен, который строится из последовательности символов, составляющих файл исходного текста, забирается максимально длинная последовательность символов, которая может являться очередным токеном.

Например, последовательность символов `a b` задаёт два токена: идентификатор `a` и идентификатор `b`. Она также может быть записана как `a  b` — число и вид пробельных символов значения не имеет. Последовательность символов `doremi` трактуется как один идентификатор, несмотря на то что его первые два символа составляют ключевое слово. Наконец, последовательность символов `for(;;)` разбирается на ключевое слово `for`, за которым следуют пунктураторы `(`, `;`,`;`,`)` — вокруг пунктуаторов пробелы не обязательны, поскольку в данном случае интерпретация однозначна. Последовательность символов `a+++++b` разбирается на токены `a`, `++`, `++`, `+` и `b`. Если ввести явные пробельные разделители можно получить другой результат: `a++ + ++b` разбирается на `a`, `++`, `+`, `++` и `b`. Таким образом, ***использование пробельных символов позволяет программисту выбрать форму записи программы, наиболее удобную для прочтения людьми***.

Другим элементом программы, содержимое которых интересно только людям, а не транслятору, являются ***комментарии (comment)***. Каждый комментарий на этапе предварительной обработки заменяется одним пробелом, поэтому может содержать произвольный текст. ***Комментарии служат для включения в текст программы пояснений о её устройстве для прочтения людьми.*** Хорошие комментарии объясняют не *что* делает код — это следует из самого синтаксиса языка, а *как* и *почему* он это делает, в случаях, когда это не очевидно — нет смысла комментировать каждую тривиальную строку кода. Комментарии также используются для написания формальной документации и других вставок в текст программы, не являющийся кодом на языке C++. Таким образом, комментарии являются основным и самым простым инструментом устранения прагматических недоразумений — одна строка комментария в качестве первой строки программы с описанием её назначения иногда говорит незнакомым с ней читателям больше, чем весь остальной её текст.

Комментарии имеет две формы: многострочные и однострочные. Многострочные комментарии начинаются с пары символов `/*`, а заканчиваются парой символов `*/`. Поскольку внутри комментариев особый смысл имеет только пара символов, заканчивающих его, такие комментарии не могут быть вложены друг в друга — первая же последовательность символов конца комментария завершит его, независимо от числа вхождений начальных символов комментария после первого. Однострочные комментарии начинаются парой символов `//` и продолжаются до конца строки.

Приведём пример:

```
1  /*************************
2   * Это "шапка" программы. *
```

```
3   *************************/
4  int/**/a; // Определение идентификатора объекта
5  /* Эти символы не имеют специального смысла:
6     /* /* // <-- поскольку они внутри комментария,
7     который кончается здесь --> */
```

Этот фрагмент программы содержит только три токена: ключевое слово `int`, идентификатор `a` и пунктуатор `;`.

3.4. Типизация

Одной из характеристик языка программирования является его подход к ***типизации (typing)***. Под ***типом (type)*** понимается свойство элемента программы, определяющее его допустимые значения, их представления и операции над ними. Всё множество ***фундаментальных (fundumental)*** (встроенных в язык) типов и способы определения новых типов образуют ***систему типов (type system)***.

Языки программирования могут быть классифицированы с нескольких точек зрения, связанных с понятием типизации:

- По времени, когда осуществляется проверка допустимости совершаемых программой операций к объектам в соответствии с их реальными типами: ***статическая (static)*** (во время трансляции) или ***динамическая (dynamic)*** (во время выполнения).
- По возможности использования значений одного типа в качестве других за счёт неявных (автоматических) преобразований типов: ***слабая (weak)*** (множество разнообразных неявных преобразований) и ***строгая (strong)*** (требуется строгое соответствие типов, бóльшая часть преобразований типов должна быть выполнена явно).
- По наличию или отсутствию ***механизма вывода типов (type inference)*** — автоматическому определению типов элементов программы в противоположность необходимости явного указания типов программистом.

Язык C++ можно классифицировать как имеющий статическую типизацию силы выше средней с элементами механизма вывода типов.

3.5. Обзор структуры программы

Запись математического алгоритма на естественном языке является последовательностью действий, оперирующих некоторыми величинами. Величины, в первую очередь не являющиеся временными результатами промежуточных вычислений, необходимо где-то хранить, для чего среда выполнения обладает некоторым объёмом памяти. Область памяти среды выполнения, содержимое которой может быть представлением некоторого значения, называется ***объектом (object)***. Это использование слова «объект» не имеет прямого отношения к объектно-ориентированной парадигме программирования. ***Значение (value)*** — это точный смысл содержимого объекта, которое определено только с точки зрения обладания объектом конкретного типа. Объект может использоваться для получения его значения — ***чтения (read)*** или ***модификации (modify)*** значения, хранящегося в нём. Модификацию объекта также называют ***записью (write)***. Операции чтения и записи вместе называются ***доступом (access)*** к объекту. Способность многократно воспроизводить значение, которое было записано в него последним, является основным свойством объекта. Значения могут существовать и независимо от объектов, например, как промежуточные значения в вычислениях, но в таком случае после однократного их использования в том контексте, где они были получены, они теряются безвозвратно. Такие значения тоже обязательно обладают типом.

С помощью токенов-операций можно записать последовательность вычислений над значениями. ***Выражением (expression)*** называют последовательность операций и операндов. Отдельно взятые значения, например, в виде литералов, являются простейшей формой выражений.

Более сложные выражения включают одну или несколько операций. В качестве входных данных для операций указываются ***операнды (operand)***. Как и в математике, ***арностью (arity)*** операции называют количество её операндов. В общем случае операция над n операндами называется n-арной, особые термины употребляются для одного (***унарная (unary)*** операция), двух (***бинарная (binary)*** операция) и трёх операндов (***тернарная (ternary)*** операция).

Результат вычисления, соответствующий операции, вновь является значением, которое может быть снова использовано в качестве операнда другой операции. Выражения, не являющиеся частью других выражений, называют ***полными (full-expression)***, а являющиеся — ***подвыражениями (subexpression)***. Синтаксис записи операций по отношению к операндам и их порядок вычисления в сложном выражении определяется правилами языка, которые будут рассмотрены далее. Полный список операций дан в приложении А.

Результат вычисления выражения — не обязательно временная величина, она может соответствовать хранящемся в памяти сущностям. Характеристикой выражения, определяющей трактовку временности его результата, является его ***категория значения (value category)***. Эта характеристика в языке C++ является частью системы типов и потому также определяет возможные действия над этим значением.

Наконец, в процессе вычисления выражения могут происходить побочные эффекты. ***Побочным эффектом (side effect)*** называют изменение состояния среды выполнения. Большинство побочных эффектов — это модификация объектов и ввод-вывод данных в процессе обмена информации между программой и средой выполнения. Обратите внимание, что в данном случае «побочный» не означает «нежелательный» (как это и есть в англоязычном термине) — напротив, в побочных эффектах заключена практически вся полезная работа программы! Это неожиданное утверждение вполне естественно для языка, не имеющего функциональную парадигму программирования в качестве основной.

Выражения лишь описывают последовательность вычислений, но не совершают их самостоятельно — это задача других языковых конструкций. ***Вычислением (evaluation)*** выражения называется процесс выполнения указанных в нём операций — вычисления значения выражения и инициация побочных эффектов. Большинство выражений в программе указываются именно в контекстах, где происходит их вычисление, но некоторые выражения или их части могут быть ***невычисляемыми (unevaluated)*** всегда или в некоторых случаях.

Так же как память среды выполнения разбивается на объекты, алгоритм, являющийся последовательностью действий, согласно процедурной парадигме программирования, разбивается на части — ***функции (function)***. Функции в языке C++ соответствуют процедурам или подпрограммам: последовательности действий, содержащиеся в функциях, могут включать в себя действие ***вызов функции (function call)*** в виде одноимённой операции — приостановку выполнения текущей функции для выполнения команд из другой. Функции могут иметь ***параметры (parameter)*** — объекты, принимающие значения, указываемые как ***аргументов (argument)*** при каждом вызове функции. Функция также ***возвращает (return)*** некоторое значение в качестве результата своего выполнения.

Операторы (statement) — конструкции языка, определяющие выполняемые программой действия. Большинство операторов языка C++ отвечают за изменение естественного последовательного порядка выполнения программы и вычисление выражений.

Описания (declaration) задают интерпретацию и атрибуты имён. Смысл большинства имён, используемых в программе, задаётся в описаниях. ***Определение (definition)*** — частный случай описания, который помимо перечисления свойств имени задаёт содержимое сущности, ему соответствующей. В частности, определения объектов отвечают за выделение им памяти в среде выполнения, а определения функций содержат запись последовательностей действий.

Объединим всё описанное выше в краткое описание сущности программы на языке C++. ***Программа состоит из описаний, задающих атрибуты и содержание сущностей, используемых в программе, с указанием их имён. Функции являются частями алгоритма, реализуемого программой, и содержат операторы, которыми записываются его шаги и порядок их выполнения. Вычисления, совершаемые программой, задаются операциями, используемыми в выражениях. Последовательность побочных эффектов, вызываемых ими, приводит к изменению содержимого объектов и другого состояния среды выполнения для достижения необходимого результата работы программы.***

3.6. Пример первой программы

Чтобы только что сформулированное основополагающее утверждение об общей структуре программы не осталось без материального подкрепления, приведём пример простейшей программы на языке C++:

```
1  #include <iostream>
2
3  int main()
```

```
4  {
5      std::cout << "Input two real numbers: ";
6      double a,b,c;
7      if(std::cin >> a >> b){
8          c = a*b;
9          std::cout << a << " * " << b << " = " << c << '\n';
10     }
11 }
```

Эта программа запрашивает у пользователя два числа и выводит их произведение. Разберём её по строкам, указав к каким из рассмотренных элементов языка они относятся.

1. Строки, начинающиеся с символа `#` содержат директивы, обрабатываемые на этапе предварительной обработки. В последовательность токенов, составляющих единицу трансляции, эти директивы не попадают, поскольку уже обработаны и не относятся к синтаксису самого языка. Данная директива включает в программу необходимые описания из стандартной библиотеки, необходимые для взаимодействия с пользователем.
2. Это пустая строка. В большинстве контекстов пробельные символы, такие как пробелы и переносы строк, являются разделителями токенов, но их количество и состав не имеет значения, важен сам факт разделения, поэтому программист волен применять любые их последовательности для улучшения зрительного восприятия текста программы.
3. Это описание функции с именем `main`, которая не имеет параметров, и возвращает некоторое целое числовое значение. Функция, названная `main` имеет особый смысл: это функция, с которой начинается и которой заканчивается выполнение программы. Её результат возвращается операционной системе. Все остальные строки программы задают тело этой функции, т.е. соответствующую ей последовательность действий, поэтому это описание также является определением. Других функций в этой программе не определено, поэтому весь алгоритм её работы содержится в одной функции без разделения на части, что допустимо для столь простой программы.
4. Пунктуатор `{` на данной строке обозначает начало блока, содержащего последовательность операторов, соответствующего алгоритму в функции `main`.
5. На данной строке записан оператор, предписывающий вычисление выражения, побочным эффектом которого является вывод на стандартное устройство вывода приглашения пользователю ввести два числа.
6. Эта строка содержит описание трёх идентификаторов `a`, `b`, `c`, о которых сообщается, что это идентификаторы объектов типа, способного хранить действительные числа с ограниченной точностью и допускающего обычные арифметические операции. В соответствии с семантикой языка данное описание является определением, что приводит к выделению памяти среды выполнения под соответствующие объекты.
7. Выражение в скобках на данной строке в процессе своего вычисления в качестве побочных эффектов осуществляет ввод двух значений со стандартного устройства ввода и запись их в объекты `a` и `b`. Результат вычисления этого выражения — логическое значение, характеризующее успешность этой операции. Оператор `if`, начало которого записано на данной строке вычисляет это выражение, вызывая его побочные эффекты, и выполняет следующую группу команд в фигурных скобках, только если выражение оказалось истинным, то есть ввод был успешен.

   Можно перечислить все токены, на которые будет разбита эта строка по порядку:

   - Ключевое слово `if`;
   - Пунктуатор `(`;
   - Идентификатор `std`;
   - Операция `::`;
   - Идентификатор `cin`;
   - Операция `>>`;
   - Идентификатор `a`;
   - Операция `>>`;
   - Идентификатор `b`;
   - Пунктуатор `)`;

- Пунктуатор `{`;

Токенов, являющихся литералами, в этой строке нет.

8. Эта строка выполняет основную смысловую работу программы: она вычисляет выражение, побочным эффектом которого является запись произведения значений, хранящихся в объектах `a` и `b` в объект `c`.
9. Выражение в данной строке в качестве побочного эффекта выводит на стандартное устройство вывода исходные данные и результат вычислений.
10. Закрывающая фигурная скобка на данной строке обозначает конец блока условно выполняющихся команд, начатый в строке 7.
11. Последняя строка программы содержит пунктуатор `}`, являющийся парным к пунктуатору на строке 4. Он показывает, где заканчивается определение функции `main`.

Перейдём к детальному рассмотрению элементов языка, чтобы получить полное объяснение структуры и смысла данной программы. Любопытно, что дать исчерпывающие объяснение даже такой простой программе удастся только после рассмотрения практически всего языка.

Глава 4

От теории к первой программе

В этом разделе мы дадим минимальные сведения о структуре приведённой ранее первой программы-примера. Значительная часть этих знаний будет поверхностной, что, к сожалению, в случае использования языка C++ неизбежно. После того, как мы получим понимание данной программы в первом приближении, мы займёмся уточнением требуемых понятий в будущих разделах, а имеющиеся знания позволят нам приступить к практике программирования.

4.1. Основные арифметические типы данных

Начнём рассмотрение системы типов с ***фундаментальных (fundamental)*** типов — фиксированного множества типов, встроенного в язык. Типы объектов, которые не содержат в себе других объектов, то есть являющиеся неделимыми, называются ***скалярными (scalar)***. Это большинство типов, обладающих самодостаточными значениями, с которыми мы будем иметь дело в начале.

Арифметические (arithmetic) типы данных предназначены для хранения значений, являющихся по смыслу числами. Соответственно, имеется набор операций, позволяющих совершать над значениями этих типов обычные арифметические действия. Язык C содержит несколько различных арифметических типов, отличающихся занимаемым объёмом памяти и представлением. Это позволяет программисту выбирать необходимый тип данных исходя из требований задачи по диапазону и точности хранения значений, а также взаимодействовать с различными программными и аппаратными интерфейсами, использующими различные представления данных. Неформально о типе данных, требующем n бит памяти для представления своих значений говорят как о n***-битном*** типе. Арифметические типы делят на ***целые (integer)*** типы и типы ***с плавающей точкой (floating point)***.

4.1.1. Стандартные целые типы данных

Множества значений целых типов данных содержат только целые числа. Целые типы разделяют по ***знаковости (signedness)*** на ***знаковые (signed)*** и ***беззнаковые (unsigned)***. В допустимые значения беззнаковых типов входят только неотрицательные целые числа, знаковые типы такого ограничения не имеют, их диапазон значений обычно симметричен или почти симметричен относительно нуля. Среди целых типов также выделяют ***символьные (character)*** типы, предназначенные для хранения числовых кодов символов, и булевский тип для логических значений. Нас в первую очередь будут интересовать так называемые ***узкие символьные типы (narrow character types)***. Основные ***стандартные (standard)*** целые типы языка C приведены в таблице 4.1.

Можно заметить, что имена этих типов состоят из ключевых слов языка. В таблице приведены «канонические» названия типов, но их обычно сокращают до наиболее кратких форм. Допустимы следующие изменения записи имён типов, содержащих в канонической форме ключевое слово `int`:

- Слово `int` можно опустить, если только оно не единственное. Пример: `unsigned` — более короткое имя типа `unsigned int`.
- Для знаковых типов этот факт может быть подчёркнут указанием ключевого слова `signed` перед остальными: `signed int`, или просто `signed` (см. предыдущее правило) — другое имя типа `int`.

	знаковые	беззнаковые
	`bool`	
узкие символьные	`char`	
	`signed char`	`unsigned char`
	`short int`	`unsigned short int`
	`int`	`unsigned int`
	`long int`	`unsigned long int`
	`long long int`	`unsigned long long int`

Таблица 4.1: Основные стандартные целые типы данных

Конкретная реализация языка может содержать и другие целые типы, называемые ***расширенными (extended)***.

Тип `bool` позволяет представлять одно из двух значений «ложь» или «истина». В тексте программы эти значения могут быть записаны логическими литералами `false` и `true` соответственно. Значения этого типа часто связывают в семантике языка и представлении в памяти с числами 0 и 1 соответственно, поэтому этот тип и относят к целым.

Представление целых чисел без знака в памяти среды выполнения соответствует позиционной системе счисления с основанием 2: каждому биту присваивается вес, соответствующий степеням числа 2, начиная с нулевой. Значением объекта в таком случае является сумма весов всех битов его содержимого, имеющих значение 1:

$$x = \sum_{i=0}^{N-1} b_i \times 2^i, \tag{4.1}$$

где b_i — значение i-го бита, а `N` — число ***значащих (value)*** битов в представлении значения числа. Это число называют ***точностью (precision)*** целого типа. Чаще всего все биты представления объекта вносят свой вклад в данную формулу, т.е. являются значащими.

Знаковые целые типы, помимо M значащих, содержат ***бит знака (sign bit)***. Поскольку один бит из представления объекта используется в качестве бита знака, число значащих бит в знаковом типе на один меньше, чем в соответствующем беззнаковом. ***Шириной (width)*** целого типа называют число его значащих и знаковых битов. Для беззнаковых типов ширина совпадает с точностью, для знаковых — на единицу больше. Как следствие, множество неотрицательных значений знакового типа есть подмножество значений соответствующего знакового. Когда он равен 0, значение объекта интерпретируется как для соответствующего беззнакового типа.

Для случая, когда бит знака имеет значение 1, есть три основных варианта интерпретации значения:

- ***Прямой код (sign-magnitude)***: итоговое значение равно таковому для представления с нулевым битом знака со знаком минус.
- ***Дополнительный код (two's complement)***: бит знака имеет вес $-(2^M)$.
- ***Обратный код (one's complement)***: бит знака имеет вес $-(2^M - 1)$.

Рассмотрим в качестве примера представление целых чисел со знаком и без, использующее 3 бита:

Значения бит (бит знака)	Значение представления			
	Без знака	Код со знаком		
		Прямой	Обратный	Дополнительный
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

Таблица 4.2: Интерпретация 3-битных целых чисел

Перечислим основные свойства данных представлений:

- При использовании одинакового объёма памяти, знаковые коды обеспечивают представление вдвое меньшего диапазона чисел по абсолютному значению, зато в обе стороны от нуля.
- Прямой и обратный код имеют два представления для числа ноль, которые называют положительным и отрицательным нулём. Они ведут себя одинаково с математической точки зрения, но тот факт, что не все значения типа имеют однозначное представление может усложнять работу с ними на аппаратном или программном уровне.
- Для смены знака числа в прямом коде достаточно инвертировать бит знака. Для числа в обратном коде нужно инвертировать все биты. Операция смены знака в дополнительном коде сложнее и обычно требует двух шагов: инвертирования всех бит и прибавление единицы, рассматривая представление как беззнаковое (Пример: $-(2_{(10)}) = -(010_{(2)}) = bitwiseNot(010_{(2)}) + 1 = 101_{(2)} + 1 = 110_{(2)} = -2_{(10)}$, где $bitwiseNot$ — инверсия битов).
- В дополнительном коде диапазон представимых значений не симметричен относительно нуля, и при смене знака у максимальной по абсолютному значению отрицательной величины получается значение, не представимое в данном типа.
- Операции сложения и вычитания в дополнительном коде могут быть выполнены корректно, даже если рассматривать хранимые данные как беззнаковые. При этом вычитание может быть реализовано как сложение уменьшаемого с вычисленным в дополнительным коде значением, противоположным вычитаемому. Это позволяет использовать одну и ту же аппаратную или программную реализацию для операций сложения и вычитания как для знаковых, так и для беззнаковых типов.

Таким образом, для n-битного представления целые беззнаковые типы допускают целые значения в интервале $[0; 2^n)$, а знаковые — $(-(2^{n-1}); 2^{n-1})$ (для некоторых представлений интервал замкнут слева).

Набор инструкций рассматриваемой нами архитектуры x86 подразумевает работу со знаковыми величинами в дополнительном коде. Все реализации языка C++ для неё используют это представление для фундаментальных типов. Это один из примеров того, как разрешение стандартом нескольких вариантов реализации одного из аспектов языка позволяет в большинстве случаев воспользоваться прямой поддержкой тех или иных возможностей аппаратных средств конкретной платформы для максимальной производительности, оставаясь в рамках стандарта.

Множество значений типа `char` включает все элементы базового набора символов времени выполнения в виде положительных значений. Он совпадает по поведению, представлению и множеству значений либо с `signed char`, либо с `unsigned char` по выбору реализации, но считается отдельным от них обоих типом. Отметим, что `char` не является сокращением от `(signed char)` и считается отдельным типом.

Размер представления всех символьных типов по определению — 1 байт.

Тип `int` отличается тем, что является минимально широким типом, для которого рассматриваемая архитектура процессора имеет инструкции по эффективной обработке. Это свойство определяет широкое применение этого типа, когда нет каких-либо других требований, а также отражено в других конструкциях языка.

Точность стандартных целых типов определяется реализацией исходя из минимумов,

заданных в стандарте и свойств аппаратной и программной сред. Совокупность характеристик базовых типов данных принято называть ***моделью данных (data model)***, которые включают в своё определение и размеры фундаментальных типов. Требования стандарта и наиболее широко используемые модели данных с этой точки зрения приведены в таблице 4.3.

Разрядность	Модель	`short`	`int`	`long`	`long long`
Минимум по стандарту		16	16	32	64
32-битные	ILP32	16	32	32	64
64-битные	LLP64	16	32	32	64
	LP64	16	32	64	64

Таблица 4.3: Популярные модели данных

На архитектуре x86 в защищённом 32-битном режиме подавляющее большинство реализаций используют модель данных ILP32. Модель LLP64 применяется в 64-битных версиях ОС Windows, а LP64 — в большинстве других 64-битных операционных системах персональных компьютеров.

К категории токенов-литералов относят ***целочисленные литералы***, которые позволяют использовать в тексте программы конкретные значения целочисленных типов. Синтаксически целочисленные константы состоят из трёх частей, записываемых слитно:

1. Опциональный префикс системы счисления. Литералы, начинающиеся с символа `0`, задаются в восьмеричной системе счисления, с символов `0b` или `0B` — в двоичной. Литералы, начинающиеся с символов `0x` или `0X`, задаются в шестнадцатеричной системе счисления, при этом в роли цифр со значениями 10–15 выступают буквы A–F (в любом регистре). При отсутствии одного из этих префиксов литерал задаётся в десятичной системе.
2. Само значение литерала, записываемое последовательностью цифр в соответствующей системе счисления. Цифры записываются слитно, но могут быть разделены произвольным количеством знаков `'`, которые транслятором игнорируются, для облегчения прочтения, например, `123'456'789`.
3. Опциональные суффиксы `U`, `L` или `LL` (в любом регистре и последовательности, из `L` и LL может присутствовать максимум один). Эти суффиксы влияют на тип значения литерала.

Для сравнения диапазонов значений целых типов с каждым целым типом связывают некоторую количественную величину — ***ранг целочисленного преобразования (integer conversion rank)***, которые назначаются следующим образом:

- Среди знаковых каждый тип имеет ранг больше, чем все остальные с меньшей точностью. Все знаковые типы должны обладать разным рангом, даже если они имеют одинаковое представление. Стандартные знаковые типы обладают рангами по возрастанию в порядке `signed char`, `short`, `int`, `long`, `long long`, т.е. в порядке перечисления в таблице 4.1.
- Ранги беззнаковых типов равны рангам соответствующих знаковых типов.
- Ранги всех узких символьных типов равны.
- Тип `bool` обладает минимальным рангом.

Таким образом в отношении рангов стандартных типов можно записать:

$$\begin{gathered} rank(\texttt{bool}) < \\ < rank(\texttt{char}) = rank(\texttt{signed char}) = rank(\texttt{unsigned char}) < \\ < rank(\texttt{short}) = rank(\texttt{unsigned short}) < \\ < rank(\texttt{int}) = rank(\texttt{unsigned}) < \\ < rank(\texttt{long}) = rank(\texttt{unsigned long}) < \\ < rank(\texttt{long long}) = rank(\texttt{unsigned long long}), \end{gathered} \tag{4.2}$$

где под выражением вида $rank$(имя типа) подразумевается ранг указанного типа.
Точный алгоритм определения типа целочисленной константы таков:

1. Установить интервал рангов типов-кандидатов: минимальный ранг — ранг `int` (нет суффиксов `L` и `LL`), ранг `long` (суффикс `L`) или ранг `long long` (суффикс `LL`); максимальный ранг — ранг `long long`.
2. Рассмотреть беззнаковые типы, если есть суффикс `U` или литерал не в десятичной системе счисления. Рассмотреть знаковые типы, если суффикса `U` нет.
3. Расположить рассматриваемые типы по возрастанию ранга, знаковые перед беззнаковыми с одинаковым рангом.
4. Выбрать первый по порядку тип, в интервал допустимых значений которого попадает значение литерала.
5. Если ни один из стандартных целых типов не подходит, могут рассматриваться ***расширенные (extended)*** целочисленные типы, определяемые реализацией.

Отметим, что в языке C++ знак числа не входит в синтаксис задания литералов, поэтому все числовые литералы имеют неотрицательные значения. Также целочисленными литералами не представимы значения рангов ниже `int`. Для задания таких значений можно применить к литералам операции смены знака и приведения типов, которые будут рассмотрены далее.

Примеры:

- `12` $= 12_{(10)}$;
- `012` $= 12_{(8)} = 10_{(10)}$;
- `0x1'2` $= 12_{(16)} = 18_{(10)}$;
- `0` $= 0_{(8)} = 0_{(10)}$;
- `08` — синтаксически неверная запись, константа начинается с 0, т.е. задаётся в восьмеричной системе счисления, в которой следующей цифры 8 нет.

- `1ULL` — тип `unsigned long long`, как единственный кандидат на основании применённых суффиксов.
- `0x100000000` — из-за шестнадцатеричного представления без суффикса рассматриваются как знаковые, так и беззнаковые типы, начиная с ранга `int`. Это значение, равное 2^{32}, будет иметь тип `long long` в моделях памяти ILP32 и LLP64 и тип `long` в модели LP64, как знаковые (рассматриваемые в первую очередь) типы с минимальным рангом и более чем 32 битами точности (ровно 32 бит не хватит даже в беззнаковом случае, т.к. интервал значений беззнакового типа открыт справа).

Некоторые биты в представлении целых типов могут не использоваться — их называют ***заполняющими битами (padding bits)***, их значение в представлении не уточняется. Биты заполнения в целочисленных типах на практике встречаются только в исключительных случаях, а для символьных типов гарантируется их отсутствие. В моделях данных архитектуры x86, например, их нет вообще.

При хранении целых чисел и других данных, занимающих более одного байта, возникает вопрос о ***порядке разрядов (endianess)***. Поскольку веса битов назначаются в пределах байта последовательно, и отдельно взятый байт может рассматриваться как отдельное число, представление N-байтного беззнакового числа может рассматриваться как запись в позиционной системе счисления с основанием $2^{\texttt{CHAR\_BIT}}$, где `CHAR_BIT` — число бит в байте:

$$x = \sum_{i=0}^{i<N} B_i \times (2^{CHAR\_BIT})^i, \tag{4.3}$$

где B_i — значение «цифры» соответствующей i-му байту. В этих обозначениях байты, соответствующие меньшим индексам i хранят младшие биты числа — биты с меньшим весом. При расположении в памяти этих байтов по порядку возможны два направления:

- ***От старшего к младшему (big-endian)***: байты хранятся начиная со старшего и далее по порядку до младшего по увеличению адресов памяти.
- ***От младшего к старшему (little-endian)***: байты хранятся начиная с младшего и далее по порядку до старшего по увеличению адресов памяти.

Например, число 258 в 4-байтном представлении big-endian хранится в виде последовательности байтов `00 00 01 02`. В этой записи каждый байт представлен двузначным числом в шестнадцатеричной системе счисления, такая нотация используется многими инструментальными средствами для представления значений в памяти.

Отметим, что big-endian также является системой, используемой при обычной записи чисел арабскими цифрами при письме слева направо: запись начинается слева и первыми пишутся старшие цифры числа. То же самое число в представлении little-endian задаётся последовательностью байтов `02 01 00 00`. Поскольку байт с точки зрения языка C++ является минимальной адресуемой единицей, варианты размещения мелких единиц в более крупных удаётся заметить только на этом уровне. Например, на аппаратном уровне биты внутри байта также хранятся в определённой последовательности, но обнаружить соответствующие эффекты на уровне языка C++ обычно не удаётся.

Архитектура x86 использует, в отличие от большинства других архитектур, представление little-endian.

В редких случаях можно столкнуться со смешанными формами (**mixed-endian**), например, когда последовательность байтов в двухбайтных словах одна, а последовательность этих слов в 4-байтных единицах — другая.

Понятие порядка разрядов может применяться и к отдельным битам, когда рассматриваются их последовательности, но с точки зрения представления целых чисел в памяти оно не различимо с точки зрения программы и потому не существенно.

Символьные типы относятся к целочисленным типам, поскольку хранят коды символов в виде целых чисел по соответствию, заданному используемой кодировкой. В начале для простоты мы будем использовать только базовый набор символов и исходить из того, что одному символу соответствует в точности один байт, т.е. одно значение узкого символьного типа. Конкретный вид таблицы символов стандартом не оговаривается за исключением требования, чтобы арабским цифрам от 0 до 9 были назначены последовательные значения.

Для задания кодов символов в программе используются символьные литералы, состоящие из ровно одного требуемого символа, заключённого в одинарные кавычки, например, `'A'` — символьный литерал со значением кода буквы A. Символьные литералы имеют тип `char`.

Нам уже встречались последовательности символов, т.е. строки, которые могут быть заданы ***строковыми литералами***, являющимися последовательностью символов (возможно пустой), заключённой в двойные кавычки, например, `"string"`. Тип и представление строковых литералов будут рассмотрены позднее.

Для представления специальных значений внутри символьных и строковых литералов используются ***escape-последовательности***, начинающиеся со знака `\`. В процессе предварительной обработки каждая такая последовательность заменяется одним соответствующим символом без использования его специального значения (которого «избежали» — отсюда и название). Существующие escape-последовательности приведены в таблице 4.4:

Escape	Значение
`\'`	'
`\"`	"
`\?`	?
`\\`	\
`\a`	(Alert) «Звонок». Воспроизводит звуковой или визуальный сигнал на терминале.
`\b`	(Backspace) Сдвиг курсора на символ назад.
`\f`	(Form feed) Переход на новую страницу.
`\n`	(New line) Переход на первую позицию следующей строки.
`\r`	(carriage Return) Переход на первую позицию текущей строки.
`\t`	(horizontal Tab) Переход на следующую позицию горизонтальной табуляции.
`\v`	(Vertical tab) Переход на следующую позицию вертикальной табуляции.

Таблица 4.4: Фиксированные escape-последовательности

Символы `'` и `\` всегда необходимо представлять escape-последовательностями в символьном литерале, поскольку первый является знаком её конца, а второй сам начинает escape-последовательность. То же требуется для двойной кавычки и обратного слеша в строковом литерале.

Также с помощью escape-последовательностей можно напрямую указать символ с необходимым значением, что можно сделать двумя способами:

- Числом из от 1 до 3 восьмеричных цифр после символа `\`, например `'\0'` — ***нулевой символ (null character)*** (символ со значением 0, не путать с символом `'0'`!).
- Шестнадцатеричным числом после префикса `\x`, например, `\x41` — символ со значением 65.

C++ опционально допускает запись нескольких символов внутри символьных литералов, их значение при этом имеет тип `int` и определяется реализацией. Большинство компиляторов выдают в таком случае предупреждение, поскольку это почти наверняка не то, чего хотел программист — это либо опечатка, либо строковый литерал, записанный по ошибке в одинарных кавычках вместо двойных.

Не путайте символьные литералы со строковыми, выражения `'a'` и `"a"` имеют принципиально разные типы и значения! Тип строковых литералов будет рассмотрен позднее, мы вводим их сейчас исключительно для удобства представления последовательностей символов в нашей первой программе.

Также в фундаментальные типы входят ***широкие символьные типы (wide character types)*** `wchar_t`, `char16_t`, `char32_t`, а также типы `void` и `std::nullptr_t`, которые будут рассмотрены позднее.

4.1.2. Стандартные типы с плавающей точкой

К стандартным ***типам с плавающей точкой (floating point types)*** в языке C++ относятся `float`, `double` и `long double`. Множество значений каждого следующего из перечисленных типов включает все значения предыдущего. Типы с плавающей точкой позволяют хранить значения действительных чисел с фиксированной точностью, определяемой числом значащих цифр в некоторой позиционной системе счисления независимо от порядка.

Поясним смысл понятия «плавающая точка»:

$$12.3 = 12.3 \times 10^0 = 1.23 \times 10^1 = 0.123 \times 10^2 = 123 \times 10^{-1} \tag{4.4}$$

Таким образом, точка в записи числа в позиционной системе счисления может быть расположена в произвольном месте при условии компенсации домножением на соответствующую степень основания. В общем случае запись числа x в виде

$$x = mantissa \times base^{exponent} \tag{4.5}$$

называют ***экспоненциальной записью (scientific notation)***, где ***mantissa*** — мантисса (значащая часть числа), *base* — основание используемой системы счисления, а ***exponent*** — экспонента (порядок числа). Очевидно, что одно и то же число может быть представлено бесконечно большим числом вариантов экспоненциальных форм. Одну из них, в которой слева от точки в мантиссе стоит ровно одна ненулевая цифра, называют ***нормализованной (normalized)***. Договорившись об использовании нормализованной формы и конкретной системы счисления, для хранения числа в такой форме достаточно сохранить знаки мантиссы и экспоненту. Выделив под хранение мантиссы фиксированный объём памяти, мы получим способ записи чисел с конечной точностью в широком диапазоне порядков, поскольку хранение экспоненты не занимает много места. Хранение чисел с неполной точностью, но зато в гораздо большем интервале порядков является отличительной особенностью типов с плавающей точкой, по сравнению с целыми типами (являющимися представителями типов с ***фиксированной точкой (fixed-point)***), которые хранят свои значения всегда точно, но в гораздо меньшем интервале значений.

В языке C++ экспоненциальная запись числа имеет вид слитной записи мантиссы и экспоненты, разделённой буквой `e` или `E`. В этой записи используется только десятичная система счисления. Экспонента и символ-разделитель `E` могут отсутствовать, экспонента в таком случае принимается равной 0, но в таком случае в мантиссе обязательно должна присутствовать точка, поскольку иначе эта запись синтаксически будет являться записью целочисленной константы. В мантиссе, как и в целочисленном литерале, могут для наглядности использоваться разделители `'`. Примеры:

- `12.3` = 12.3;
- `1.23e1` = $1.23 \times 10^1 = 12.3$;

- `1'2'3E-1` $= 123 \times 10^{-1} = 12.3$;
- `0.` $= 0 \times 10^{0} = 0$ (без точки константа была бы целочисленной);
- `.1` $= 0.1 \times 10^{0} = 0.1$ (ноль перед точкой можно опускать).

Тип всех приведённых выше констант — `double`. Этот тип и следует использовать, если нет жёстких требований, например по точности или занимаемому объёму памяти, поскольку он нередко является представлением, наиболее удобным для аппаратной составляющий реализации в обработке.

Следует отметить, что большинство вариантов представления значений с плавающей точкой в памяти среды выполнения используют запись мантиссы в двоичной системе счисления. Поскольку в разложении основания 10, в котором задаются литералы с плавающей точкой, на простые множители содержится множитель 5, взаимно простой с 2, бо́льшая часть мантисс, являющихся конечными в десятичной системе счисления, становятся бесконечными периодическими дробями в двоичной. Поскольку разрядность мантиссы конечна, происходит потеря точности. Эти потери следует учитывать в вычислениях, которые могут давать неточный результат в большинстве случаев, когда используются типы с плавающей точкой.

Из-за этого свойства, усложняющего получение точных ответов и влияющего на корректность реализации алгоритмов, зависящих от сравнений значений с плавающей точкой, использовать их следует только в случае явной необходимости. Другой проблемой этих типов является отсутствие прямой их поддержки маломощными архитектурами микропроцессоров, где операции с ними могут выполнятся очень медленно или не поддерживаться аппаратно вообще.

При задании константы с плавающей точкой, за ней может следовать один из суффиксов (в любом регистре), меняющих её тип: `F` (`float`) или `L` (`long double`). Например, `1.f` — константа со значением 1 типа `float`.

Возможные варианты представления значений с плавающей точкой стандартом не ограничиваются. Одним из часто используемых представлений чисел с плавающей точкой является стандарт ISO/IEC/IEEE 60559:2011 [2], более широко известный под своим старым наименованием IEEE 754, он используется и на архитектуре x86. В нём определены несколько форматов хранения чисел с плавающей точкой разной точности и правила выполнения операций над ними. Рассмотрим представление формата binary64 иначе называемого форматом ***с двойной точностью***. Тип `double` получил своё имя именно потому, что часто ему и соответствует. Это 64-битная величина, биты который разбиты на следующие группы:

- Бит знака.
- «Смещённая» экспонента — 11 бит.
- Мантисса — 52 бита.

С точки зрения способа хранения отрицательных чисел, этот формат относится к прямому коду. Интерпретация мантиссы зависит от значения экспоненты как беззнакового целого:

- Экспонента $000_{(16)}$: если мантисса — 0, то значение равно нулю (со знаков), иначе это денормализованное значение, которое не использует всю возможную точность данного типа (встречается редко, обычно все хранимые значения нормализованы, чтобы использовать все значащие биты мантиссы).
- Экспонента $7FF_{(16)}$: если мантисса — 0, то значение равно «бесконечности» (со знаком), иначе это одно из представлений специального значения NaN — Not a Number, являющегося результатом выполнения некорректных операций, результат которых не может быть представлен другим образом (деление ноля на ноль, логарифм от отрицательных чисел, ...).
- Другая экспонента: обычное число в нормализованной форме. Поскольку в двоичной системе записи мантиссы, соответствующей ненулевому значению, единственное возможное значение знака перед точкой — 1, то этот знак не хранится. Считается, что эффективное число двоичных знаков мантиссы — 53 = 52 хранимых + одна неявная единица. К хранимому беззнаковому значению экспоненты прибавляется смещение, чтобы оно могло означать как положительные, так и отрицательные порядки. Итоговое

значение равно

$$x = (-1)^{sign} \times 2^{exponent-1023} \times 1.mantissa_{(2)}, \tag{4.6}$$

где *sign* — бит знака, *exponent* - экспонента, *mantissa* — хранимые биты мантиссы.

Приведём пример интерпретации значения объекта типа `double` в представлении binary64 в среде выполнения, использующей порядок байтов little-endian. Содержимое объекта — байты со значениями `00 00 00 00 00 00 e4 3f`. В соответствии с порядком байтов, биты значения разбиваются на группы следующим образом:

- Бит знака — 0.
- Смещённая экспонента — $3FE_{(16)} = 1022_{(10)}$.
- Хранимая часть мантиссы — $010...0_2$.

Экспонента соответствует нормализованному числу, которое равно

$$x = (-1)^0 \times 2^{1022-1023} \times 1.01_{(2)} = 0.625_{(10)}. \tag{4.7}$$

4.1.3. Стандартные преобразования арифметических типов

Одной из основных характеристик типа является множество его значений. У разных типов эти множества могут пересекаться, например, число 0 представимо в точности всеми известными нам типами. В программах нередко возникает необходимость изменения типа значения с сохранением его смысла, насколько это возможно, например, при использовании значений разных типов в одном выражении.

Язык C++ различает ***неявные (implicit)*** и ***явные (explicit)*** преобразования типов значений. Разница между ними в том, что явные преобразования выполняются только тогда, когда вручную затребованы программистом с помощью операций ***приведения типов (type cast)***, а неявные могут быть также выполнены и без такого указания, исходя из требований к типу значения в том или ином контексте. В данном разделе мы начнём рассмотрение ***стандартных преобразований (standard conversions)*** — неявных преобразований, встроенных в язык. (C++ также позволяет программисту определять дополнительные преобразования между определяемыми им типами, которые носят название ***пользовательских (user-defined)***.)

Стандартные преобразования разделены на несколько групп, мы начнём рассмотрение с тех из них, которые касаются известных нам типов: арифметических.

- ***Целочисленные повышения (integral promotions)***: значение целого типа ранга ниже `int`, кроме bool, преобразуется к типу `int`, если все значения исходного типа представимы в нём, иначе к `unsigned`, смысл значения сохраняется. Значение типа `bool` может быть преобразовано к типу `int`: `false` переходит в `0`, `true` — в `1`. Целочисленные повышения соответствуют расширению значений ширины ниже, чем `int`, до таковой, с которой процессор может напрямую выполнять вычисления.
- ***Целочисленные преобразования (integral conversions)*** определяют остальные преобразования между двумя целыми типами, не входящие в целочисленные повышения, за исключением преобразований к `bool` (о них см. ниже). Если исходное значение представимо в новом типе, оно сохраняется без изменений.

  Иначе если новый тип — беззнаковый, результат есть представимое в новом типе значение, сравнимое с исходным по модулю 2^n. Например, значения $-65528 = -2^{16} + 8$ и $262152 = 2^{16} * 4 + 8$ при преобразовании к 16-битному целому беззнаковому типу дадут в результате 8. Это неестественное, на первый взгляд, свойство соответствует отбрасыванию переносов и заёмов при реализации двоичной арифметики аппаратно, и таким образом соответствует действительному поведению большинства процессоров и не затрачивает дополнительных ресурсов на приведение по модулю, имеющееся в формальном математическом описании. Хотя это свойство может использоваться в интересных оптимизациях, тот факт, что в языке фактически нет настоящих беззнаковых типов, а есть только типы представления остатков от деления на степени числа 2, препятствует некоторым оптимизациям компилятора и является регулярным источником ошибок. Будьте бдительны при использовании беззнаковых типов!

  Если приведение осуществляется к знаковому типу, а значение в нём не представимо, результат определяется реализацией. К сожалению, существуют программы, ошибочно опирающиеся на эффекты этого поведения, характерные для реализаций с представлением

отрицательных чисел в дополнительном коде, которые также просто отбрасывают значения битов переноса.

Если новый тип — `bool`, то значение 0 преобразуется в `false`, а все остальные ненулевые значения — в `true`.

- ***Повышение с плавающей точкой (floating point promotion)***: значение типа `float` может быть преобразовано к `double`, смысл сохраняется в точности. Выделение этого преобразования в отдельную категорию связано с унаследованными от языка C возможностями (само преобразование, разумеется, необходимо).
- ***Преобразования с плавающей точкой (floating point conversions)***: преобразования между двумя типами с плавающей точкой, не включая соответствующее повышение. Если старое значение представимо в новом типе в точности, оно сохраняется без изменений. Если это не так, но оно лежит между двумя смежными значениями, входящими в множество значений нового типа, происходит округление до одного из них, определяемого реализацией. Иначе если старое значение не входит в множество значений нового типа, поведение не определено.
- ***Преобразования между целыми значениями и значениями с плавающей точкой (floating-integral conversions)***. Преобразование из типа с плавающей точкой к целочисленному типу, за исключением `bool`, отбрасывает дробную часть, если такое значение не представимо в новом типе, поведение не определено.

  Преобразование из целочисленного типа в тип с плавающей точкой сохраняет значение, если оно представимо в новом типе в точности, или округляет до ближайшего представимого значения. Если исходное значение вне пределов значений нового типа, поведение не определено. Значение типа `bool` преобразуется в 0 или 1, как и для приведения к целочисленному типу.

  > Пример: приведение значения $2^{56} - 1$ типа `long long` к типу `double`, использующему представление binary64. Исходное значение состоит из 56 единиц в двоичной записи, но мантисса в представлении binary64 хранится с точностью до 53 бит. Тем не менее, сама величина лежит в интервале допустимых значений этого типа. На реализации, в которой произойдёт округление вверх, результатом будет значение 2^{56}, как ближайшее представимое 53-битной мантиссой.

- ***Булевские преобразования (boolean conversions)***: значение 0 любого арифметического типа может преобразовано в `false`, а ненулевое — в `true` типа `bool`.

Данная классификация охватывает все возможные комбинации исходного и нового арифметических типов.

На архитектурах, использующих представление чисел с плавающей точкой, включающее специальные значения $\pm\infty$, неопределённое поведение при преобразовании к таким типам не будет наблюдаться, т.к. фактически при выходе за пределы конечных значений, представимых в этом типе, результат всё равно остаётся в полном диапазоне представимых значений $(-\infty; +\infty)$. Это следует учитывать при разработке переносимых программ.

4.1.4. Операция приведения типов static_cast

Явное преобразование может быть затребовано с помощью операции ***приведения типов (type-cast)*** `static_cast`, имеющей следующий синтаксис:

`static_cast<` type-id`>(` expression`)`

Здесь type-id — имя типа, к которому следует преобразовать значение выражения expression. Эта операция может выполнить явно все стандартные преобразования, например: `static_cast<short>(2)` — значение 2 типа `short` (исходный — `int`, целочисленное преобразование) и `static_cast<bool>(3.7)` — значение `true` типа `bool` (исходный — `double`). Преобразование к тому типу, которое выражение уже имеет, тоже возможно: `1` и `static_cast<int>(1)` — одно и то же значение 1 типа `int`.

4.2. Основные выражения с арифметическими операндами

Начнём рассмотрение выражений, содержащих арифметические значения и операции над ними.

4.2.1. Арифметические операции

Все арифметические операции объединяет то, что если их результат не представим в типе результата или не определён математически, поведение не определено. К сожалению, поскольку большинство реализаций игнорируют тем или иным образом переполнения, большое количество программ пытаются это использовать, что некорректно.

Простейшие унарные операции, применимые к арифметическим типам — это ***операции смены - и сохранения + знака***, также называемые просто унарным плюсом и минусом. Они имеют префиксную форму записи, то есть записываются перед своим операндом. Операция смены знака возвращает значение, противоположное операнду, а сохранения знака — равное. Операция сохранения знака применяется редко и присутствует в языке, скорее, для симметрии с математикой. Обе эти операции выполняют целочисленные повышения своего операнда, если это преобразование к нему применимо, перед вычислением. Это действие как раз соответствует неявному преобразованию к удобному для вычислений типу, о котором было сказано в определении целочисленных повышений. Приведём примеры:

- `-20` есть применение операции `-` к литералу `20` типа `int`. Значением этого выражения является -20 типа `int`.
- `-20u` есть применение операции `-` к литералу `20u` типа `unsigned int`. Значение -20 не входит в интервал значений типа `unsigned int`, но поскольку этот тип был беззнаковым, результатом в модели данных, где этот тип имеет точность в 32 бита, например, становится значение $4294967276 = -20 + 1 \times 2^{32}$. Несмотря на применение операции «смены знака» к положительному значению, результат остаётся положительной величиной беззнакового типа! В то же время применение операции смены знака к левой границе интервала значений знакового типа с представлением в дополнительном коде — неопределённое поведение, поскольку результат в том же типе не представим.
- `+1L` есть применение операции `+` к литералу `1L` типа `long`, результат имеет тот же тип и значение, что и сама константа, поскольку целочисленные повышения не затрагивают значения типов с рангами от `int` и выше.
- `+static_cast<short>(7)` — в этом выражении две операции. Сначала выполняется операция приведения типов, которая осуществляет преобразование значения литерала 7 типа `int` в тип `short`. Вторая операция `+` тоже сохраняет значение, но за счёт целочисленных повышений результат её вычисления, являющийся значением всего выражения, есть 7 опять типа `int`.

Когда речь идёт об арифметической операции с двумя операндами, встаёт вопрос об определении ***общего типа (common type)***, к которому они предварительно преобразуются, и который имеет результат, исходя из потенциально разных типов операндов. Этот вопрос решает процедура, называемая ***обычные арифметические преобразования (usual arithmetic conversions)***. Общий смысл этой процедуры состоит в том, что из двух типов выбирается тип с бóльшим интервалом значений или точностью, при этом типы с плавающей точкой имеют приоритет над целыми. В некоторых случаях тип результата будет отличен от типа обоих операндов.

Полный алгоритм обычных целочисленных повышений:

1. Если тип одного из аргументов `long double`, `double` или `float`, то это и есть общий тип. Проверка осуществляется в указанном порядке.
2. Если оба значения целочисленные (выявлено на прошлом шаге), то над операндами осуществляются целочисленные повышения, если они к ним применимы. Если теперь они одного типа, то это и есть общий тип.
3. Иначе если знаковость операндов одинаковая, то их общий тип — тип операнда с бóльшим рангом.
4. Иначе один аргумент имеет знаковый тип, а другой — беззнаковый. Если ранг типа

беззнакового операнда не меньше ранга знакового, то общий тип — тип беззнакового операнда.

5. Иначе, если знаковый тип может представлять все значения беззнакового, то общий тип — тип знакового операнда.
6. Иначе общий тип — беззнаковый целый тип того же ранга, что и тип аргумента, имеющий знаковый тип.

Одним из примеров неуточняемого поведения в языке C++ является порядок вычисления операндов большинства бинарных операций.

Большинство бинарных операций в языке C++ являются ***инфиксными (infix)*** по синтаксису — их обозначения пишутся между операндами, которые называют левым и правым.

В языке C++ пять основных арифметических операций, все они определяют тип своего результата согласно обычным арифметическим преобразованиям и являются инфиксными. Это операции ***сложения*** `+`, ***вычитания*** `-`, ***умножения*** `*`, ***деления*** `/` и ***взятия остатка от деления*** `%`, их смысл соответствует математическому. Отметим, что знаки `+` и `-` могут использоваться как для унарных операций, так и для бинарных, в зависимости от контекста с разным смыслом.

Поведение операций деления и взятия остатка при правом операнде равном 0 не определено. Операция взятия остатка от деления применима только к операндам целых типов и её поведение определено только в случае, когда результат операции деления с теми же операндами определён:

```
( a / b ) * b + a % b = a.
```
(4.8)

Исходя из это соотношения, например, остаток от деления -11 на 3 есть -2. При делении с целым типом результата дробная часть отбрасывается, т.е. результат округляется до целого в сторону нуля. Обратите внимание, что `1/2` есть 0 в силу того, что тип результата определён как целочисленный — `int`. Для получения результата 0.5 типа с плавающей точкой можно воспользоваться конструкциями `1./2` (один из операндов имеет тип `double`, определяющий тип результата) или `static_cast<double>(1)/2` (явное приведение одного из операндов к типу с плавающей точкой, полезно, когда операнд — нетривиальное подвыражение). Также отметим, что деление левой границы множества значений знакового типа на -1 — неожиданный для многих случай неопределённого поведения, хотя делитель и не нулевой. Это особенность использовалась в реальных атаках по отказу в обслуживании.

В качеств примера выражения с операцией, тип результата которой не совпадает ни с одним их типов операндов, приведём `'\x1'+static_cast<short>(2)`. Левый операнд имеет значение 1 типа `char`, заданное escape-последовательностью в символьном литерале, а правый — 2 типа `short`, к которому выполнено явное приведение типов. По алгоритму обычных целочисленных преобразований, т.к. среди операндов нету типов с плавающей точкой, типы обоих повышаются до `int`. Теперь они одинаковы, и таков тип значения выражения, равный трём.

4.2.2. Операции с логическими значениями

Для вычисления различных условий в языке имеются соответствующие операции.

Для сравнения арифметических значений используются операции ***сравнения (equality)*** на равенство `==` и неравенство `!=`. Обратите внимание, что операция, обозначаемая одним знаком `=`, которая тоже есть в C++ — это совсем другая операция! Также имеются операции ***отношения (relational)*** `<`, `<=`, `>` и `>=`. Все эти операции совершают обычные арифметические преобразования над операндами, после чего результатом становится значение типа `bool`, соответствующее истинности отношения. Например, выражение `1<2.5` имеет значение `true`, а `10==010` — `false` (правый операнд записан в восьмеричной системе).

Отметим следующую проблему, для которой многие компиляторы выдают предупреждение: следует избегать применения операций отношения к операндам разной знаковости, поскольку результат, вычисленный по правилам языка, может оказаться математически некорректным. Рассмотрим выражение `-3<3u` на 32-битной архитектуре, его значение — `false`, что не соответствует наивному ожиданию. Согласно обычным арифметическим преобразованиям, общий тип для операндов `int` и `unsigned` — `unsigned`, и значение -3 приводится к нему. По правилам приведения значений к беззнаковым типам, -3 приводится к диапазону $[0; 2^{32})$ и становится значением $2^{32} - 3$, что, теперь очевидно, не меньше, чем значение правого операнда — 3.

Для построения сложных условий предназначены ***логические (logical)*** операции, соответствующие основным булевым функциям. Операция логического отрицания `!` — префиксная операция, возвращающая ложь, если её операнд — истина, и наоборот. Она требует операнда типа `bool`, к которому будет произведена попытка неявно привести её операнд. Например: выражение `!false` имеет значение `true`, а `!42` — `false`: сначала операнд 42 типа `int` будет неявно преобразован к `true` по правилу булевского преобразования.

Две оставшиеся логические операции в языке C++ являются бинарными и имеют инфиксную форму записи. Операция логического И `&&` возвращает истину, когда оба её операнда истинны, а операция логического ИЛИ `||` — когда истинен хотя бы один операнд. В противном случае обе операции возвращают ложь. Они, как и операция логического отрицания, требуют операндов типа `bool`, выполняя неявные преобразования, если это не так. Например, `21&&3.7` — истина, а `false||!true` — ложь. Эти операции являются исключением из общей схемы приоритетов и ассоциативности операций, поскольку вычисляют свои значения по ***короткой схеме (short-circuit)***. Для них порядок вычисления операндов строго определён: сначала левый, потом, возможно, правый. При этом, если значение операции однозначно определяется значением первого операнда, то после его вычисления второй операнд ***не вычисляется***. Из свойств соответствующих булевых функций:

$$\begin{aligned} 0 \wedge x &= 0 \\ 1 \vee x &= 1, \end{aligned} \tag{4.9}$$

где x — произвольное логическое значение. Таким образом, вычисления правого операнда не происходит, когда левый равен 0 для операции И и 1 для операции ИЛИ. Это первый пример потенциально не вычисляемых выражений.

Отметим, что выражение `1<x<10`, где x — произвольное подвыражение, синтаксически корректно, но имеет отнюдь не тот смысл, который оно имеет в математике. Вначале выполняется сравнение `1<x`, после чего результат этой операции типа `bool` становится левым операндом в сравнении `<10`. Поскольку оба возможных значения левого операнда после обычных целочисленных преобразований (0 и 1) заведомо меньше десяти, результатом этого выражения является логическая истина для любых значений подвыражения `x`! Чтобы добиться желаемого эффекта необходимо записать условие в полной форме: `1<x&&x<10`.

При построении сложного выражения, содержащего множество операций сравнения и логики, следует обращать особое внимание на их порядок вычисления, который мы и рассмотрим далее.

4.2.3. Порядок вычисления операций в выражениях

В математике порядок вычисления операций в сложном выражении определяется их приоритетом и ассоциативностью. Более приоритетные операции выполняются перед менее приоритетными, а среди операций с одинаковым приоритетом, идущими подряд, ассоциативность определяет порядок выполнения — слева направо или наоборот.

В языке C++ таких понятий формально нет. Последовательность вычислений операций вместо этого определяется иерархией синтаксических правил языка, рассмотрение которых мы начнём в этом разделе. В общем случае, порядок вычисления операций в выражениях не может быть полностью определён в привычных математических терминах, но для удобства, понятия приоритета и ассоциативности всё же применяют, оговариваясь, когда происходят не описанные ими эффекты. Список всех операций языка и их классификация по приоритетам и ассоциативности приведена в приложении A.

В данном разделе приведена только часть полных определений синтаксиса языка, соответствующих интересующим нас в данным момент конструкциям. Рассмотрим первое определение:

```
primary-expression:
    literal
    (  expression  )
    …
```

Данное определение показывает основную форму записи синтаксических конструкций в стандарте языка: даётся определение элементу primary-expression (элементарное выражение), а затем перечислены варианты его задания по одному на строку: либо литерал, либо последовательность из пунктуатора `(`, выражения и пунктуатора `)`. Таким образом, любой литерал является элементарным выражением. Кроме этого, любое сколь угодно сложное выражение

может быть заключено в скобки, чтобы снова стать элементарным и неделимым выражением. Это соответствует использованию скобок в математической записи для группировки подвыражений с целью изменения порядка вычисления операций в нём относительно определяемого приоритетом и ассоциативностью операций. Автор рекомендует в начале изучения языка не бояться ставить лишние скобки, чтобы получить требуемый порядок вычисления операций, поскольку их естественная последовательность выполнения в некоторых случаях контринтуитивна.

Чем дальше мы движемся по цепочке этих правил, тем ниже получается неформальный приоритет соответствующих операций.

```
postfix-expression:
    primary-expression
    static_cast< type-id>( expression)
    …
```

Два пока известных нам варианта определения так называемого постфиксного выражения есть элементарное или операция `static_cast`. Все последующие правила будут содержать правило предыдущего уровня, задавая последовательность разбора.

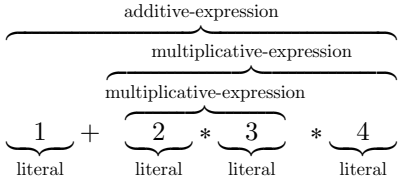
```
unary-expression:
    postfix-expression
    unary-operator    cast-expression
    …

unary-operator:    one of
    + - !
    …
```

Выражение с унарными операциями есть постфиксное выражение, или часть выражения следующего уровня, называемая выражением с приведением типа (не имеет отношения к рассмотренной операции `static_cast`), перед которым приписана одна из трёх известных нам унарных префиксных операций. Из-за префиксной формы записи это правило ссылается в том числе не только на конструкцию предыдущего уровня, но и на следующего.

```
cast-expression:
    unary-expression
    …

pm-expression:
    cast-expression
    …

multiplicative-expression:
    pm-expression
    multiplicative-expression  *   pm-expression
    multiplicative-expression  /   pm-expression
    multiplicative-expression  %   pm-expression

additive-expression:
    multiplicative-expression
    additive-expression  +   multiplicative-expression
    additive-expression  -   multiplicative-expression
```

Правила cast-expression и pm-expression не содержат известных нам элементов. Следующие два правила содержат синтаксис записи основных арифметических операций. Их порядок ссылок друг на друга обеспечивает наблюдаемый приоритет: у мультипликативных операций он выше, чем у аддитивных. В то же время тот факт, что ссылка происходит в правом операнде определяет видимую ассоциативность: слева направо.

Например, выражение `1+2*3*4` по этим правилам однозначно разбирается как:



```
shift-expression:
    additive-expression
    ...

relational-expression:
    shift-expression
    relational-expression  <   shift-expression
    relational-expression  >   shift-expression
    relational-expression  <=  shift-expression
    relational-expression  >=  shift-expression

equality-expression:
    relational-expression
    equality-expression  ==   relational-expression
    equality-expression  !=   relational-expression

and-expression:
    equality-expression
    ...

exclusive-or-expression:
    and-expression
    ...

inclusive-or-expression:
    exclusive-or-expression
    ...

logical-and-expression:
    inclusive-or-expression
    logical-and-expression  &&   inclusive-or-expression

logical-or-expression:
    logical-and-expression
    logical-or-expression  ||   logical-and-expression

conditional-expression:
    logical-or-expression
    ...
```

Здесь показаны остальные правила, соответствующие известным нам операциям и промежуточные, которые нам пока не известны. Из этого, например, видно, почему общие по смыслу операции сравнения и отношения относятся формально к разным группам, хотя близки по смыслу — они в разных правилах синтаксиса и по их построению имеют разный наблюдаемый приоритет.

Чтобы показать структуру выражения, может использоваться представление выражения в виде дерева. Узлами дерева являются значения, соответствующие литералам, результатам операций и другим способам их получения. Если значение зависит от других операндов, они являются его дочерними узлами. Соответствующую операцию можно записать между ними, а в каждом узле указать помимо самого значения его тип. Разобранное в этом виде выражение `(1.1f+static_cast<float>(2.2))*3` приведено на рис. 4.1. Скобки, выполняющие группировку операций в выражении, в дереве не присутствуют — порядок выполнения операций задаётся самой структурой дерева. Вершина дерева является результатом выражения, листья — значениями, явно заданными в выражении, а остальные узлы — промежуточными результатами. Дерево выражения без значений промежуточных и итогового результатов является одним из способов представления выражений в процессе их трансляции в реализациях компиляторов.

4.3. Объекты и доступ к ним

4.3.1. Простые определения объектов

Пока нами были рассмотрены только выражения, начинающие своё построение с литералов, т.е. фиксированных явно заданных значений. В то же время нам известно существование

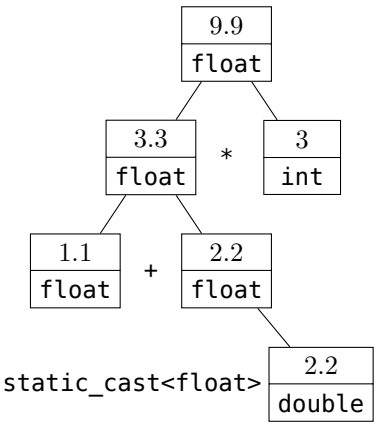


Рис. 4.1: Дерево выражения `(1.1f+static_cast<float>(2.2))*3`

объектов, с которыми можно выполнять операции чтения и записи значений. Как же создать объект и воспользоваться им? Ответ на оба этих вопроса лежит в использовании ***определений идентификаторов объектов***, являющихся частным случаем ***описаний***.

Рассмотрим пример:

```
int x,y;
```

Синтаксически описания начинаются с атрибутов, в первую очередь типа конструкций, которые описываются, после чего указываются эти конструкции, называемые ***описателями (declarator)***. Описатели перечисляются через запятую, если их несколько, завершается описание обычно точкой с запятой. Неформально его следует читать как «следующие конструкции соответствуют сущностям типа `int`: идентификатор `x` и идентификатор `y`». Поскольку любое определение также является описанием, оно задаёт интерпретацию и атрибуты идентификаторов, в данном случае `x` и `y`. Они же и являются описателями в данном случае. Поскольку тип `int` является типом данных, это описание задаёт интерпретацию соответствующих ему идентификаторов как ***имён объектов***. Основным (но не единственным) атрибутом этого имени является его тип — `int`. В примерах, которые мы будем рассматривать, пока не узнаем все атрибуты описаний, все описания объектов будут определениями, т.е. помимо задания вышеперечисленного смысла, они также будут приводить к выделению памяти среды выполнения под описываемые объекты. Итак, в нашем распоряжении два объекта типа `int`. Отдельные идентификаторы — лишь частный случай имён, которые в общем случае могут состоять из нескольких различных токенов.

4.3.2. Категории значений

Все выражения относят к одной из трёх фундаментальных ***категорий значений (value category)*** исходя из того, идентифицируют ли они некоторый объект или функцию:

- ***Леводопустимое (lvalue)*** — выражения, идентифицирующие объекты или функции.

> - ***Истекающие (xvalue (expiring value))*** — специальная категория значений, идентифицирующая объекты, которые требуется трактовать как временные.

- ***Чисто праводопустимое (prvalue (pure rvalue))*** — значение, не связанное с объектом.

Помимо общей категории «значение», есть две вспомогательных категории:

- ***Обобщённые леводопустимые (glvalue (generalized lvalue))*** — объединение леводопустимых и истекающих выражений.
- ***Праводопустимые (rvalue)*** — объединение истекающих и чистых праводопустимых выражений.

Истекающие выражения входят в обе из этих дополнительных категорий, но пока в наших программах встречаться не будут, так что пока мы будем работать только с двумя категориями значений.

4.3.3. Чтение и запись объектов. Операция простого присваивания. Оператор-выражение.

Все пока рассмотренные нами литералы и результаты выражений являлись чисто праводопустимыми выражениями, т.к. не были связаны с объектами. Теперь в нашем распоряжении новое элементарное выражение — идентификатор объекта:

```
primary-expression:
    …
    id-expression

id-expression:
    unqualified-id
    …

unqualified-id
    identifier
    …
```

В этом фрагменте правил синтаксиса показано, что идентификаторы тоже могут выступать в качестве элементов выражений. Поскольку в нашем случае они являются именами объектов, то их категория значения — леводопустимое, и это пока единственный известный нам случай такого значения.

Вспомним, что к объектам применимо две операции доступа: чтение и запись. В большинстве случаев, когда выражение, идентифицирующее объект, используется в выражении, в него необходимо подставить значение, хранимое в этом объекте. Это достигается за счёт стандартного преобразования, называемого ***преобразование леводопустимого выражения (lvalue-to-rvalue)***: обобщённое леводопустимое выражение может быть преобразовано к чисто праводопустимому того же типа со значением, считанным из идентифицированного им объекта. Тип выражения в этом случае определяет, как интерпретируются данные при чтении из памяти.

Обратное действие — запись — позволяет осуществить ***операция простого присваивания (simple assignment)*** `=`. Это инфиксная операция, требующая в качестве своего левого операнда ***модифицируемое (modifiable)*** леводопустимое выражение. ***Побочным эффектом*** этой операции является запись в объект, идентифицируемый её левым операндом, значения, задаваемого правым операндом, неявно преобразованного к типу левого. Значением самой операции является её левый операнд, по прежнему идентифицирующий объект, в который была произведена запись. Тот факт, что операция присваивания сама имеет значение-результат, а её основная задача — изменение значения указанного объекта — считается «всего лишь» побочным эффектом, может показаться странным, особенно тем, кто знаком с другими языками программирования. Эта операция раскрывает этимологию термина «леводопустимое выражение» — исторически так начали называть значения, которые допустимо использовать в качестве левого операнда операции присваивания.

> Не все выражения, идентифицирующие объекты, позволяют их изменять, но используемые нами пока имена объектов арифметических типов относятся к модифицируемым.

Рассмотрим примеры.

```
int a,b,c;
double d,e;
```

Перед нами два определения. Теперь в нашем распоряжении есть три объекта, имеющих достаточное место для хранения значений типа `int`, идентифицируемых как `a`, `b` и `c` — это строго формальная интерпретация. Обычно говорят проще (про второе описание): у нас также есть объекты `d` и `e` типа `double`. Эта формулировка используется повсеместно, но скрывает некоторые важные детали происходящего. Во-первых, сами идентификаторы `d` и `e` объектами не являются, а лишь используются для идентификации некоторых объектов в качестве их имён. В общем случае между именами (в частном случае, идентификаторами) и объектами нет отношения один-к-одному: одному объекту может соответствовать несколько выражений, в том числе более сложных, чем отдельное имя. Во-вторых, сам объект является просто некоторой областью памяти, которая может иметь разную интерпретацию. Тип объекта определяется типом выражения, которое используется для доступа к нему. В данном случае описание задаёт тип идентификатора, и именно его тип используется для выбора интерпретации содержимого

объекта, ему соответствующему, при чтении и записи его значения. Учитывая эти детали, будем прибегать к более простому объяснению происходящего там, где это не критично, чтобы укоротить текст.

Другим распространённым термином для имени объекта является «переменная». Автор считает, что хотя бы на начальном этапе обучения стоит вообще не упоминать этот термин. С ним связаны довольно глубокие математические ассоциации, которые скрывают в том числе приведённые выше детали о соотношении между именами и объектами, которые могут помешать на этом критическом этапе овладения языком. Покажем использование операции присваивания на практике.

```
a = 0;
```

Это пример одного из основных операторов языка C++ — оператора-выражения. ***Оператор-выражение (expression statement)*** имеет вид выражения, за которыми следует точка с запятой. Он предписывает вычислить содержащееся в нём выражение и отбросить результат (!).

В данном случае оператор-выражение содержит выражение, состоящее из одной операции простого присваивания. В качестве побочного эффекта процесса вычисления этого выражения значение 0 типа `int`, заданное литералом, являющимся правым операндом операции присваивания, будет записано в объект, идентифицируемый леводопустимым выражением в его левой части — идентификатором `a`. Поскольку тип идентификатора объекта `a` совпадает с типом записываемого значения, неявных преобразований не требуется. Результатом этой операции является выражение, вновь идентифицирующее объект `a`. Это значение всего выражения, содержащегося в операторе-выражении, которое он отбрасывает.

Столь странная формулировка, в которой основное по сути действие является побочным эффектом, обусловлена тем, что термин «побочный эффект» в языке C++ означает как раз полезную работу программы. Отбрасывание значений, вычисляемых выражениями в операторах-выражениях также является естественным: после того, как это значение вычислено, а все побочные эффекты возымели действие, занимаемое ими место может использоваться для других целей.

В известном нам синтаксисе, начальное значение определяемых объектов не задано и является ***неопределённым (indeterminate)***. Отметим важное правило, касающееся порядка операций над объектами с такими значениями: кроме редких случаев, указанных в стандарте, ***чтение значений объектов с неопределённым значением есть неопределённое поведение, поэтому первой операцией доступа к ним должна быть запись***.

```
b = c = 1;
```

Перед нами часто используемый в программах пример использования значения, возвращаемого операцией присваивания, который используется для записи одного и того же значения в несколько объектов сразу. Операция присваивания обладает ассоциативностью справа налево, что следует из синтаксического правила:

```
assignment-expression:
    conditional-expression
    logical-or-expression    assignment-operator    initializer-clause
  …

assignment-operator:    one of
  = *= /= %= += -= >>= <<= &= ^= |=

initializer-clause:
    assignment-expression
  …
```

Таким образом, первой операцией, вычисляемой в рассматриваемом выражении, является запись в `c` единицы. После записи в объект `c` единицы, значение этого выражения, соответствующее объекту с только что записанным значением, выступает в роли правого операнда второй (первой по порядку) операции присваивания с `b` в качестве левого операнда. Сначала происходит преобразование леводопустимого выражения, т.е. формально записанное значение тут же считывается, после чего записывается в объект, идентифицируемый `b`. Бояться этих «лишних» операций чтения не следует, т.к. это лишь формальное описание процесса, которое будет оптимизировано транслятором.

```
e = a = d = 3.7;
```

Рассмотрим этот пример на неявные приведения типов, происходящие в операциях присваивания. Значение 3.7 типа `double` вначале записывается в объект `d`. Это значение теперь необходимо записать в объект, содержимое которого следует истолковывать как `int` в соответствии с типом леводопустимого выражения `a`, являющегося именем соответствующего объекта. Происходит преобразование к типу `int` значения 3.7 типа `double` с результатом 3 типа `int`, который записывается в объект, идентифицируемый `a`. Наконец, это значение теперь необходимо записать в объект `e` типа `double`. Эта запись оставляет значение без изменений, поскольку значение 3 может быть в точности представлено типом `double`, поэтому неявное преобразование меняет только тип, а не само значение.

```
c = a+b;
```

Теперь, когда все объекты имеют конкретные значения, можно рассмотреть процесс чтения их значений. Данный оператор-выражение предписывает:

1. Первой операцией, в соответствии с правилами разбора выражения, является сложение. Её операнды — идентификаторы объектов, являющиеся леводопустимыми выражениями. Они преобразовываются в значения объектов, ими идентифицируемыми, считываемыми из них в соответствии с типами этих выражений. Считывания происходят в неуточняемом порядке.
2. Над операндами 3 и 1 типа `int` совершается операция сложения, результат которой — 4 типа `int`.
3. Вычисляется значение операции простого присваивания, оно идентифицирует объект `c` типа `int`. В качестве побочного эффекта происходит запись в него значения 4.
4. Вычисленное значение выражения, идентифицирующее объект `c`, отбрасывается.

Особо отметим, что данное выражение имеет принципиально другой смысл, нежели его математическое толкование. Знак `=`, соответствующий операции присваивания, не имеет никакого отношения к понятию «равенства», в том числе к операции сравнения на равенство `==`, также входящей в язык. Путаница между ними — обычная ошибка, которую могут допускать не только новички. Данная запись не является утверждением об отношении между значениями некоторых переменных, а задаёт последовательность действий, приведённую выше, состоящую из арифметических расчётов, а также чтения и записи значений в областях памяти среды выполнения, которые с точки зрения программиста имеют некоторые имена. Чтобы подчеркнуть это различие, многие языки программирования не используют для операции присваивания математический знак равенства, хотя в C++ это так.

> Операция `static_cast` с известными нам типами возвращает не леводопустимый результат, поэтому её формально можно использовать для явного выполнения преобразования леводопустимого значения за счёт приведения объекта к его собственному типу:
>
> ```
> 1  int x;
> 2
> 3  // ОШИБКА: результат приведения типов уже не леводопустим,
> 4  //         хотя и сохранил свой первоначальный тип.
> 5  static_cast<int>(x) = 3;
> ```
>
> Обычно смысла это делать нет.

4.3.4. Операции составного присваивания, инкремента и декремента

В приведённом выше правиле показаны и другие операции присваивания, помимо простого, — это операции ***составного присваивания (compound assignment)***. Они являются краткой формой записи часто встречаемых выражений с операцией присваивания, следующие две формы эквивалентны:

```
a = a  знак-операции b
a  знак-операции= b
```

Например, `a = a*2` можно записать короче в виде `a *= 2`. В этой записи `*=` — один токен, а не два, т.к. присутствует в списке операций языка. Составные операции присваивания существуют не для всех возможных операций, а только для указанных в правиле выше.

Поскольку прибавление и вычитание единицы встречается при счёте в алгоритмах очень часто, для выражений, соответствующих этому вычислению, есть ещё более короткая запись в виде операций ***инкремента (increment)*** `++` и ***декремента (decrement)*** `--`. Префиксные формы их записи `++x` и `--y` полностью эквивалентны выражениям `x += 1` и `y -= 1` соответственно. Имеются также постфиксные формы `x++` и `y--`. Они отличаются тем, что их значение соответствует значению объекта, идентифицируемому операндом, до изменения его значения этой операций, хотя побочный эффект их тот же. Так как результат вычисления операции не соответствует хранимому после её вычисления значению в объекте, результат праводопустимых форм операций инкремента и декремента не леводопустим:

```
1  int x,y;
2  x = 5;
3  ++x;     // x == 6
4  ++++x;   // разбирается на токены как ++, ++ и  x; x == 8
5  y = x++; // x == 9, y == 8
```

Выше показан типичный оператор, увеличивающий значение объекта на единицу — `++x`. Поскольку в нём значение выражения всё равно отбрасывается, выбор префиксной или постфиксной формы операции инкремента кажется неважным. Автор предлагает в таких ситуациях приучить себя пользоваться префиксной формой, поскольку она не создаёт дополнительных значений. Для арифметических типов, которыми мы оперируем сейчас, разницы в машинном коде не будет, но привычка поможет в будущем, когда для сложных типов данных формы эти операции будут заметно отличаться по производительности.

Итак, в наших руках механизм совершения арифметических расчётов: мы знаем, как формулировать на языке C++ алгоритмы (или их части), соответствующие последовательностям арифметических действий над заданными величинами, с возможностью сохранения промежуточных и итоговых результатов в памяти.

4.3.5. Пример компиляции выражения

Рассмотрим пример трансляции простого выражения в машинный код архитектуры x86, используя его запись на ассемблере в синтаксисе Intel. Пускай имеется следующий фрагмент кода:

```
1  unsigned x,y,z;
2  // ...
3  z = x+y;
```

Для хранения типа `unsigned` в модели памяти ILP32 требуется 4 байта, что соответствует длине машинного слова рассматриваемого 32-битного процессора. Когда говорят о хранении объектов, занимающих более одного байта, в качестве адреса объекта называют адрес младшего из непрерывной последовательности байтов, хранящих представление объекта. Предположим, что объекты, идентифицируемые `x`, `y`, `z` имеют адреса `0x3000`, `0x3004` и `0x3008` соответственно, а машинный код, соответствующий вычислению данного выражения, хранится начиная с адреса `0x1000`. Воспользовавшись, например, дизассемблером, встроенным в отладчик, можно получить следующую запись:

```
10000: a1 00 30 00 00      mov eax,[0x3000]
10005: 8b 1d 04 30 00 00   mov ebx,[0x3004]
1000b: 01 d8               add eax,ebx
1000d: a3 08 30 00 00      mov [0x3008],eax
```

Эта запись является комбинированным представлением, показывающим содержимое памяти в виде значений байтов, т.е. машинного кода с последующей расшифровкой в синтаксисе ассемблера: каждая строка содержит адрес первого байта инструкции, затем через двоеточие записаны значения байтов, ей соответствующих, после чего приведена запись на ассемблере. Все адреса и значения записаны в шестнадцатеричной системе. Инструкция ассемблера x86 состоит из мнемонической аббревиатуры, соответствующей требуемой команде, после которой через запятую перечислены её операнды (если они есть). В зависимости от конкретной команды, операнды могут быть целочисленными константами, именами регистров или более сложными выражениями, соответствующими одному из режимов адресации процессора.

Рассмотрим первую инструкцию. Инструкция `mov` (MOVe) используется для перемещения данных между регистрами процессора и/или памятью. Её имя несколько не соответствует происходящему, поскольку данные на самом деле копируются, а не перемещаются — при выполнении инструкции старое место хранения данных не изменяет своего значения. В синтаксисе Intel первый аргумент является местом назначения, а второй — источником. Архитектура x86 имеет 8 32-битных регистров общего назначения: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` и `ESP`. Префикс `E` обозначает, что это 32-битное расширение соответствующего 16-битного регистра (архитектура изначально разрабатывалась как 16-битная). Изначально каждому этому регистру приписывалось особое назначение, что отражено в их названии: Accumulator, Base, Counter, Data, Source Index, Destination Index, Base Pointer, Stack Pointer. Хотя многие из них по прежнему имеют специальные функции, все они, кроме `EBP` и `ESP` обычно могут использоваться для большинства операций. В данной инструкции источником является регистр `EAX`. В качестве источника указано выражение в квадратных скобках — это означает, что речь идёт не о самом значении, в нём записанном, а о содержимом памяти по этому адресу. Т.е. `[0x3000]` — «содержимое памяти по адресу 0x3000». Число байтов, о содержимом которых начиная с указанного идёт речь, определяется в данном случае неявно по размеру места назначения, в данном случае 4 байта. Итак, эта инструкция читается как «прочесть 4 байта, начиная с адреса `0x3000`, в регистр `EAX`».

Вторая инструкция аналогична первой и считывает 32-битное значение по адресу `0x3004` в регистр `EBX`. В обеих этих инструкциях видно, что 32-битный адрес, заданный в инструкции как ***непосредственный (immediate)*** операнд присутствует в машинном коде в little-endian формате. Несмотря на идентичную структуру, машинный код этих инструкций отличается даже длиной, поскольку для работы с регистром `EAX` в наборе команд x86 предусмотрены более короткие формы кодирования инструкции `mov`, а для второй инструкции используется общая форма.

Третья инструкция `add` осуществляет целочисленное сложение. Её операнды заданы именами регистров, куда уже считаны необходимые значения. Эта инструкция, как и большинство инструкций x86, записывает результат в первый из операндов, который в данном случае является одновременно источником и местом назначения. Эта инструкция читается как «прибавить к содержимому регистра `EAX` содержимое регистра `EBX`».

Последняя инструкция также соответствует операции перемещения, которая осуществляет запись содержимого регистра `EAX` по адресу `0x3008`.

Один и тот же код на машинно-независимом языке может быть преобразован в машинный код множеством эквивалентных по поведению способов. Например, запросив у транслятора оптимизацию кода, получим для данного примера:

```
10000: a1 00 30 00 00      mov eax,[0x3000]
10005: 03 05 04 30 00 00   add eax,[0x3004]
1000b: a3 08 30 00 00      mov [0x3008],eax
```

Эта версия прибавляет к первому слагаемому, перенесённому в регистр, второе прямо из памяти, без промежуточного копирования во второй регистр. Такой вариант занимает меньше памяти, почти наверняка выполняется быстрее, и использует меньшее число регистров, которые на архитектуре x86 являются очень ценным ресурсом из-за их малого количества. Современные компиляторы способны выполнять куда более сложные преобразования, которые часто могут быть неочевидными даже для специалистов. Укоротить сложение двух слагаемых в памяти больше не получится — архитектура x86 не позволяет иметь больше одного операнда в форме доступа к памяти в каждой инструкции, а сама задача подразумевает три обращения к ней.

Хотя преобразования леводопустимых выражений с точки зрения языка всегда связаны с операциями чтения, а операции присваивания — с записью, соответствующий машинный код не обязан содержать соответствующих инструкций доступа к памяти, поскольку одним из направлений оптимизации кода является как раз минимизация обмена данными между регистрами и памятью. Это возможно потому, что регистры процессора не являются рассматриваемой с точки зрения языка сущностью. Ещё раз обратим внимание: пока результирующий код выполняет операции согласно контракту, закреплённому стандартом языка, компилятор имеет право выдавать его в любом соответствующем виде. Именно поэтому чрезвычайно важно знать, что в этом контракте является гарантируемым (определённое поведение), а что — нет: неуточняемое поведение, на конкретный

> вариант которого нельзя рассчитывать, и неопределённое поведение, которое в глазах неопытного программиста может ложно выглядеть неизвестным ему, но стабильным элементом поведения, или не наблюдаться вообще.

4.4. Введение в функции

Даже при записи простейших последовательных вычислительных алгоритмов часто возникают ситуации, когда та или иная последовательность действий встречается более одного раза. Это типичная ситуация, где следует воспользоваться функциями. Преимущества использования функций таковы:

- Заданная функцией последовательность действий получает имя, которое облегчает понимание смысла программы её читателями.
- Исчезает необходимость повторной записи одних и тех же действий, что сокращает объём программы и уменьшает вероятность возникновения ошибки как при первичной записи, так и при последующих изменениях алгоритма.
- Функции могут выделяться в библиотеки подпрограмм, способствуя повторному использованию кода как в рамках одной программы, так и между различными программами.

4.4.1. Определения функций

С функцией в языке C++ связаны:

- Имя функции, данное ей программистом, которое может использоваться для её идентификации.
- Количество параметров функции и тип каждого из них — объекты, начальные значения которых указываются при каждом вызове и к которым она имеет доступ.
- Тип возвращаемого функцией значения — тип значения (всегда ровно одного), которое становится доступно в точке вызова данной функции по завершению её исполнения в качестве «результата».

Описание функции связывает с именем функции тип, имеющий перечисленные характеристики. Определение функции помимо этого задаёт саму последовательность действий, ей соответствующую. Кроме самих действий, функции могут требоваться дополнительные описания объектов, которыми она оперирует. ***Блок (block)*** или ***составной оператор (compound statement)*** позволяет группировать последовательность операторов в одну синтаксическую единицу, внутри которой они исполняются последовательно по порядку, если не указано обратное. Содержимое блока записывается в фигурных скобках. Определение функции содержит её ***тело (body)***, являющееся составным оператором.

Разберём пример определения функции, реализующей подсчёт объёма детали заданной толщины квадратного сечения с круглым отверстием:

```
1  double part_volume(double size,
2                     double hole_radius,
3                     double thickness)
4  {
5      double part_surface,hole_surface;
6      part_surface = size*size;
7      hole_surface = 3.1415926*hole_radius*hole_radius;
8      return (part_surface-hole_surface)*thickness;
9  }
```

Первая строка содержит собственно описание идентификатора `part_volume`. Данное описание говорит о типе `double`, соответствующему конструкции `part_volume(double size,double hole_radi` являющейся сложным описателем. Такая запись является описанием функции, указанный в начале строки тип соответствует её возвращаемому значению, а содержимое скобок говорит о её параметрах. Параметры перечисляются через запятую с указанием идентификаторов, которые будут им соответствовать, и типа тем же образом, что и в описаниях идентификаторов объектов. Каждый параметр является отдельным описанием, так что имя типа необходимо указывать для каждого параметра, даже если оно одинаково для нескольких подряд идущих. Эти имена могут использоваться в теле функции для идентификации объектов, хранящих значения аргументов, заданных при очередном вызове функции. Таким образом, данное описание соответствует функции, идентифицируемой как `part_volume`, имеющей три параметра

типа `double`, значения которых хранятся в объектах, идентифицируемых в её теле как `size`, `hole_radius` и `thickness` соответственно, и возвращающей значение типа `double`.

Определение функции может содержать только один описатель и завершается не точкой с запятой, а телом функции — составным оператором. Его содержимым являются строки 5–8, задающие последовательность операторов, реализующих часть алгоритма, соответствующего данной функции. Строка 5 содержит определение двух объектов типа `double`, идентифицируемых `part_surface` и `hole_surface`, используемых для хранения промежуточных значений вычислений. Описания в языке C++ могут считаться операторами. Разбиение вычислений на несколько шагов в данном случае не является необходимым, но сделано для примера. Строка 6 содержит оператор-выражение, вычисляющий площадь сечения детали без учёта выреза и записывающий его в объект, идентифицируемый `part_surface`. Строка 7 вычисляет и записывает площадь отверстия в объект `hole_surface`.

Идентификаторы, соответствующие объектам, хранящим значения параметров функции, а также из всех определений в её теле (в том виде, в котором они известны нам на данный момент), могут использоваться для именования соответствующих им объектов только до окончания тела данной функции, т.е. конкретного блока. Идентификаторы параметров функции относятся к блоку-телу функции, несмотря на то, что в тексте программы расположены не внутри соответствующих фигурных скобок. Память, необходимая для хранения значений объектов, соответствующих этим идентификаторам, выделяется автоматически в начале выполнения блока (***входе (enter)*** в блок), в котором они содержатся, и освобождается по завершении (***выходе (exit)*** из блока). Эти свойства соответствуют двум атрибутам идентификаторов, задаваемым в описаниях, помимо типа: блочной ***области видимости*** и ***автоматическому*** времени хранения. Мы рассмотрим их подробно в отдельном разделе.

Последняя строка тела функции содержит новый для нас оператор `return`. ***Оператор возврата из функции (return statement)*** `return` задаёт возвращаемое функцией значение и завершает её выполнение. Он имеет форму ключевого слова `return`, за которым следует выражение, а затем точка с запятой. Значение выражения и задаёт возвращаемое функцией значение. Если его тип отличается от описанного возвращаемого значения функции, происходит неявное преобразование типа. Выполнением оператора `return` завершается выполнение функции. При записи алгоритмов, операторы которой выполняются не последовательно, оператор `return` может встречаться несколько раз, в том числе в середине тела функции.

Определения функций не могут встречаться внутри других определений функций — пока наша единица трансляции будет состоять из последовательности определений функций.

4.4.2. Операция вызова функции

Как и большинство имён, имя функции может использоваться только после своего описания, например, в виде определения функции. В наших первых программах оно может быть использовано начиная с места описания и до конца данного файла исходного текста, например, в других функциях, которые следуют за ней. Приведём пример такого использования:

```
1  double weight_diff_factor(double size,
2                            double old_radius,
3                            double new_radius,
4                            double thickness)
5  {
6      return part_volume(size,new_radius,thickness)/
7             part_volume(size,old_radius,thickness);
8  }
```

В данном примере используется операция ***вызова функции (function call)***. Её синтаксис состоит из значения, идентифицирующего вызываемую функцию, за которым следует список аргументов, разделённых запятыми, в круглых скобках. Пока единственная известная нам форма такого значения — имя ранее определённой функции. Это пока ограничивает нас необходимостью располагать функции в определённом порядке. Формально операция вызова функции имеет арность, равную числу передаваемых вызываемой функции ***аргументов (argument)*** — значений параметров — плюс один, необходимый для идентификации вызываемой функции. Операция вызова функции осуществляет следующие действия:

1. Происходит вычисление значений всех аргументов в неуточняемом порядке.
2. Происходит приостановка выполнения текущей функции, и управление передаётся функции, идентифицируемой первым операндом (перед скобками) операции вызова функции.

3. Перед началом выполнения вызываемой функции объекты, соответствующие её параметрам, получают начальные значения, вычисленные на первом шаге.
4. Происходит выполнение вызванной функции до окончания её тела или выполнения ей оператора `return`.
5. Значение выражения в операторе `return` в вызываемой функции после приведения к типу возвращаемого функцией значения становится результатом операции вызова функции, и выполнение функции, содержащей эту операцию, продолжается. Пока мы будем рассматривать функции, которые должны завершаться оператором `return`.

Круглые скобки могут означать не только операцию вызова функции, но и группировку операций в сложных выражениях, однако любое синтаксически корректное использование любой из этих конструкций имеет однозначную трактовку. Рассмотрим процесс выполнения функции `weight_diff_factor`:

1. В некотором месте программы вычисляется операция вызова функции с `weight_diff_factor` в качестве первого операнда и четырьмя значениями типа `double` в качестве остальных. Выделяется память под хранение четырёх объектов, идентифицируемых `size`, `old_radius`, `new_radius` и `thickness`, которые получают эти значения как при присваивании. Функция получает управление.
2. Единственный оператор в теле функции — `return` начинает своё выполнение с вычисления значения выражения, в нём содержащегося. Для выполнения операции деления необходимо вычислить значения её операндов, порядок выполнения которых не уточняется. Предположим, что вначале вычисляется значение левого операнда.
3. Левый операнд является операцией вызова функции `part_volume` с аргументами `size`, `new_radius` и `thickness`. Значения этих аргументов вычисляются в неуточняемом порядке. Выполнение функции `weight_diff_factor` приостанавливается и начинается выполнение функции `part_volume`.
4. Создаются объекты, соответствующие идентификаторам `size`, `hole_radius`, `thickness` и им присваиваются значения аргументов, вычисленные на предыдущем шаге. Начинается выполнение тела функции.
5. При входе в блок, являющийся телом функции, выделяется память под объекты, идентифицируемые `part_surface` и `hole_surface`. Доступ к этим объектам возможен после того, как будут даны их описания. Операторы функции выполняются по порядку.
6. Управление доходит до оператора `return`. Выражение, записанное в нём, вычисляется и его результат запоминается. Происходит завершение выполнения этой функции, что приводит к выходу из блока, являющегося её телом. Как следствие, освобождается память, выделенная для хранения всех объектов с автоматическим временем хранения, соответствующим этому блоку: трём параметрам функции и двум, соответствующим определениям в её теле.
7. Возобновляется выполнение функции `weight_diff_factor`. Запомненное значение выражения в операторе `return` из функции `part_volume` становится значением первой операции вызова функции и левым операндом операции деления.
8. Вторая операция вызова функции, являющаяся правым операндом операции деления, вычисляется аналогичным образом, включая процесс выделения и освобождения памяти под соответствующие объекты. Эти объекты никак не связаны с объектами из прошлого вызова той же функции, хотя определены в ней же под теми же именами. Результатом этой операции является возвращённое функцией `part_volume` для аргументов `size`, `old_radius` и `thickness` значение.
9. Выполняется операция деления, результат которой является значением выражения в операторе `return` в функции `weight_diff_factor`. Это значение возвращается в качестве результата операции вызова этой функции туда, где был совершён этот вызов. Перед этим освобождается память под хранение объектов-параметров данной функции.

Следует чётко понимать разницу между параметрами и аргументами: параметры есть объекты, хранящие значения, переданные функции при её вызове и доступны только внутри неё. Аргументы функции — значения операндов операции вызова функции в круглых скобках, которые задают значения параметров вызываемой функции. Таким образом, параметры и аргументы являются внутренней и внешней сторонами механизма передачи функции значений. В литературе могут также встречаться следующие термины для обозначения аргументов и параметров функции — фактические и формальные аргументы или параметры соответственно.

В этой книге автор будет придерживаться терминов «аргумент» и «параметр», используемых в стандарте языка C++.

Одна из функций программы имеет специальную роль функции, выполнение которой и составляет выполнение всей программы — ***функция, именуемая `main`***. На её параметры и возвращаемое значение имеются определённые ограничения, накладываемые стандартом. Простейшая её форма — отсутствие параметров и тип возвращаемого значения `int`.

```
1  int main()
2  {
3      return 0;
4  }
```

Это пример простейшей, ничего не делающей программы. Возвращаемое значение функции `main` является признаком успешности завершения её работы, возвращаемым операционной системе, и обычно является нулём в случае успешной отработки программы.

В любой момент выполнения программы в ней имеется цепочка функций, начинающаяся с `main`, состоящая из функций, выполнение которых приостановлено на время выполнения другой, запущенной операцией вызова функции. Последняя функция в этой цепочке не приостановлена, а выполняется в данный момент. Эта последовательность называется ***стеком вызова (call stack)***. Рассмотрим пример, в котором используются процедуры из данного раздела:

```
1  int main()
2  {
3      weight_diff_factor(10.,2.,2.5,3.);
4      return 0;
5  }
```

Приведём пример стека вызова на момент первого выполнения программой функции `part_volume`:

```
part_volume(size=10,hole_radius=2.5,thickness=3)
weight_diff_factor(size=10,old_radius=2,new_radius=2.5,thickness=3)
main()
```

Данный формат соответствует тому, который обычно используется интерактивными отладчиками: функции приведены по порядку, начиная с активной, в скобках после их имён перечислены имена параметров функции и переданные в соответствующей операции вызова функции аргументы.

4.5. Введение в форматированный ввод/вывод

Рассмотренные нами в данной главе возможности позволяют формулировать последовательные вычислительные алгоритмы из арифметических операций над заданными константами, с возможностью многократного использования как частей алгоритмов в виде функций, так и значений, сохраняемых в объектах. Единственный способ «связи с внешним миром», который был упомянут — возвращаемое значение функции `main`, чего явно не достаточно для ведения полноценного диалога с пользователем в каком-либо виде.

Одним из основных принципов, которые легли в основу операционной системы UNIX, для разработки которой создавался язык C, является понятие ***файла (file)***. Речь идёт не только о поименованном участке устройства долговременного хранения данных — большинство физических и логических устройств также рассматриваются как «файлы» — сущности, к которым применимы операции чтения и записи данных. В данном контексте термины «чтение» и «запись» синонимичны понятиям «ввод» и «вывод», а подразумевается под ними обмен данными от устройства к программе и наоборот соответственно. Единицей обмена выступают байты или узкие символы.

Классическими устройствами ввода/вывода являются клавиатура и устройство печати текста. Обычно, если отсутствуют специальные указания, программа на языке C++ использует виртуальный ***терминал (terminal)*** для общения с пользователем, который в операционных системах с оконным интерфейсом является отдельным окном, представляющим программу. Термин «виртуальный» говорит о том, что терминал является элементом интерфейса ОС, а не отдельным физическим устройством, каковым он обычно был десятки лет назад, в те времена, когда он являлся основным способом взаимодействия человека с машиной. Окно

терминала разбито на фиксированное количество строк и столбцов для отображения текста моноширинным шрифтом. Символы, вводимые с клавиатуры, когда оно является активным, могут быть считаны программой. Они также отображаются в окне терминала. Данные, выводимые программой, отображаются в окне терминала в виде соответствующих символов из набора среды выполнения. Они выводятся последовательно, при этом текущее место, куда будет выведен очередной символ, записанный программой или введённый пользователем, отмечается курсором. По окончании строки, или ранее, если программа пытается вывести символ конца строки, курсор перемещается в начало следующей строки. После заполнения последней строки терминала все строки сдвигаются вверх, чтобы освободить место для новой пустой строки в самом низу, самая верхняя строка при этом теряется или добавляется в историю окна терминала, если он поддерживает такую возможность. При работе программы под управлением отладчика или в интегрированной среде разработки, терминал может быть частью их интерфейса, а не отдельным окном.

Вернёмся к первой полной программе, приведённой ранее и постараемся в ней разобраться:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Input two real numbers: ";
6      double a,b,c;
7      std::cin >> a >> b;
8      c = a*b;
9      std::cout << a << " * " << b << " = " << c << '\n';
10     return 0;
11 }
```

Данный исходный текст начинается с директивы предварительной обработки, включающей в текст единицы трансляции описания необходимых для осуществления ввода-вывода имён.

Далее следует стандартное определение функции `main`, тело которой содержит весь алгоритм, реализуемый данной программой, поскольку других функций в ней не определено.

4.5.1. Форматированный вывод

Первая строка тела функции `main` содержит оператор-выражение, содержащий неизвестные нам имена и операции.

Последовательность токенов `std::cout` включает идентификатор `std`, операцию `::` и идентификатор `cout`, которые вместе составляют имя объекта. Это первый для нас пример имени, не являющегося отдельным идентификатором. Описание данного имени было включено в эту единицу трансляции директивой из её первой строки и является именем объекта, соответствующего ***стандартному потоку вывода (standard output stream)*** и может применяться для вывода текста из программы. Мы пока не можем дать осмысленное описание типа этого объекта, поэтому будем описывать его свойства формально в терминах применимых к нему операций и их свойств.

Бинарная инфиксная операция `<<`, левым операндом которой является объект, именованный `std::cout`, осуществляет в качестве побочного эффекта вывод значения, заданного её правым операндом. В качестве правого операнда могут использоваться значения всех известных нам типов, которые будут выведены в виде, сходным с записью соответствующих литералов, за исключением `bool`, который будет выведен в виде 0 или 1, а для символьных типов выводится символ, соответствующий числовому коду, хранящемуся в них. Кроме этого, в качестве правого операнда допускаются значения, к которым неявно могут быть преобразованы значения, задаваемые строковыми литералами. Побочный эффект таких операций — вывод содержимого, заданного строковым литералом, что и используется на рассматриваемой строке. В случае, когда требуется вывести один символ, следует предпочесть символьный литерал строковому, такая операция эффективнее.

Результат вычисления этой операции — значение, вновь идентифицирующее объект `std::cout`, это свойство вместе с ассоциативностью операции `<<` слева направо может использоваться для вывода нескольких значений по цепочке, как показано на строке 9.

4.5.2. Форматированный ввод

Парным к объекту `std::cout` является объект `std::cin`, соответствующий ***стандартному потоку ввода (standard input stream)***, позволяющий осуществить ввод данных

в программу. Для этого он используется в качестве левого операнда бинарной инфиксной операции `>>`, правый операнд которой должен быть леводопустимым выражением, идентифицирующим объект, в который будет записано считанное значение после его интерпретации из последовательности введённых символов. Эта операция тоже возвращает левый операнд как леводопустимое значение и также может применяться по цепочке в одном выражении. Формат вводимых данных определяется типом правого операнда и для арифметических типов практически совпадает с синтаксисом соответствующих литералов за следующими исключениями:

- Перед началом интерпретации символов, соответствующих вводимому значению, все пробельные символы пропускаются.
- Префиксы систем счисления, суффиксы типов и одиночные кавычки-разделители не распознаются, ввод осуществляется всегда в десятичной системе.
- Ввод, соответствующий целочисленному литералу, принимается и при вводе значений с плавающей точкой.
- Перед вводимыми числами допускаются знаки `+` и `-` для задания знака значения.
- При вводе значений символьных типов считывается один символ, и в объект заносится соответствующий ему код.

Ввод завершается на первом символе, который не подходит под ожидаемый формат. Что происходит в ситуации, когда ввод значения завершается ошибкой, в том числе, когда первый непробельный символ не подходит, мы рассмотрим в следующем разделе, пока будем наивно исходить из того, что он всегда успешен, т.е. пользователь всегда вводит значения в требуемом формате и они представимы в соответствующем типе. Тем не менее в любых программах, претендующих больше, чем на звание простейшего примера, данные, считываемые из файлов, включая вводимые пользователем через стандартный поток ввода, подлежат обязательной проверке. Отсутствие таковых является чрезвычайно распространённой ошибкой, влекущей за собой, в том числе и проблемы безопасности.

За счёт того, что пробельные символы пропускаются в начале ввода и не входят в представление ни одного из известных нам типов, при вводе нескольких значений они могут разделяться любыми последовательностями пробельных символов. Стандартные настройки взаимодействия с терминалом приводят к тому, что символы, вводимые с клавиатуры, становятся доступными программе только целыми строками — пока не нажата клавиша `Enter`, программа «не увидит» введённых на этой строке значений, поэтому именно перевод строки следует вводить по окончании очередной порции ввода, чтобы вычисление операции `>>` завершилось и программа могла продолжить работу. Вывод программы также отображается целыми строками, завершающимися символом новой строки. В показанном примере мы пользуемся исключением из этого правила: неполная выводимая строка будет полностью отображена перед запросом ввода у пользователя, что позволяет размещать приглашения к ввода и пользовательский ввод на одной строке.

Смысл остальных строк программы теперь должен быть приблизительно ясен: строка 6 определяет 3 объекта типа `double`, строка 8 производит вычисление произведения значений первых двух из этих объектов с записью результат в третий, а последняя строка тела функции `main` выполняет завершение её выполнения. Таким образом, данная программа выполняет следующие действия: выводит приглашение к вводу двух чисел, выделяет память под хранение трёх чисел, производит ввод двух числовых значений с записью их в объекты, вычисляет произведение записанных значений с записью результата в третий объект, выводит текст, содержащий запись проведённых вычислений со всеми тремя значениями, и завершается. Строго говоря, объект `c` и отдельный оператор-выражение, содержащий запись в него значения, не являются необходимыми, поскольку выражение `a*b` могло быть напрямую указано операндом одной из операций `<<`.

4.6. Использование среды Qt Creator

Практический опыт программирования является основой выработки этого навыка в любой форме. Например, даже вдумчивое прочтение этой книги без попыток написать что-либо самостоятельно будет практически бесполезно. Автор не знает, как ещё больше подчеркнуть эту мысль: ***только практика даст вам возможность овладеть навыками и теорией языков программирования — ищите время и поводы программировать даже там, где от вас этого явно не просят, иначе вы рискуете потерять нить***

понимания происходящего, которую потом очень трудно найти. Учитывая это, перейдём к обсуждению практической работы с инструментальными средствами.

Как было рассмотрено ранее, интегрированные среды разработки являются связующим элементом рабочего места программиста. Их использование особенно удобно в начале изучения дисциплины, поскольку это позволяет приступить к написанию программ без детального разбора функционирования всех вовлечённых в процесс трансляции инструментов.

Автор предлагает читателям воспользоваться средой Qt Creator. В рамках авторского курса распространяется образ виртуальной машины, в котором правильно установлены и настроены все необходимые инструментальные средства. Автор настоятельно рекомендует пользоваться этим вариантом, если имеется возможность, чтобы на начальном уровне не отвлекаться на технические проблемы.

Среда Qt Creator работает на всех трёх основных операционных системах, при этом она рассчитана на использование схожих и современных компиляторов изучаемых языков. В этом тексте будем считать, что вы пользуетесь образом виртуальной машины, приготовленной для данного курса и всё необходимое ПО уже установлено.

Теперь можно запустить сам Qt Creator. Первым делом, если язык среды оказался русским, автор ***настоятельно рекомендует*** зайти в меню Инструменты → Параметры... и в разделе «Среда» выбрать английский интерфейс пользователя — этим вы серьёзно упростите себе жизнь. На всякий случай автор будет приводить имена интерфейсных элементов на обоих языках.

Даже для простой программы из одной единицы трансляции требуется создание ***проекта (project)***. Проект по сути является объединением информации, необходимой системе сборки (единицы трансляции, входящие в программу, опции их сборки, ...) и настроек среды разработки (сценарии трансляции и отладки программы, настройки редактора, ...). Меню «Файл (File)» содержит команды по созданию, открытию, сохранению и закрытию проектов. Для создания проектов наших первых программ следует воспользоваться командой Файл → Новый файл или проект... → Проект без использования Qt — Простой проект на C++ (File → New File or Project... → Non-Qt Project — Plain C++ Project).

На первой странице мастера потребуется задать имя и расположение проекта. Проект состоит из множества файлов, которые следует организованно хранить в отдельном выбираемом каталоге. Для хранения каталогов проектов автор рекомендует выделить отдельный каталог. Если вы работает в Windows, убедитесь на всякий случай, что в его полном пути не содержится пробелов и русских букв (например, подпапка «Моих документов» не подойдёт, особенно если у вас русское имя пользователя) — это поможет избежать возможных проблем. Имя проекта впоследствии станет именем исполняемого файла программы.

На следующей странице необходимо выбрать комплект инструментальных средств, которые будут использоваться при разработке программы. Обычно единственный и уже выбранный комплект является подходящим. На последней странице мастера указаны начальные файлы проекта и предлагаются дополнительные и не нужные нам возможности, так что можно опять не внося закончить создание проект.

Основное окно среды Qt Creator содержит переключатель режимов вдоль левой части окна, а остальное пространство занимает несколько окон в зависимости от текущего режима, которые могут быть дополнительно разбиты или переключены в другой режим отображения. Потратьте некоторое время на знакомство с доступными возможностями интерфейса среды, чтобы чувствовать себя в нём уверенно. Если какое-то из окон «потерялось», вернуть его можно через одну из опций меню «Окно (Window)».

Представление «Проекты (Projects)» отображает основные файлы, входящие в текущий проект в виде дерева по категориям. Простой проект на языке C изначально состоит из трёх файлов: файла проекта (имя совпадает с именем проекта, расширение `.pro`), который содержит информацию для системы сборки, файла исходного текста первой и пока единственной единицы трансляции (`main.cpp`), в котором содержится набившая оскомину программная «рыба». Все эти файлы содержат чистый текст и могут быть отредактированы любым текстовым редактором (не путать с текстовым *процессором*, например, Microsoft Word). Также был создан двоичный файл `имяпроекта.pro.user`, который содержит настройки среды разработки, который не считается относящимся к самому проекту по сути и потому в этом дереве не отображается. Он содержит настройки среды разработки, локальные для текущего компьютера.

В левом нижнем углу интерфейса среды находятся три кнопки, отвечающие за сборку программы: «Запустить (Run)», «Начать отладку (Start Debugging)» и «Собрать проект (Build Project)» (для них имеются аналогичные команды в меню). Последняя предписывает системе сборки выполнить необходимые команды для получения готовой к запуску программы,

если при этом возникнут проблемы, они будут отображены в одноимённом окне («Issues»). Сообщения, выдаваемые инструментальными средствами, называются ***диагностическими (diagnostics)***. Чаще всего это синтаксические и семантические ***ошибки (error)***, обнаруженные компилятором, наличие хотя бы одной из них приведёт к тому, что программа оттранслирована не будет. Тем не менее, после обнаружения одной из них, компилятор будет пытаться просмотреть текст вашей программы дальше, чтобы обнаружить в нём ещё ошибки и показать вам их все сразу. Часто он сбивается и показывает не существующие ошибки, которые исчезают после исправления первых. Существуют ***фатальные ошибки (fatal error)***, настолько серьёзные, что компилятор не пытается найти другие после них, а сразу останавливается. Ознакомится с командами, выполняемыми средой и системой сборки и диагностическими сообщениями в сыром виде можно в окне «Консоль сборки (Compile Output)». После обнаружения ошибок следует ознакомиться с текстом сообщения о ней, поскольку оно часто является прямым указанием к её исправлению — английские варианты терминологии приводятся в этой книге в том числе и для того, чтобы читатель мог найти общий язык с инструментальными средствами. Почти все ошибки привязаны к конкретному месту в тексте программы, к которому можно перейти по двойному щелчку на тексте её описания в окне «Проблемы (Issues)». После первого внесения изменения и попытки сборки программы вы, скорее всего, получите сообщение о том, что имеются не сохранённые файлы, изменения в которых не будут учтены при сборке. В этом окне рекомендуется поставить флажок, чтобы оно больше не появлялось, а все файлы сохранялись перед сборкой автоматически — работать с программой, которая не соответствует тексту на экране попросту нелогично.

Команда «Запустить» выполняет сборку проекта, и, если она успешна, запускает готовую программу. Для программ с консольным интерфейсом среда Qt Creator предоставляет дополнительное удобство — после завершения их работы появляется предложение нажать `Enter`, чтобы была возможность просмотреть вывод программы перед закрытием её окна. Будьте внимательны при переключении между работающей программой и средой разработки: можно забыть, что программа все ещё исполняется, и пытаться собрать её новую версию — в некоторых ОС этого сделать не удастся, пока прошлая версия программы ещё работает, а в некоторых получится, и вы запутаетесь в нескольких одновременно работающих экземплярах. В зависимости от наличия в вашей программе вечных циклов, неопределённого поведения или других ошибок, вы можете получать сообщения от операционной системы об аварийном завершении работы программы, или вам может понадобится воспользоваться её средствами принудительного закрытия приложений.

Команда «Начать отладку» аналогична команде «Запустить», но собранная программа запускается не отдельно от среды разработки, а под управлением интегрированного отладчика. В этом случае происходит автоматическое переключение в режим «Отладка (Debug)». Большинство рассматриваемых далее команд расположены в одноимённом меню. Как нам известно, отладчик позволяет приостанавливать процесс выполнения программы, чтобы изучить её состояние в необходимый момент времени. Когда программа остановлена, строка программы, которая должна быть исполнена следующей, отмечается жёлтой стрелкой на левом поле, а команда «Начать отладку» заменяется на «Продолжить (Continue)», которая возобновляет выполнение программы. При работе с интегрированным отладчиком остановка происходит в следующих случаях:

- По достижении ***точки остановки (breakpoint)***. Точки останова устанавливаются на строку кода щелчком по пустому месту левого поля редактора текста слева от номера строки и отмечаются красным кругом. Список всех точек останова содержится в одноимённом окне, позволяющим редактировать их свойства: отключать их временно, устанавливать дополнительные условия их срабатывания и др. Также имеется команда «Выполнить до строки (Run to Line)», которая эквивалентна установке точки останова на указанную строку, продолжению выполнения программы, а затем снятию этой точки.
- По окончании выполнения одной из команд шага. Команды шага вызываются во время остановки программы и приводят к исполнению некоторой, обычно малой, её части с повторной остановкой после этого. Команда «Перейти через (Step Over)» выполняет одну строку программы независимо от её содержания. Команда «Войти в (Step Into)» выполняет одну строку программы, если она не содержит вызовов функции, иначе она останавливается на первой строке первой вызванной функции. Войти таким образом внутрь функций стандартной библиотеки может получаться не всегда, поскольку в вашей среде может быть не доступен их исходный текст и требуемая отладочная информация. Команда «Выйти из функции (Step Out)» выполняет программу до возврата из текущей функции — эта возможность удобна, если вы случайно вошли в функцию, которая вам не интересна.

- При возникновении в вашей программе ошибочной ситуации, которая бы привела к аварийному её завершению её работы операционной системой, если бы она работала не под управлением отладчика. Обычно в таком случае вы получите сообщение о том, что программой получен сигнал от ОС и программа будет остановлена, чтобы вы могли изучить её состояние в попытке выполнить ошибочную инструкцию. Продолжить выполнение программы в такой ситуации нельзя — это тут же приведёт к повторному возникновению ошибки.
- По команде «Прервать (Interrupt)», которая заменяет команды «Начать отладку» и «Продолжить», когда программа работает. Эта команда позволяет приостановить выполнение работы программы в любой момент. Если только ваша программа не выполняет большого объёма вычислений, вы, скорее всего, остановитесь где-то глубоко в библиотечной функции, и чтобы добраться до написанного вами кода, вам потребуется неоднократное применение команды «Выйти из функции». Из-за этого этот способ применяют только в крайнем случае, если у вас есть подозрение на вечный или по ошибке черезвычайно медленный цикл.

Если вы ожидаете остановки программы на точке остановки или после команды шага, но его не происходит, возможно, поток управления пошёл не так, как вы ожидали. Наиболее вероятным в данной ситуации является то, что программа ожидает ввода данных пользователем. Например, если на строке с вычислением операции `>>` над объектом `std::cin` дать команду «Перейти через», программа остановится только после того, как вы переключитесь в её окно и введёте требуемые данные.

Если после остановки программы вы видете ассемблерный листинг вместо исходного текста, вероятнее всего остановка произошла в месте, для которого у отладчика нет информации о соответствующем ему коде на языке более высокого уровня. По желанию переключиться в этот вид можно командой «Уровень инструкций (Operate by Instruction)», в таком случае вместе с инструкциями ассемблера будет отображаться соответствующий им исходный текст. Для работы в этом виде удобно также вызвать из меню Окно → Обзоры (Window → Views) представление «Регистры (Registers)».

При работе программы под отладчиком, вы можете принудительно прервать её выполнение в любой момент командой «Остановить отладчик (Stop Debugger)». Этот способ предпочтительнее завершения программы средствами ОС, которые могут не работать вовсе, когда программа приостановлена отладчиком. Будьте внимательны: не забывайте завершать выполнение программы (штатным образом или с помощью отладчика) перед внесением в неё изменений, иначе текст программы на экране и то, что реально продолжает выполнятся перестанет соответствовать друг другу, что может привезти к конфузам.

Когда программа остановлена, интегрированная среда отображает большое количество информации о её состоянии, получаемой от отладчика. Помимо отметки текущей строки, которая должна быть выполнена следующей, одно из окон режима отладки отображает весь стек вызовов программы, в котором по двойному щелчку можно перейти к соответствующему месту вызова функции. Наиболее же ценной, пожалуй, является информация о значениях всех объектов, идентификаторы которых видимы в текущем месте выполнения программы. Контекстное меню этого окна позволяет менять форматы вывода значений, добавлять туда произвольные выражения и устанавливать контрольные точки (data breakpoint) — точки останова специального вида, которые срабатывают в момент изменения значений отслеживаемых объектов. В этом окне также можно напрямую изменить значения объектов, что иногда может быть полезно в отладочных целях.

В этом разделе перечислены далеко не все функции отладчика, в любом случае автор рекомендует потренироваться в его применении на простой и понятной программе, прежде чем вам придётся его использовать для нахождения действительно сложной проблемы, которую не удаётся отладить другим путём. Помните, что влезая во внутренности программы вы сможете наблюдать множество эффектов конкретной реализации, не описываемых в рамках стандарта языка. К сожалению, стандартный отладчик, используемый Qt Creator не всегда и не на всех операционных системах работает полностью стабильно. Поэтому не стоит забывать о самом древнем методе отладки не требующем никакого отладчика: ***отладочной печати*** — добавлению в текст программы дополнительных операций вывода, позволяющих отслеживать процесс её выполнения и выводящих те или иные значения объектов для контроля их правильности.

4.6.1. Настройка компилятора

Многие компиляторы способны на куда большую диагностику возможных проблем, в первую очередь типичных ошибок программистов, которые не являются ошибками синтаксиса и семантики языка. В таких случаях компилятор выдаёт ***предупреждения (warning)***. Хотя наличие предупреждений без единой настоящей ошибки не является поводом для отказа транслировать программу, выводимые предупреждения следует внимательно изучать, поскольку они могут подсказать наличие ошибок в программе, в первую очередь, труднообнаружимого неопределённого поведения. На начальных этапах обучения программированию рекомендуется полностью устранять все предупреждения, попутно выясняя мотивацию их выдачи. Для компиляторов `gcc` и `clang`, используемых по умолчанию в среде Qt Creator можно рекомендовать следующие опции:

- `-std=c++1z`: включает поддержку последней версии стандарта языка C++. Необходимая опция, поскольку в большинстве версий этих компиляторов по умолчанию используется самый старый стандарт языка и новые конструкции не распознаются.
- `-Wall`: помимо предупреждений, активных по умолчанию, включить большое количество дополнительных полезных предупреждений.
- `-pedantic-errors`: отключить расширения языка, не входящие в стандарт и включить все предупреждения, предписываемые стандартом.

Таковы опции для использования с компилятором при вызове его из командной строки вручную.

При использовании проектов Qt Creator следует внести изменения в файл проекта таким образом, чтобы он остался переносимым на другие ОС. Автор настоятельно рекомендует сделать это сразу после создания нового проекта, поскольку отсутствие этих настроек приведёт к тому, что не соответствующие стандарту программы будут транслироваться, а соответствующие — нет. Следует добавить следующие строки:

```
CONFIG += c++1z strict_c++
gcc:QMAKE_CXXFLAGS += -pedantic-errors
```

Их можно объединить с имеющимися, у которых те же левые части и форма задания (`=` замена значения, `+=` дополнение значения или `-=` исключение из значения).

Глава 5

Структурное программирование на языке C++

5.1. Использование функций

Приведём пример программы, использующей функции для однократной записи повторяющихся фрагментов алгоритма.

Необходимо вычислить, при каких давлениях находятся водород и воздух заданного веса в заданном объёме при двух заданных температурах и разницу этих давлений. Считать поведение обоих газов идеальным. Температуру вводить в градусах Фаренгейта, остальные величины — в единицах СИ.

Из курса физики известно, что

$$\begin{gathered} p = \frac{mRT}{MV} \\ t_K = (t_F + 459.67) \times \frac{5}{9}, \end{gathered} \tag{5.1}$$

где p — давление (в Паскалях, нет речь не о языке программирования), m — масса (в килограммах), T — температура (в градусах Кельвина), M — молярная масса (в килограммах на моль), V — объём (в кубометрах). R — универсальная газовая постоянная, равная $8.3144621 \frac{\text{Дж}}{\text{моль} \times K}$, $M_{H_2} = 0.002$ кг/моль, $M_{\text{воздух}} = 0.029$ кг/моль. Перевод из градусов Фаренгейта t_F в градусы Кельвина t_K осуществляется по второй формуле.

```
1  // Вычисление изменения давления водорода и
2  // воздуха при изменении температуры.
3
4  #include <iostream>
5
6  void line()
7  {
8      std::cout << "--------------------------\n";
9  }
10
11 double input_f()
12 {
13     double f;
14     std::cin >> f;
15     return (f+459.67)*5/9;
16 }
17
18 double pressure(double m,double M,double T,double V)
19 {
20     return m*8.3144621*T/M/V;
21 }
22
23 void pressure_diff(double m,double M,double T1,double T2,double V)
24 {
25     double p1,p2;
26     p1 = pressure(m,M,T1,V);
```

```
27      p2 = pressure(m,M,T2,V);
28      std::cout << "p1 = " << p1 << " Pa\n"
29                   "p2 = " << p2 << " Pa\n"
30                   "p difference = " << p2-p1 << " Pa\n";
31  }
32
33  int main()
34  {
35      line();
36      double m,T1,T2,V;
37      std::cout << "Mass (kg): ";
38      std::cin >> m;
39      std::cout << "T1 (degrees F): ";
40      T1 = input_f();
41      std::cout << "T2 (degrees F): ";
42      T2 = input_f();
43      std::cout << "Volume: (m^3): ";
44      std::cin >> V;
45      line();
46      std::cout << "Hydrogen:\n";
47      pressure_diff(m,0.002,T1,T2,V);
48      line();
49      std::cout << "Air:\n";
50      pressure_diff(m,0.029,T1,T2,V);
51      line();
52      return 0;
53  }
```

Файл исходного текста обычно начинается с комментария, описывающего его содержимое. В данном случае приведено описание сути программы.

Для оформления вводимых пользователем данных и выводимых результатов программа выводит горизонтальные линии из символов `--`. Поскольку эта операция необходима в программе несколько раз, она вынесена в отдельную функцию `line`. Это, например, позволяет изменением строкового литерала в ней влиять на вид линии, многократно выводимой программой.

У этой функции нет осмысленного возвращаемого значения — вся её полезная работа заключена в её побочном эффекте, который сам является побочным эффектом вычисления операции `<<`. В таком случае в качестве возвращаемого значения функции используется тип `void`.

Тип `void` — фундаментальный тип с пустым множеством значений. Описать объект, хранящий значения такого типа невозможно, однако в процессе вычислений могут появляться «значения» такого типа. Помимо явного приведения типа этого значения к собственному, единственная допустимая операция над такими значениями — их отбрасывание. Значение любого другого типа может быть приведено к типу `void` операций `static_cast`, чтобы явно указать, что оно должно быть отброшено. Наиболее частым применением этого типа является указание его в качестве возвращаемого значения функций, которые не должны возвращать никакого осмысленного значения, как в этом примере. В функциях, возвращающих значение типа `void`, применяется оператор `return` в форме без выражения для их досрочного завершения, например:

```
1  void f()
2  {
3      return;
4      // Следующий оператор не выполняется никогда:
5      std::cout << "Can't see this!\n";
6  }
```

Такие функции могут также завершаться по окончании выполнения всех операторов в их теле.

> В некоторых других языках программирования подпрограммы делятся на *функции*, возвращающие некое значение, и *процедуры*, которые ничего не возвращают. В языке C++ такого деления нет — все подпрограммы считаются функциями и возвращают значение определённого типа, даже если этот тип — `void`.
>
> Функция `main` — единственная функция, возвращающая не `void`, которой разрешено

> завершаться по окончании своего тела без явного оператора `return`. В таком случае считается, что функция вернула значение 0.

Вернёмся к разбору рассматриваемой программы. Функция `input_f` осуществляет ввод значения температуры в градусах Фаренгейта и возвращает введённое значение после его преобразования в градусы Кельвина. Функция `pressure` осуществляет расчёт давления по заданным параметрам. Функция `pressure_diff` осуществляет подсчёт и вывод значений давления и их разницы. В ней используется автоматическое склеивание подряд идущих токенов-строковых литералов препроцессором для удобной записи, соответствующей выводимым на экран данным без лишних операций `<<`. Функция `main` осуществляет ввод необходимых значений, вывод заголовков и вызов функций программы, осуществляющих вычисления.

Приведём пример работы данной программы:

```
---------------------------
Mass (kg): 0.002↵
T1 (degrees F): 40.1↵
T2 (degrees F): 60.3↵
Volume: (m^3): 1.5↵
---------------------------
Hydrogen:
p1 = 1539.006935 Pa
p2 = 1601.211429 Pa
p difference = 62.204494 Pa
---------------------------
Air:
p1 = 106.138409 Pa
p2 = 110.428374 Pa
p difference = 4.289965 Pa
---------------------------
```

Автор предлагает уделять хотя бы минимальное внимание организации интерфейса программы с текстовым интерфейсом, поскольку даже при известном назначении пользоваться программой, которая встречает тебя чёрным экраном и ждёт неизвестно чего весьма затруднительно.

Эта программа также не содержит проверок на корректность и успешность ввода данных, что является серьёзным недостатком. Кроме того лишь самые простые алгоритмы состоят из строго последовательного набора действий, который никак не зависит от входных данных. Перейдём к рассмотрению операторов, позволяющих изменять естественный последовательный порядок выполнения программы.

5.2. Ветвления

5.2.1. Условный оператор if

Логические вычисления позволяют в итоге получить значение, на основании которого может быть выбран тот или иной путь выполнения действий, задаваемых программой. Конструкцией, позволяющей выполнить те или иные операторы в зависимости от заданного логического значения является ***условный оператор `if`***. Он имеет две формы синтаксиса — полную и короткую:

`if (` expression `)` statement
`if (` expression `)` statement-1 `else` statement-2

Пара круглых скобок вокруг выражения является элементом синтаксиса оператора, не имеет отношения к самому выражению и опущена быть не может. Само выражение называют ***управляющим (controlling)***, его значение должно быть преобразуемо к булевскому типу. Короткая форма оператора предписывает: «если значение выражения истинно, выполнить оператор». В данном случае элементом синтаксиса одного оператора являются другие произвольные операторы. Длинная форма предписывает: «если значение выражения истинно, выполнить оператор-1, иначе выполнить оператор-2». Таким образом, ***оператор `if` реализует одну из конструкций структурного программирования — ветвление***. Вложенные в него операторы, контролируемые заданным в нём условием, неформально называют ***положительной и отрицательной ветвями***. Приведём пример его использования:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x;
6      std::cout << "Input an integer: ";
7      if(!(std::cin >> x)){
8          std::cerr << "Input error.\n";
9          return 1;
10     }
11     if(x>0)
12         std::cout << "x is positive.\n";
13     else if(x<0)
14         std::cout << "x is negative.\n";
15     else
16         std::cout << "x is zero.\n";
17     return 0;
18 }
```

Данная программа запрашивает у пользователя целое число и выводит его знак. В отличие от предыдущих программ, данная осуществляет проверку на успешность ввода значения. Под успешностью подразумевается то, что в самой среде выполнения не возникло ошибок ввода-вывода, и хотя бы один непробельный символ был считан и интерпретирован как значение, записанное в указанный объект, прежде чем встретились пробельные или другие символы, которые не могут принадлежать корректной записи значения в требуемом формате.

Данная программа пользуется тем фактом, что тип объекта стандартного потока ввода допускает преобразование к типу `bool`. Полученное значение истинно, если все операции ввода с момента запуска программы были успешны, и ложно в противном случае.

Оператор `if`, начинающийся на строке 7 выполняется следующим образом:

1. Вычисляется значение выражения в операторе `if`, содержащее вычисление операции, осуществляющей ввод данных их стандартного потока ввода. После её вычисления, если ввод был успешным, в качестве побочного эффекта считанное и интерпретированное значение записано в объект `x`. Значение этого выражения — отрицание значения объекта `std::cin`, которое преобразуется вначале к типу `bool`.
2. Если ввод не удачен, значение контролирующего выражения истинно. В таком случае выполняется заданный как часть оператора `if` оператор, следующий за закрывающей круглой скобкой в его синтаксисе. В данном случае это составной оператор, начинающийся с `{` телом которого являются строки 8-9 до соответствующей парной `}`. Использование составного оператора позволяет делать ветви из произвольного числа операторов, хотя синтаксис оператора ветвления и предполагает ровно один. ***Составной оператор может содержать несколько операторов, но сам синтаксически считается одним оператором. Это позволяет контролировать одним условным оператором несколько действий.*** Этот блок содержит два оператора, которые выполняются как обычно последовательно. Побочным эффектом выполнения первого является вывод сообщения об ошибке, а второй завершает выполнение функции `main` (и вместе с ней всей программы). Для вывода левым операндом операции `<<` указан объект `std::cerr` — ***стандартный поток ошибок (standard error stream)***. Он аналогичен по свойствам стандартному потоку вывода. Исторически, программа имеет два стандартных потока для записи результатов — вывода для штатных результатов и запросов и ошибок для диагностических сообщений. Обычно оба они соответствуют выводу на экран, но раздельное использование их в программах позволяет удобнее автоматизировать работу вашей программы внешними средствами.
3. Если ввод удачен, то контролирующее условие ложно, и оператор, входящий в оператор `if`, пропускается. В этом случае объект `x` содержит успешно введённое значение, которое далее можно использовать.

Этот пример также показывает, что в функции может быть несколько операторов `return`.

После успешного получения значения от пользователя, его знак определяется двумя операторами `if`, второй из которых является вторым оператором в полной форме первого. Остальные операторы, управляемые условиями, являются операторами-выражениями, используемыми для вывода соответствующих сообщений. Реализуемый алгоритм таков:

1. Если x>0, выполнить пункт 1.1, иначе пункт 1.2.

1.1. Вывести «x — положительное».

1.2. Если x<0, то выполнить пункт 1.2.1, иначе пункт 1.2.2.

1.2.1. Вывести «x — отрицательное».

1.2.2. Вывести «x — ноль».

Как видно из нумерации пунктов, вся эта конструкция является всего одним оператором `if`, в который входит множество других операторов.

Важность проверки пользовательского ввода на успешность в том, что если он неудачен, записи в объекты, содержащие начальные неопределённые значения не происходит, и программа далее вызывает неопределённое поведение при чтении значения этого объекта в последующих вычислениях. Отсутствие проверки ввода на успешность и корректность является одним из основных векторов атак на программные продукты, так как злоумышленник может использовать неопределённое или попросту некорректное поведение программы в своих целях. Автор просит уделять этому вопросу особое внимание даже в простых учебных программах, поскольку это вырабатывает важную привычку не доверять любым внешним данным без надлежащей проверки, что является первым шагом в разработке стабильных и безопасных программ.

Исходя из синтаксиса языка C++, при наличии нескольких вложенных операторов `if`, ключевое слово `else` и соответствующий ей оператор-2 соответствуют ближайшему из них, у которого соответствующей ветви ещё нет. В данной программе перед каждым ключевым словом `else` содержится только один оператор `if` без ещё заданной отрицательной ветви, так что неоднозначности не возникает. Соответствующая структура программы показана визуально в её тексте за счёт использования начальных пробелов в строке. Рассмотрим пример, где имеется подобная неоднозначность, поскольку в этой ситуации часто допускаются ошибки.

```
1  // Фрагмент
2  bool a,b;
3  // ...
4  if(a)
5      if(b)
6          std::cout << "a&&b\n";
7  else
8      std::cout << "!a\n";
```

Исходя из оформления кода можно предположить, что программист подразумевал принадлежность отрицательной ветви, содержащейся в двух последних строках, к первому оператору `if`, т.е. при условии, когда `a` ложно. Однако отрицательная ветвь относится к ближайшему оператору `if`, у которого она ещё не задана, т.е. ко второму. Чтобы не вводить читателя (например, самого себя) в заблуждение, программисту следовало оформить свой код следующим образом:

```
1  // Фрагмент
2  bool a,b;
3  // ...
4  if(a)
5      if(b)
6          std::cout << "a&&b\n";
7      else
8          std::cout << "!a";
```

Этот пример эквивалентен предыдущему, т.к. отличается только незначащими пробелами. Таким образом, последняя ветвь выполняется, когда `a&&!b`, т.е. совсем не тогда, когда это требовалось. Чтобы всё происходило в соответствии с выводимыми сообщениями, необходимо добавить кажущийся на первый взгляд лишним составной оператор из одного оператора:

```
1  // Фрагмент
2  bool a,b;
3  // ...
4  if(a){
5      if(b)
6          std::cout << "a&&b\n";
7  }else
8      std::cout << "!a";
```

Наличие составного оператора явно задаёт конец второго оператора `if`, поэтому последняя ветвь относится к первому, как и предполагалось.

> Условия вида `x!=0` и `x==0` часто записываются в виде `x` и `!x` соответственно, особенно при их использовании в качестве контролирующих условий. Вопрос, какая форма нагляднее, остаётся открытым, однако оба варианта широко распространены и их эквивалентность нужно иметь в виду.

5.2.2. Условная операция ?:

Условная операция ?: иногда называется тернарной операцией, поскольку является единственной операцией в языке C++ с фиксированным числом операндов, равным трём:

conditional-expression**:**
    logical-or-expression
    logical-or-expression **?** expression **:** assignment-expression

Эта операция, как и логические, является исключением из абстракции приоритета операций. Вначале вычисляется выражение-1. Если его значение истинно, то вычисляется выражение-2, иначе выражение-3, и вычисленное выражение становится результатом операции. Таким образом, либо выражение-2, либо выражение-3 остаётся не вычисленным. Операция часто применяется для замены конструкции

```
if(condition)
    object = expression1;
else
    object = expression2;
```

на более короткий вариант

```
object = condition?expression1:expression2;
```

Тип возвращаемого значения этой операции определяется обычно по правилам обычных арифметических преобразований, оба операнда также могут иметь тип `void`.

Приведём пример её использования:

```
1  int bound(int lower,int value,int upper)
2  {
3      return value<lower?lower:
4             value>upper?upper:
5             value;
6  }
```

Функция `bound` осуществляет проверку вхождения значения `value` в интервал $[lower; upper]$ и, если это не так, возвращает ближайшую границу интервала.

Если второй и третий операнд имеют один тип и леводопустимы, значение операции также леводопустимо.

```
1  int a,b;
2  // ...
3
4  // Занулить объект a или b, смотря в каком из них
5  // меньшее значение.
6  (a<b?a:b) = 0;
```

5.3. Операторы цикла

Для завершения рассмотрения конструкций, составляющих структурное программирование, осталось рассмотреть ***операторы цикла (iteration statements)***. Все они позволяют выполнить один и тот же оператор, называемый ***телом цикла (loop body)*** несколько раз в соответствии с заданными условиями. Каждое такое выполнение называют ***итерацией (iteration)***.

5.3.1. Оператор цикла с предусловием while

Оператор цикла с предусловием **`while`** даёт возможность выполнять один и тот же оператор многократно, пока заданное в нём условие истинно. Его синтаксис:

`while (` expression `)` statement

Выражение и оператор, входящие в оператор `while` называются так же, как и в условном операторе `if` и выполняют схожие функции. Оператор `while` предписывает: «выполнять оператор повторно, пока условие истинно»: сначала проверяется условие. Если оно ложно, тело цикла не выполняется и выполнение оператора заканчивается. Если истинно — выполняется тело цикла, после чего условие проверяется заново. Если оно истинно снова, то тело цикл выполняется ещё раз, и т.д. Таким образом, цикл с предусловием выполняет своё тело любое число раз, включая 0. Приведём пример:

```
1  // Фрагмент
2  double x;
3  while(std::cout >> x)
4      std::cout << x*x << '\n';
```

Данный фрагмент производит ввод (без какого-либо приглашения) вещественных чисел и, пока это удаётся сделать успешно, выводит их квадраты. При ошибке ввода цикл завершается.

Для завершения работы программ построенных по схеме «работать, пока ввод успешен» помимо ввода некорректной информации может использоваться явное указание пользователем конца файла, которым является поток стандартного ввода. Для этого необходимо нажать на клавиатуре комбинацию клавиш `Control-Z` (в ОС Windows) или `Control-D` (большинство других ОС). После этого может понадобится нажать клавишу `Enter` из-за построчной буферизации стандартного потока ввода.

Типичной проблемой в ситуациях, когда приглашение к вводу желательно, является необходимость дублирования некоторой части кода, например:

```
1  // Фрагмент
2  double x;
3  std::cout << "Input a number: ";
4  while(std::cin >> x){
5      std::cout << x*x << "\n"
6                   "Input a number: ";
7  }
```

В данном случае расположенный в теле цикла вывод приглашения к вводу очередного числа происходит после вывода каждого квадрата, т.е. перед всеми приглашениями, кроме первого, которое приходится дублировать перед циклом. Довольно хитрым способом избежать этого является использование операции «запятая».

expression:
    assignment-expression
    expression `,` assignment-expression

Операция запятая (comma) **`,`** является инфиксной бинарной операцией, допускающей любые типы операндов (включая `void`). Она также гарантирует порядок вычисления своих операндов: сначала левый, потом правый. Левый операнд вычисляется как и в операторе-выражении с отбрасыванием его значения. После этого вычисляется значение правого аргумента, которое становится результатом операции с сохранением категории значения. Таким образом, операция запятая может использоваться в некоторых случаях для группировки нескольких независимых побочных эффектов в одно выражение. Отметим, что запятая, используемая для разделения аргументов в операции вызова функции, как и другие её использования вне выражений, данной операцией не являются. Чтобы использовать её в аргументе функции необходимы дополнительные скобки:

```
1  int f(int x,int y,int z)
2  {
3      return 0;
4  }
5
6  int main()
```

```
{
    int a;
    return f(1,(a=1,a+1),3)
}
```

В данном примере значение второго аргумента в вызове функции `f` равно 2: при вычислении его значения сначала 1 записывается в `a`, после чего результат операции запятая вычисляется как `a+1`, т.е. 2.

Применительно к рассматриваемой проблеме, операция запятая может быть использована, чтобы объединить приглашения ко вводу с самим вводом в одно выражение, которое может быть использовано в качестве контролирующего в операторе цикла:

```
// Фрагмент
double x;
while(std::cout << "Input a number: ",
      std::cin >> x)
    std::cout << x*x << '\n';
```

Эта операция завершает «пирамиду» формального синтаксиса основных выражений языка C++, однако мы рассмотрели ещё далеко не все его операции, что было показано многоточиями в приведённых правилах. Мы будем расширять эти конструкции по мере изучения новых возможностей языка.

5.3.2. Оператор цикла с постусловием do

Оператор цикла с постусловием do работает аналогично оператору цикла с предусловием `while`, только проверка условия, от которого зависит продолжение выполнения цикла, выполняется не *перед* каждой итерацией, а *после*. Это также отражено в его синтаксической форме: условие пишется после тела цикла:

`do` statement `while (` expression `) ;`

Таким образом, в отличие от цикла с предусловием `while`, тело оператора `do` выполняется минимум один раз. Пример:

```
#include <iostream>

int main()
{
    double sum,term,x;
    sum = x = 0;
    do{
        ++x;
        term = 6./(x*x);
        sum += term;
    }while(sum/term<1e10);
    std::cout << "pi^2 = " << sum << " (" << it << " iterations used)\n";
    return 0;
}
```

Данная программа вычисляет приближённое значение π^2, пользуясь тем, что

$$\sum_{x=1}^{\infty} \frac{6}{x^2} = \pi^2. \tag{5.2}$$

Программа вычисляет сумму конечного числа первых членов ряда, останавливая вычисления, когда очередной вычисленный член становится на 10 порядков меньше накопленной суммы. Результат выполнения программы:

```
pi^2 = 9.869527 (77970 iterations used)
```

Ряд сходится медленно, поэтому выполнено достаточно большое число итераций, а в результате только 4 знака верные.

Обычно в программах циклы с постусловием используются реже циклов с предусловием, поскольку обычно все итерации требуют проверки необходимости их выполнения. Конструкции циклов с пред- и постусловием в целом равносильны, поскольку каждый из них может быть преобразован в другой, например:

```
do
    statement;
while(condition);
```

равносильно

```
statement;
while(condition)
    statement;
```

В данном примере происходит дублирование кода тела цикла. В реальных задачах не всегда удаётся полностью его избежать в той или иной форме цикла, допускающих проверку условия продолжения (или, эквивалентно, выхода) из него строго до или после тела. Минимизация этого дублирования и определяет выбор соответствующей формы.

5.3.3. Оператор цикла for

Оператор цикла `for` является краткой формой записи циклов с предусловием, явно выделяющим этапы установления начального состояния, от которого зависит условие продолжения его выполнения, и изменения этого состояния после каждой итерации цикла. Синтаксис оператора `for`:

`for (` expression-1_{opt} `;` expression-2_{opt} `;` expression-3_{opt} `)` `statement`

Выражение-1 вычисляется перед входом в цикл, его значение отбрасывается. Перед каждой итерацией цикла вычисляется выражение-2, которое определяет, следует ли продолжить выполнение цикла, как и в операторе `while` — оно является контролирующим. Выражение-3 вычисляется после каждой итерации цикла до проверки условия его продолжения, результат отбрасывается. Таким образом цикл `for` во многом эквивалентен конструкции

```
 expression-1;
while( expression-2){
      statement
      expression-3;
}
```

Рассмотрим пример, печатающий все целые числа по порядку от 1 до 100:

```
1  // Фрагмент
2  int i;
3  i = 1;
4  while(i<=100){
5      std::cout << i << '\n';
6      ++i;
7  }
```

С помощью цикла `for` этот пример может быть записан в виде

```
1  // Фрагмент
2  int i;
3  for(i=1;i<=100;++i)
4      std::cout << i << '\n';
```

Таким образом, вся логика связанная с объектом, исполняющим роль счётчика, сосредоточена в заголовке цикла: задание его начального значения, изменение его значения между итерациями и проверка на конец цикла.

> В таком виде цикл `for` является «циклом с параметром», который в некоторых других языках программирования обозначается тем же ключевым словом. Подчеркнём, что в языке C++ это не так — цикл `for` является лишь краткой записью цикла с предусловием и дополнительных выражений и допускает гораздо более сложные конструкции. В частности, объект, выступающий в роли счётчика никаким особым свойством не обладает и может, например, быть модифицирован в теле цикла, чтобы обеспечить «возврат» к предыдущим итерациям или досрочное завершение цикла. Данная роль объекту приписывается программистом, число и назначение объектов, используемых в синтаксисе цикла `for` ничем не ограничены.

Все три выражения (но не точки с запятой их разделяющие) являются опциональными элементами синтаксиса независимо друг от друга, что показано в правиле синтаксиса нижним индексом opt. При отсутствии выражения-1 или выражения-3, ничего не вычисляется до входа в цикл или после каждой итерации соответственно. При отсутствии выражения-2 подразумевается, что оно имеет вид `true`, т.е. условие продолжения цикла всегда истинно и изменить его нельзя — цикл ***вечен***. По-настоящему вечный цикл, разумеется, приводит к зависанию программы, но наличие заведомо истинного условия продолжения цикла не обязательно означает, что он вечный, поскольку выход из цикла может быть осуществлён из его тела, например, оператором `return`, который осуществляет возврат управления из функции, данный цикл содержащей.

Приведём дополнительные примеры использования цикла `for`:

```
// Фрагмент
int i,j;

// от 100 до 2 с шагом в -2
for(i=100;i>=2;i=i-2)
    std::cout << i << '\n';

// Степени числа 2 до 1024
for(i=1;i<=1024;i=i*2)
    std::cout << i << '\n';

// Два счётчика сразу:
// i = 0,1,2,...,9
// j = 1,3,9,...,19683
for(i=0,j=1;i<10;i=i+1,j=j*3)
    std::cout << i << " - " <, j << '\n';

// Вводить числа, пока ввод успешен,
// вывести число введённых чисел по окончании.
for(i=0;std::cin >> j;++i)
    std::cout << j << '\n';
std::cout << "Total inputs: " << i <, '\n';
```

При написании циклов следует обращать особое внимание на начальные и конечные значения счётчиков, а также на используемые операции сравнения, особенно их строгие и не строгие варианты, например:

```
int i;

// [0;9]:
for(i=0;i<=9;++i)
    std::cout << i << '\n';

// [0;10), что то же самое:
for(i=0;i<10;++i)
    std::cout << i << '\n';

// [0;10] - другой интервал,
// в который также входит число 10:
for(i=0;i<10;++i)
    std::cout << i << '\n';

// От 10 до 0:
for(i=10;i>=0;--i)
    std::cout << i << '\n';

// Особое внимание при счёте вниз и
// беззнаковых типах - следующий цикл
// вечный, потому что unsigned всегда >=0 !
unsigned j;
for(j=10;j>=0;--j)
    std::cout << j << '\n';
```

В последнем примере, когда `j==0`, вычитание из него единицы даёт максимальное значение типа `unsigned int` по правилам выполнения операций над беззнаковыми типами.

5.4. Изменение порядка выполнения команд в машинном коде

В архитектуре x86, как нам уже известно, регистр процессора `EIP` содержит адрес инструкции, следующей за выполняемой в данный момент. За счёт его автоматического изменения после прочтения каждой инструкции обеспечивается их последовательное выполнение, т.е. конструкция «последовательность» структурной парадигмы программирования. Однако машинный код не соответствует парадигме структурного программирования — имеющиеся в нём команды и возможности соответствуют гораздо более низкоуровнёвым конструкциям, которые тем не менее могут использоваться и для реализации конструкций структурного программирования.

Регистр `EIP` является особым, и его значение не может быть изменено обычной арифметической инструкцией или инструкцией переноса, для его изменения используются отдельные команды. Инструкция `jmp` (JuMP) осуществляет *безусловный* переход по адресу, указанному в качестве его операнда. Также имеется семейство инструкций, осуществляющих *условный* переход — переход либо осуществляется, либо нет (т.е. продолжается последовательное выполнение), в зависимости от одного из битов регистра флагов `EFLAGS`. Регистр `EFLAGS` — специальный 32-битный регистр, отдельные биты которого меняются многими инструкциями в зависимости от результата их выполнения. Эти биты имеют собственные имена, рассмотрим наиболее часто используемые из них. Перед этим вспомним, что x86 использует представление знаковых чисел в виде дополнительного кода, поэтому для сложения и вычитания знаковых и беззнаковых чисел используются одни и те же инструкции.

- `ZF` (Zero Flag) — выставляется в 1 тогда и только тогда, когда результат операции равен нулю.
- `CF` (Carry Flag) — выставляется в 1 тогда и только тогда, когда при сложении произошёл перенос из старшего разряда или при вычитании потребовался забор в старшем разряде. Для операций с беззнаковыми величинами это означает, что произошло приведение результата, не попадающего в интервал значений 32-битного беззнакового типа по модулю 2^{32}.
- `SF` (Sign Flag) — выставляется в старший бит результата. Поскольку старший бит в дополнительном коде является битом знака, то 1 в этом бите означает то, что получен отрицательный результат в операции со знаковым 32-битным типом.
- `OF` (Overflow Flag) — выставляется в 1, если при сложении двух чисел с одинаковым старшим битом старший бит результата отличается от них. Вычитание в данном случае эквивалентно сложению с противоположным значением. Для знаковой арифметики это признак того, что результат операции вышел за границы допустимых значений знакового типа.

Указанные правила выставления флагов соответствует поведению инструкций сложения и вычитания, другие операции могут иметь другое поведение.

Отметим, что простое игнорирование бита переноса, по сути являющегося 33-им битом результата для операций сложения и вычитания, автоматически даёт требуемое по стандарту языка C++ поведение для беззнаковой арифметики.

Все операции сравнения и отношения для знаковых и беззнаковых чисел можно выразить через проверку определённых битов регистра флагов после операции вычитания их операндов. Поскольку такая проверка осуществляется часто, существует инструкция `cmp` (CoMPare), которая осуществляет выставление битов регистра флагов так, словно было выполнено вычитание, но сама разность нигде не сохраняется. Например, проверка на равенство эквивалентна проверке `ZF==1` для любой знаковости, проверке «меньше или равно» для беззнаковых величин соответствует `ZF==1||CF==1`, а для знаковых — `ZF==1||SF!=OF`. Последнее соответствие можно получить следующим образом:

- Если операнды равны, то при вычитании будет получен ноль, и ZF будет равным единице. Осталось рассмотреть строгое отношение «меньше».
- Если $a < b$, то тогда и только тогда $a - b < 0$. Возможны случаи, когда результат вычитания не входит в диапазон значения знакового типа. Предположим, что он входит (`OF==0`), тогда результат корректен и отрицателен: `SF==1`. Если же произошло знаковое переполнение (`OF==1`), то в результате получено неправильное значение с

противоположным знаком (по алгоритму установки бита `OF`), т.е. положительным: `SF==0`.

- Объединив все случаи и упростив выражение, получим требуемое `ZF==1||SF!=OF`.

Таким образом, бит `OF`, формально выставляемый в 1 только в ситуациях, объявленных по стандарту языка C++ определяемым реализацией поведением, тем не менее, используется в реализации конструкций языка, поведение которых строго определено.

Рассмотрим фрагмент:

```
1 unsigned a,b,c;
2 // ...
3 if(a==2)
4     b = -b;
5 else
6     b -= 5;
7 c = 4;
```

Предположим, что объекты `a`, `b` и `c` расположены по адресам `0x3000`, `0x3004` и `0x3008` соответственно. Самым прямым (и не обязательно оптимальным) вариантом трансляции вышеприведённого фрагмента может быть

```
1000: 83 3d 00 30 00 00 02          cmp dword [0x3000],0x2
1007: 75 08                         jne 0x1011
1009: f7 1d 04 30 00 00             neg dword [0x3004]
100f: eb 07                         jmp 0x1018
1011: 83 2d 04 30 00 00 05          sub dword [0x3004],0x5
1018: c7 05 08 30 00 00 2a          mov dword [0x3008],0x4
```

Первая инструкция выставляет биты регистра флагов как при вычитании значения 2 из 4-байтного значения по адресу `0x3000`. Поскольку в этой операции не участвуют обычные регистры, размер операнда в памяти задаётся словом `dword` явно. (Архитектура x86 изначально была 16-битной и понятие машинное ***слово (word)*** закрепилось за этой величиной. После расширения в процессоре 80386 большинства регистров до 32 бит, эта величина стала называться ***двойным словом (double word)*** или сокращённо `dword`.)

Вторая инструкция `jne` (Jump if Not Equal) является инструкцией условного перехода, которая осуществляет переход по адресу `0x1011`, если `ZF==0`, т.е. в предшествующей инструкции сравнения операнды отличались. Видно, что адрес `0x1011` задан в машинном коде смещением `+0x08` относительно значения регистра `EIP`, равного `0x1009`, которое тот имеет в момент начала исполнения этой инструкции.

Третья инструкция `neg` (NEGate) осуществляет смену знака значения, хранящегося в объекте `b`, а четвёртая осуществляет безусловный переход, чтобы обойти отрицательную ветвь кода условного оператора, содержащую инструкцию вычитания `sub` (SUBtract). В конце отрицательной ветви ничего пропускать не надо, естественный последовательный порядок управления переходит к выполнению последней инструкции, реализующей оператор, следующий за условным в первоначальном тексте на языке C++.

Как можно видеть, за счёт условных переходов можно строить произвольные конструкции по передаче управления. Именно их чрезмерное усложнение и привело к разработке парадигмы структурного программирования. При программировании на языке C++ следует придерживаться её по возможности, а уже компилятор подберёт наиболее оптимальное представление в машинном коде. Хотя оно может быть значительно более сложным, это представление для программиста на языке C++ не является основным, с которым он работает. Таким образом мы получаем совмещение читаемого кода на машинонезависимом языке и оптимальный код, построенный оптимизирующим компилятором.

5.5. Операторы перехода break, continue, goto

Парадигма структурного программирования возникла из хаоса потока управления, творившегося в большинстве программ, когда в условиях ограниченности ресурсов первых ЭВМ программисты на каждом шагу прибегали ко всевозможным хитростям. Ресурсы машин росли, но привычка сохранялась, что пагубно сказывалось на читаемости всё разраставшихся программ. Не прибегая к догматизму, следует чётко понимать, какую реальную проблему решали создатели этой парадигмы. Имея это в виду, рассмотрим ***операторы перехода (jump statements)***, во многом нарушающие принципы структурного программирования,

оставаясь в одноимённом разделе.

К операторам перехода относят оператор `return`, который был рассмотрен ранее.

5.5.1. Оператор break в телах циклов

Оператор break может использоваться внутри тела любого оператора цикла. Он производит досрочный выход из цикла, т.е. передачу управления оператору, следующему за оператором цикла, в теле которого он используется. Приведём пример его использования.

```
#include <iostream>

int main()
{
    // Запросим у пользователя, сколько
    // он будет вводить чисел.
    unsigned n;
    std::cout << "How many numbers? ";
    if(!(std::cin >> n)){
        std::cerr << "Input failed or invalid count.\n";
        return 1;
    }
    // Посчитаем их сумму.
    int sum;
    sum = 0;
    while(n){
        std::cout << "Input a number: ";
        int x;
        if(!(std::cin >> x))
            // Пользователь не ввёл очередное число, выходим из цикла досрочно.
            break;
        sum += x;
        --n;
    }
    // Выведем сумму и сколько ещё чисел пользователь обещал ввести, но не ввёл.
    std::cout << "Sum of numbers: " << sum << "\n"
                 "You failed to input " << n << " numbers.\n";
    return 0;
}
```

Действия, производимого оператором `break` часто можно добиться искусственным изменением значения объектов, от которых зависит контролирующее условие цикла, но это не всегда возможно, и использование оператора `break` позволяет наглядно передать синтаксисом программы желание программиста без неестественных изменений. Вот ***плохой*** пример:

```
// !!! Умышленно плохой пример !!!

// Фрагмент
int i = 0;
while(i<10){
    // ...
    if(condition){
        // Условие истинно,
        // выполняем требуемые действия.
    }else{
        // Условие ложно, цикл нужно
        // завершить заранее.
        // Присвоим i значение,
        // которое к этому приведёт.
        i = 10;
    }
}
// А что делать, если значение i, на котором
// цикл прерван досрочно потребуется далее?
```

Для сравнения приведём правильный вариант с использованием оператора `break`:

```
// Фрагмент
int i = 0;
while(i<10){
    // ...
    if(!condition)
        // Условие ложно, цикл нужно
        // завершить заранее.
        break;
    // Условие истинно,
    // выполняем требуемые действия.
    // Не нужно перестраивать структуру программы, чтобы действия,
    // выполняемые после проверки дополнительного условия продолжения
    // цикла шли в блоке, зависящем от него.
    // ...
}
// Значение i, на котором завершился цикл (досрочно или нет), сохранено.
```

Теперь мы можем привести пример решения проблемы с циклом, в котором беззнаковый счётчик считает вниз до нуля:

```
unsigned i;
for(i=10;;--i){
    std::cout << i << '\n';
    // Условие завершения цикла вынесено из контролирующего в само тело
    // цикла и проверяется до декрмента счётчика.
    if(!i)
        break;
}
```

5.5.2. Оператор continue

Оператор `continue` досрочно завершает итерацию цикла, т.е. переходит в самый конец тела цикла. Пример его использования:

```
#include <iostream>

int main()
{
    int sum,x;
    sum = 0;
    std::cout << "Input positive numbers:\n";
    while(std::cin >> x){
        if(x<=0){
            std::cerr << x << " is negative, try again: ";
            if(!(std::cin >> x)||x<=0){
                std::cerr << "Failed again, try with next index.\n";
                continue;
            }
        }
        sum += x;
        std::cout << "Sum = " << sum << '\n';
    }
    return 0;
}
```

В данной программе суммируются введённые пользователем положительные числа. Если введённое число отрицательное, то даётся вторая попытка ввести допустимое значение. Если и она неудачна, то выводится соответствующее сообщение и программа продолжает ввод числа под новым номером. В данном случае оператор `continue` позволяет завершить итерацию цикла из вложенной серии условных операторов. Пример сеанса работы с этой программой:

```
Input positive numbers:
2↵
Sum = 2
3↵
Sum = 5
```

```
0↵
0 is negative, try again: 4↵
Sum = 9
-1↵
-1 is negative, try again: -5↵
Failed again, try with next position.
6↵
Sum = 15
x↵
```

Этот оператор также может может применяться для уменьшения горизонтальной ширины кода, вместо

```
// Фрагмент
while(loop_condition){
    // ...
    if(condition1){
        // ...
        if(condition2){
            // ...
            if(condition3){
              // ...
            }
        }
    }
}
```

можно записать

```
// Фрагмент
while(loop_condition){
    // ...
    if(!condition1)
        continue;
    // ...
    if(!condition2)
        continue;
    // ...
    if(!condition3)
        continue;
    // ...
}
```

5.5.3. Оператор goto

Использование оператора `goto` вызывает наибольший накал страстей среди большинства программистов. Существует множество рекомендаций о том, что его не следует применять вовсе, однако автор считает, что его использование вполне допустимо, если это *повышает* читаемость кода, что вполне возможно. Рассмотрим пример:

```
// Фрагмент
while(condition1){
    // ...
    while(condition2){
        // ...
        while(condition3){
            // ...
            if(condition4){
                // Здесь обнаружилось,
                // что нужно выйти из
                // всех трёх циклов -->
            }
            // ...
        }
        // ...
    }
```

```
17      // ...
18  }
19  // <-- т.е. вот сюда.
20  std::cout << "Loops done\n";
```

Оператор `break` сам по себе здесь не поможет, он выходит только из того одного цикла, в теле которого непосредственно находится.

> В C++, в отличие от некоторых других языков, у оператора break нет дополнительных параметров, определяющих, из скольких циклов следует выйти.

Можно, конечно, пытаться сделать что-то с несколькими операторами `break` в каждом цикле в условиях, проверяющих дополнительные флаги и/или менять значения объектов, контролирующие условия... но будет ли наглядной такая программа?

Оператор `goto` осуществляет переход к ***помеченному (labeled)*** оператору в теле функции, в которой используется. Чтобы пометить оператор, перед ним следует записать идентификатор, называемый ***меткой (label)***, отделив его двоеточием. Переход возможен только в рамках одной функции, но передача управления может осуществляться в любом направлении, вперёд или назад, внутрь или наружу блоков. Все метки в пределах одной функции должны иметь уникальные имена, задаваемые их идентификаторами. Оператор `goto` состоит из ключевого слова `goto`, идентификатора метки, к которой осуществляется переход, и точки с запятой. С помощью этого оператора вышеприведённая задача легко решается:

```
1   // Фрагмент
2   while(condition1){
3       // ...
4       while(condition2){
5           // ...
6           while(condition3){
7               // ...
8               if(condition4){
9                   // Здесь обнаружилось,
10                  // что нужно выйти из
11                  // всех трёх циклов -
12                  // поможет goto:
13                  goto end_of_loops;
14              }
15              // ...
16          }
17          // ...
18      }
19      // ...
20  }
21  end_of_loops:
22  std::cout << "Loops done\n";
23  // ...
```

Приведём осмысленный пример использования оператора `goto` в аналогичной ситуации.

```
1   #include <iostream>
2
3   int main()
4   {
5       // Ввести целое число больше единицы.
6       std::cout << "Input a number >1: ";
7       int a;
8       if(!(std::cin >> a)||a<2){
9           std::cerr << "Invalid input.\n";
10          return 0;
11      }
12      // Попробовать найти такие числа x и y,
13      // из [2;9], что введённое число делится
14      // и на (x+y-1), и на (x+1)*(y-1).
15      // Разные промежуточные ситуации отразить
16      // знаками в таблице символов, где
17      // значения x и y являются строками и
```

```
18      // столбцами соответственно.
19      std::cout << "x/y:23456789\n";
20      int x,y;
21      for(x=2;x<=9;++x){
22          std::cout << x << "  :";
23          for(y=2;y<=9;++y){
24              // Проверить требования
25              bool cond1,cond2;
26              cond1 = a%(x+y-1)==0;
27              cond2 = a%((x+1)*(y-1))==0;
28              if(cond1)
29                  if(cond2){
30                      // Выполнены оба.
31                      std::cout << '3';
32                      // Выйти из обоих циклов.
33                      goto done;
34                  }else
35                      // Только cond1.
36                      std::cout << '1';
37              else
38                  if(cond2)
39                      // Только cond2.
40                      std::cout << '2';
41                  else
42                      // Ни то, ни другое.
43                      std::cout << ' ';
44          }
45          std::cout << '\n';
46      }
47  done:
48      if(x==10&&y==10)
49          // Циклы дошли до конца, нужное
50          // значение не найдено.
51          std::cout << "Not found.\n";
52      else
53          // Нашли. Выведем конец строки
54          // для последней неполной
55          // и найденные значения.
56          std::cout << "\nFound: x=" << x << ", y=" << y << '\n';
57      return 0;
58  }
```

Пример сеанса работы:

```
Input a number: 77↵
x/y:23456789
2  :    1
3  :   1   1
4  :  1   1
5  : 1   1
6  :3
```

С помощью оператора `goto` можно представить работу операторов `break` и `continue`:

```
1   while(condition1){
2       // ...
3       if(condition2)
4           break; // эквивалентно goto break_label;
5       if(condition3)
6           continue; // эквивалентно goto continue_label;
7       // ...
8   continue_label: ;
9   }
10  break_label: ;
```

Так что позиция специалистов, запрещающих `goto`, но допускающих `break` и `continue`, автору кажется неоднозначной.

В этом примере также показано использование ***пустого оператора (null statement)***, синтаксически являющегося оператором-выражением с отсутствующим выражением, т.е. просто пунктуатором «точка с запятой». Пустой оператор не делает ничего. Здесь он оказался необходим, чтобы поставить метку на конец блока, поскольку метка не может существовать без оператора, который метит. Некоторые программисты предпочитают пустому оператору другую конструкцию, не задающую действий, но являющуюся синтаксически оператором: `{}` — пустой составной оператор. Будьте особенно аккуратны: лишняя точка с запятой не в том месте может оставить программу синтаксически корректной, но семантически далёкой от желаемой:

```
1  // Фрагмент
2  while(condition); // <-- лишняя точка с запятой!
3  {
4      // Это "тело" цикла просто составной
5      // оператор, ничем не контролируемый,
6      // он выполняется один раз
7      // независимо от condition,
8      // а тело цикла while - пустой оператор!
9      // ...
10 }
```

К счастью, большинство компиляторов выдают в таких случаях предупреждения.

5.6. Константные выражения

В некоторых конструкциях языка требуются выражения, удовлетворяющие дополнительным ограничениям, позволяющим вычислить их без выполнения программы на этапе компиляции отдельной единицы трансляции — такие выражения называют ***константными (constant)***.

Формально они имеют синтаксис условного выражения:

constant-expression**:**
    conditional-expression

Основным константным выражением (core constant expression) называют выражение указанного синтаксиса, вычисление которого не должно включать:

- Преобразование леводопустимого выражения;
- Изменение значений объектов;
- Вызов функций;
- Слишком большое число шагов, превышающих лимиты, устанавливаемые транслятором.

Целочисленное константное выражение (integral constant expression) — это выражение целочисленного типа, которое вместе с его неявным преобразованием к чисто праводопустимому значению, является основным константным выражением.

Когда в языке требуется константное целочисленное выражение конкретного типа, используют следующую терминологию: к ***преобразованным константным выражениям (converted constant expression)*** типа `T` относят константные выражения, неявное преобразование которых к типу `T` не содержит преобразований, связанных с плавающей точкой.

Пока это единственные формы константных выражений, с которыми нам предстоит встречаться.

5.7. Оператор switch

Оператор ***`switch`*** имеет синтаксис

`switch (` expression `)` statement

Выражение в скобках, как и в условном операторе, называется контролирующим выражением, а оператор — телом. Оператор `switch` осуществляет переход к метке внутри оператора,

определяемой значением контролирующего выражения после его целочисленного повышения, поэтому чаще всего тело является составным оператором. Оператор `switch` осуществляет переход к меткам специального вида, имеющим вид

`case` expression `:`

Выражения в метках `case` должны быть преобразованными константными выражениями к типу управляющего выражения после повышения. При этом в одном теле оператора `switch` не может быть двух меток `case` с одинаковым значением выражения. При выполнения оператора `switch` вычисляется значение контролирующего выражения, после чего выполнение передаётся оператору внутри его тела, помеченного меткой `case` с соответствующим значением. Если метки `case` с искомым значением в теле нет, происходит переход к метке, определяемой ключевым словом `default`. Если и её нет, то тело оператора не выполняется. Метки специального вида `case` и `default` могут применяться только в теле оператора `switch`, и при их поиске не учитываются метки из вложенных в тело других операторов `switch`. ***В теле оператора `switch` может применяться оператор `break`, который в этом случае осуществляет преждевременное завершение его выполнения, как и при его использовании в теле цикла.***

Основная функция оператора `switch` — быть более удобной формой записи последовательности условных операторов, осуществляющих последовательное тестирование одного значения. Вместо

```
if(x==1){
    // ...
}else if(x==2){
    // ...
}else if(x==3){
    // ...
}else{
    // ...
}
```

можно записать:

```
switch(x){
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
    default:
        // ...
}
```

Следует отметить, что в отличие от других языков, где присутствует аналогичная конструкция, оператор `switch` является более мощной и низкоуровнёвой конструкцией. Из-за этого в том числе типичной является ошибка, когда забывают завершать каждую «ветку», кроме последней, оператором `break` — при этом выполняется не только фрагмент кода после соответствующей метки, но и все последующие, чего в большинстве случаев, соответствующих приведённой выше цепочке условий, не требуется. С другой стороны, такая возможность имеется, если это необходимо, хотя в таком нетрадиционном случае рекомендуется явно помечать комментарием места, где управление ***проваливается (falls through)*** сквозь метку в следующий фрагмент. Также малоизвестным является приём, когда с помощью меток `case` осуществляется переход в середину не последовательно выполняемого кода непосредственно в теле оператора `switch`, а в середину, например, входящего в него цикла — метки `case` и `default` могут помечать операторы на любом уровне вложенности внутри тела, если они не входят во вложенные операторы `switch` (в этом случае они относятся к нему). Приведём пример использования этого оператора.

```
1  // Вычисление дня недели по дате
2
```

```
3  #include <iostream>
4
5  int main()
6  {
7      // Ввести дату
8      std::cout << "Input day of month, month and year: ";
9      int d,m,y;
10     if(!(std::cin >> d >> m >> y)){
11         std::cerr << "Input failed!\n";
12         return 1;
13     }
14     // Проверить дату. Последней на григорианский
15     // календарь перешла Греция в 1923-м году.
16     if(y<1924){
17         std::cerr << "Invalid year!\n";
18         return 2;
19     }
20     if(m<1||m>12){
21         std::cerr << "Invalid month!\n";
22         return 3;
23     }
24     int max_dom;
25     // Найдём максимальный номер дня в данном месяце.
26     // Это не самый простой и понятный
27     // способ, он выбран для демонстрации
28     // возможностей оператора switch.
29     // В любом месяце есть 28 дней.
30     max_dom = 28;
31     switch(m){
32         case 2:
33             // Февраль. Для високосного года добавим один день
34             // (тот год, который делится на 4,
35             //  но не тот, что делится на 100 и
36             //  при этом не делится на 400).
37             max_dom += !(y%4)&&((y%100)||!(y%400));
38             // Выйти из switch
39             break;
40         case 1:
41             // Помеченный оператор - тоже оператор, поэтому также может быть помечен,
42             // т.е. несколько меток могут метить один оператор.
43         case 3:
44         case 5:
45         case 7:
46         case 8:
47         case 10:
48         case 12:
49             // Месяцы из 31 дня.
50             // Прибавим 1 к 28 начальным...
51             ++max_dom;
52             // и "провалимся" в общий случай,
53             // чтобы там прибавили ещё 2.
54             // break нет.
55         default:
56             // Все остальные: прибавить 2 дня
57             // до 30 или 31.
58             max_dom += 2;
59     }
60     if(d<1||d>max_dom){
61         std::cerr << "Invalid day of month!\n";
62         return 4;
63     }
64     // Проверка завершена. Вычислим день недели из сравнения Зеллера.
65     // Январь-Февраль считаются месяцами 13-14 прошлого года.
66     if(m<3){
67         m += 12;
68         --y;
```

```
69      }
70      switch((d+13*(m+1)/5+y+y/4+6*(y/100)+y/400)%7){
71          case 0:
72              std::cout << "Saturday\n";
73              break;
74          case 1:
75              std::cout << "Sunday\n";
76              break;
77          case 2:
78              std::cout << "Monday\n";
79              break;
80          case 3:
81              std::cout << "Tuesday\n";
82              break;
83          case 4:
84              std::cout << "Wednesday\n";
85              break;
86          case 5:
87              std::cout << "Thursday\n";
88              break;
89          // Остался один случай, можно не указывать его значение.
90          default:
91              std::cout << "Friday\n";
92      }
93      return 0;
94  }
```

5.8. Стили оформления программ

Рассмотрев основные конструкции, комбинации которых могут задавать алгоритмы любой сложности, обсудим вопросы, связанные со стилем оформления программ. Не говоря о том, какие комбинации конструкций эквивалентны, остановимся на записи конкретной структуры программы тем или иным образом, не меняя её смысла с точки зрения синтаксиса. В таком случае в распоряжении программиста остаются два способа изменения текста:

- Изменение последовательностей незначащих пробельных символов, используемых для разделения токенов.
- Изменение написания идентификаторов, поскольку транслятору не важно, из каких конкретно символов они состоят — ему важен не символьный их состав, а их совпадение с другими их вхождениями в программу.

В среде языков программирования сформировалось множество общепринятых стилей, как специфических для конкретного языка, так и более общих (многие из них носят фамилии людей, сделавших их популярными). ***Основной целью использования стиля является улучшение зрительного восприятия программы путём систематического использования незначащих с точки зрения транслятора элементов исходного текста программы.*** Каждый программист волен выработать свой стиль, однако главным фактором, определяющим наличие того или иного стиля, является его систематическое использование — когда один и тот же элемент в программе оформлен по разному в нескольких местах, теряется основное свойство стиля нести дополнительную информацию для читателя. Из любых правил, в том числе и стилистических, разумеется, возможны исключения, если в конкретном случае изначальная задача — улучшение восприятия текста программы — будет таким образом решена лучше. Отметим также, что в большинстве крупных проектов, над которыми работает более одного человека, обычно один стиль назначается стилем проекта, и ему следует все, несмотря на личные предпочтения.

Начнём обсуждение с пробельных символов. Три наиболее часто используемых пробельных символа — пробел, горизонтальная табуляция (обычно вводится клавишей `Tab`) и перевод строки (обычно вводится клавишей `Enter`).

Использование символа табуляции следует считать проблемным в связи с тем, что его отображение зависит от настроек текстового редактора. В редакторах чистого текста, которые используются для написания исходного кода программ, символ табуляции осуществляет заполнение пробелами (или сдвиг вправо) до следующего столбца, номер которого кратен параметру «ширина табуляции». Из-за различия значений этого параметра в разных средах

(по умолчанию или изменённых их пользователями) нельзя гарантировать одинаковый вид исходного текста программы в любой среде. В связи с этим, настоятельно рекомендуется настроить используемый текстовый редактор, чтобы нажатие клавиши `Tab` вставляло необходимое число пробелов до достижения столбца, кратного ширине табуляции, вместо самого символа табуляции.

Использование символа перевода строки относительно однозначно в оформлении программ на языке C++. Перечислим стилевые аспекты, связанные с его использованием:

- В директивах препроцессора символ перевода строки значащий, вариантов использования нет.
- Чаще всего каждый оператор, включая операторы в блоках и другие, являющиеся синтаксически частями других операторов, записывается на отдельной строке. Иногда допускается запись в одну строку нескольких операторов, если они очень коротки и логически связаны.
- Длинные операторы могут разбиваться на несколько строк, чтобы ограничить ширину текста программы и исключить необходимость прокрутки его по горизонтали. Часто стараются придерживаться ширины текста в 80 символов, как стандартной ширины терминала. В текстах книг, например, в этой, приходится придерживаться ещё меньшего значения.
- Пустые строки могут использоваться (часто вместе с комментариями) для разделения длинных фрагментов текста на логические части. Чаще всего пустые строки оставляют между определениями функций, группами связанных описаний или между частями тел длинных функций.

Незначащие пробельные символы перед первым значащим символом строки называются ***отступом (indent)***. Они используются для задания «глубины» конструкции на данной строке, чтобы показать, частью какой другой конструкции она является. Например, блоки и их содержимое или отдельные операторы входят как части в операторы, изменяющие естественную последовательность выполнения программы, и являются контролируемыми ими частями программы. Чтобы показать это отношение, их отступ увеличивают относительно отступа самих контролирующих конструкций. Если они сами содержат вложенные контролирующие операторы или блоки, отступ увеличивается снова. Иерархия уровней отступа показывает, что чем контролируется. Отступы чаще всего являются кратными ширине табуляции. Наиболее часто используемые — 8, 4 или 2 пробела. Поскольку границы блоков в языке C++ обозначаются фигурными скобками, встаёт вопрос и о их расположении. Рассмотрим пример:

```
1  int main()
2  {
3      int i,j;
4      for(i=0;i<10;++i){
5          for(j=0;j<10;++j)
6              std::cout << (i+j)%10;
7          std::cout << '\n';
8      }
9      std::cout << "Input an integer: ";
10     if(std::cin >> i){
11         i = i*2+3;
12         std::cout << "Output: " << i << '\n';
13     }else
14         std::cerr << "Input error.\n";
15     return 0;
16 }
```

В данном примере используется ширина табуляции (или отступа) в 4 пробела. Автором этого текста выбраны следующие соглашения:

- Если контролируемый оператор всего один, блок вокруг него не обязателен (строка 6).
- Содержимое блоков или отдельные контролируемые операторы имеют отступ на один шаг больше, чем их контролирующий оператор или заголовок определения функции.
- Открывающая фигурная скобка тела функции пишется на отдельной строке на уровне самого определения (строка 2).
- Открывающая фигурная скобка остальных блоков пишется в конце строки, на которой содержится основная часть управляющей конструкции, к которой данный блок относится (строка 4).

- Закрывающая фигурная скобка ставится на отдельной строке на уровне той конструкции, к которой относится данный блок (строки 8,16).
- Ключевое слово `else` пишется на отдельной строке вместе с символами `}` от блока в положительной ветви (строка 13) и `{` от отрицательной, если они есть.
- Необязательные пробелы ставятся только вокруг операции присваивания в операторе-выражении и операциях `<<` и `>>`.

Это стиль, которого придерживается автор этой книги. Он является, за исключением последнего пункта, соответствующим стилю K&R, названного так по инициалам авторов первой книги по языку C, где впервые применялся. Приведём для примера тот же текст, оформленный в стиле Whitesmiths с шириной отступа в 2 пробела:

```
1  int main()
2    {
3    int i, j;
4    for (i = 0; i < 10; ++i)
5      {
6      for (j = 0; j < 10; ++j)
7        {
8        std::cout << (i + j) % 10;
9        }
10     std::cout << '\n';
11     }
12   std::cout << "Input an integer: ";
13   if (std::cin >> i)
14     {
15     i = i * 2 + 3;
16     std::cout << "Output: " << i << '\n';
17     }
18   else
19     {
20     std::cerr << "Input error.\n";
21     }
22   return 0;
23   }
```

Здесь приняты следующие соглашения

- Все контролируемые операторы являются блоками, даже если контролируется всего один оператор (строки 7-9).
- Содержимое блоков и соответствующие им фигурные скобки пишутся на одном уровне, который на один больше, чем у контролирующей конструкции (строки 19-21).
- Обе фигурные скобки пишутся на отдельных строках (строки 19,21).
- Ключевое слово `else` пишется на отдельной строке (строка 18).
- Незначащие пробелы ставятся вокруг всех бинарных операций.

Оба стиля позволяют, например, легко понять, что оператор, печатающий один символ перевода строки, относится к телу первого оператора `for`, а не второго. По мере изучения дополнительных конструкций языка будут появляться новые вопросы, на которые предстоит ответить программисту, определяющему свой стиль.

Не показанным в данном примере аспектом стиля является решение о записи цепочки из операторов `if`, расположенных в отрицательных ветвях друг друга. Соответствующий записанным выше требованиям вариант выглядит следующим образом:

```
1  // Фрагмент
2  if(a==1){
3      // ...
4  }else
5      if(a==2){
6          // ...
7      }else
8          if(a==3){
9              // ...
10         }
```

Достаточно длинная цепочка подобного вида может создать проблемы с горизонтальным пространством из-за постоянного увеличения отступа. Компромиссным путём записи этой конструкции можно было бы считать одноуровневый выбор альтернатив, например:

```
// Фрагмент
if(a==1){
    // ...
}else if(a==2){
    // ...
}else if(a==3){
    // ...
}
```

Наконец, рассмотрим именование сущностей, т.е. выбор идентификаторов. Не считая очень ограниченного набора сущностей, имена которых фиксированы (например, `main`), все идентификаторы элементов программы, которые определяются программистом, определяются им самим. Именование сущностей по праву считается одной из фундаментальных трудностей в программировании вообще, поскольку определяет основной объём прагматического смысла программы:

```
a1 = a2*a3;
```

Какое по смыслу действие выполняет данный оператор — вычисление площади прямоугольника по сторонам или мощности по току и напряжению? Более тщательный выбор имён позволил бы с большой долей уверенности разрешить эту неопределённость.

Использование однобуквенных имён переменных удобно с точки зрения минимизации числа нажатий на клавиши, но очень быстро приводит к программам с объектами из всех букв латинского алфавита, в которых не может разобраться сам их автор через час после последнего прочтения. Следует уметь вовремя остановиться при использовании таких имён. Обычно однобуквенные имена используют только в роли счётчков итераций циклов, которым сложно найти осмысленное имя, в таком случае используют имена `i`, `j`, `k` — если хочется продолжить этот ряд дальше, следует хорошо задуматься о более подходящих именах. Другим примером частого использования однобуквенных имён, где оно оправдано, является перенос в программу математических формул из предметной области, где система обозначений хорошо устоялась, например, из физики или математики.

В большинстве случаев идентификатор является словом, несколькими словами или сокращением от них (желательно на английском языке). В языке C++ регистр в идентификаторах различается, поэтому часто используются следующие варианты:

- `lowercase`, `lowercase_with_underscores` — все буквы строчные, слова разделяются символами подчёркивания. Данный стиль используется для большинства идентификаторов в стандартной библиотеке языка C++.
- `UPPERCASE`, `UPPERCASE_WITH_UNDERSCORES` — все буквы заглавные, слова разделяются символами подчёркивания. Данный стиль используется для идентификаторов макросов в стандартной библиотеке языка C++.
- `TitleCase` — первые буквы слов заглавные, разделителей между словами нет. Стиль как в заголовках применяется во многих библиотеках для именования типов.
- `camelCase` — первые буквы слов заглавные, кроме первого, разделителей между словами нет. Данный стиль идентификаторов используется во многих библиотеках для именования обычных идентификаторов объектов и функций.

В данной книге автор будет придерживаться тех же соглашений, что и стандартная библиотека языка C++, пока речь идёт о нём.

На самом деле, рассмотренные в этом разделе аспекты стилей относятся скорее к языку C, а не C++, в котором стилевых вопросов ещё больше. Автор рекомендует уделить вопросу выработки собственного стиля особое внимание, поскольку он часто недооценивается, что затрудняет сам процесс обучения.

5.9. Примеры реализации простых алгоритмов

Рассмотрим простые алгоритмы, оперирующие фиксированным числом объектов в каждый момент времени с использованием основных конструкций структурного программирования.

5.9.1. Алгоритмы над фиксированным числом объектов

Реализуем алгоритм Евклида нахождения наибольшего общего делителя двух чисел. Воспользуемся его формальным определением из [8]:

Алгоритм 1: Алгоритм Евклида

Вход: целые числа a, b такие, что $b > a > 0$.

Выход: НОД(a, b) — наибольший общий делитель чисел a и b.

1 Определить переменные $r_{-1} = b, r_0 = a$;

2 **Пока** $r_0 > 0$ **выполнить**

3 Определить $q = \left\lfloor \frac{r_{-1}}{r_0} \right\rfloor$;

4 Определить $r = r_{-1} - qr_0$ и присвоить $r_{-1} = r_0, r_0 = r$;

5 Определить НОД$(a, b) = r_{-1}$.

В данном случае алгоритм уже записан в структурной форме. При оформлении также использовался знак равенства для обозначения операции присваивания, как и в языке C++, с добавлением слов «определить» или «присвоить», чтобы отличить их от записываемой тем же знаком операции сравнения на равенство. В другой литературе можно встретить иное обозначение: $x \leftarrow 0$ читается «присвоить x значение 0». При переводе математических алгоритмов часто удобно оставлять те же обозначения, приведя их к виду, допустимому для идентификаторов. Также следует перевести традиционные для математической записи к операциям языка C++. В данном случае присвоение $q = \left\lfloor \frac{r_{-1}}{r_0} \right\rfloor$ и последующее $r = r_{-1} - qr_0$ выполняют по смыслу операция взятия остатка, для которой в языке C++ имеется соответствующая операция. Отметим, что в данном алгоритме делается копия значений a и b в переменные r_0 и r_{-1}, однако при оформлении этого алгоритма в виде функции языка C++ при передаче ей аргументов их значения будут скопированы в объекты, являющиеся параметрами функции автоматически, так что дополнительных копий не требуется. Наконец, возложим на эту функцию обеспечение условия $b > a$. Оставшиеся проверки корректности, обеспечивающие корректность поставленной задачи, а также все операции ввода-вывода сделаем в самой функции `main`. Приведём текст полученной программы:

```
1  // Вычисление наибольшего общего делителя
2
3  #include <iostream>
4
5  // Функция gcd оперирует беззнаковыми значениями,
6  // поскольку по условию нашего алгоритма определена
7  // только для натуральных чисел.
8
9  unsigned gcd(unsigned a,unsigned b)
10 {
11     // Проверить b > a и
12     // обменять значения местами,
13     // если это не так.
14     if(b<a){
15         // Поскольку сделать два присваивания одновременно
16         // нельзя, нужен временный объект.
17         unsigned t;
18         t = a;
19         a = b;
20         b = t;
21     }
22     // Основной цикл алгоритма.
23     while(a>0){
24         unsigned r;
25         r = b%a;
26         b = a;
27         a = r;
28     }
29     return b;
30 }
31
32 int main()
33 {
```

```
34      // Ввести значения
35      std::cout << "Input two natural numbers: ";
36      unsigned x,y;
37      if(!(std::cin >> x >> y)){
38          std::cerr << "Input failed!\n";
39          return 1;
40      }
41      // Проверить введённые числа.
42      if(x<1||y<1){
43          std::cerr << "Invalid input!\n";
44          return 2;
45      }
46      // Вычислить и вывести результат.
47      std::cout << "GCD(" << x << ',' << y << ") = " << gcd(x,y) << '\n';
48      return 0;
49  }
```

В таком виде функция `gcd` может быть использована в более сложных программах.

5.9.2. Алгоритмы обработки последовательностей

Рассмотрим пример, когда обрабатывается множество значений, задаваемых пользователем, но при этом используется фиксированное число объектов. Это означает, что при обработке очередного элемента мы «не помним» всех предыдущих значений, а только некоторую информацию фиксированного объёма, не зависящую от длины последовательности. Поскольку мы «просматриваем» данные только один раз, такие алгоритмы называются ***однопроходными (single pass)***. Они применяются для решения простых задач, где этого достаточно, но иногда к ним приходится сводить более сложные задачи в условиях, когда данные поступают из некоторого источника (в данном случае, пользовательского ввода), и нет возможности предварительно сохранить их в памяти (из-за недостаточного её объёма или, как в нашем случае, из-за отсутствия пока необходимых знаний). Простым примером такого алгоритма может являться поиск максимального элемента:

```
1   #include <iostream>
2
3   int main()
4   {
5       std::cout << "Input numbers:\n";
6       // Ввести первое значение
7       int max;
8       if(!(std::cin >> max))
9           std::cerr << "No input received!\n";
10      else{
11          // Вводить остальные значения по одному.
12          int current;
13          while(std::cin >> current)
14              // Обновить наше "состояние" max,
15              // если очередной элемент current
16              // больше него.
17              if(current>max)
18                  max = current;
19          std::cout << "max = " << max << '\n';
20      }
21      return 0;
22  }
```

Обратным примером является случай, когда требуется сгенерировать некоторую последовательность в заранее определённом порядке. Например, поскольку вывод на экран мы пока умеем производить только по строкам сверху вниз и слева направо внутри строки, при выводе на экран двумерного псевдографического изображения, следует организовать циклы, осуществляющие вывод именно в этом порядке, а на очередной итерации вычислять, что следует вывести в соответствующей позиции. Покажем это на примере вывода прямоугольника, заштрихованного по диагонали.

```
1   #include <iostream>
2
```

```
3  int main()
4  {
5      int row,column;
6      // 10 строк
7      for(row=0;row<10;++row){
8          // 10 столбцов в каждой строке
9          for(column=0;column<10;++column)
10             // Диагональная штриховка с шагом в 1 пробел между линиями.
11             std::cout << ((row+column)%2?'/':' ');
12         // Перевод строки в конце строки
13         std::cout << '\n';
14     }
15     return 0;
16 }
```

5.10. Задания на структурное программирование

0. Запросите у пользователя целое число $n, n > 2$ и выведите псевдографическую аппроксимацию круга из символов # диаметра n, например для $n = 10$:

```
   ##
  ####
 ########
 ########
##########
##########
 ########
 ########
  ####
   ##
```

   Для больших n должно быть видно, что это действительно круг, а не ромб, как кажется в приведённом выше случае. Поскольку знакоместо моноширинного шрифта в консоли не является квадратным, даже если математически всё сделано верно, круг будет казаться сжатым по горизонтали на экране (как в этом тексте), это нормально.

1. Запросите у пользователя целое число $n, n \geqslant 0$ и выведите число единичных бит в его двоичной записи.
2. Запросите у пользователя целое число $n, n \geqslant 0$ и выведите его цифры по порядку в столбец, начиная с младшей.
3. Запросите у пользователя целое число $n, n > 1$ и выведите, является ли оно простым или нет.
4. Запросите у пользователя целое число $n, n \geqslant 0$, вещественные число x и коэффициенты $a_0 \dots a_n$ и выведите значение многочлена $P(x) = \sum_{i=0}^{n} a_i x^i$ в заданной точке.
5. Запросите у пользователя последовательность вещественных чисел (продолжайте ввод, пока он успешен), после чего выведите среднее арифметическое всех неотрицательных из введённых чисел.
6. Прямая на плоскости, не параллельная оси y, задаётся уравнением с угловым коэффициентом $y = kx + b$. Запросите у пользователя параметры двух прямых k_1, b_1, k_2, b_2 и выведите координаты точки их пересечения, если она есть. Решать дополнительные задачи, пока удаётся вводить новые четвёрки параметров.
7. Выведите таблицу значений функции

$$f(x) = \begin{cases} x^2 + 3x - 2, & \text{если x делится на 2, но не делится на 6} \\ x + 5, & \text{если x делится на 3, но не делится на 6} \\ 42, & \text{если x делится на 6} \end{cases}$$

   в интервале $[a, b]$ с шагом в 1. Целые a и b запросите у пользователя. Значения в точках, где функция не определена, не выводить.

8. Выведите значение суммы
$$\sigma = \sum_{x=0}^{\infty} \frac{1}{x^2 + bx + c}$$
для заданных пользователем значений b и c. Суммирование следует прекращать, когда очередной член ряда меньше текущего значения суммы более чем в 10^6 раз.

9. Программа отслеживает перемещение предмета по плоскости. Предмет изначально находится в начале координат. При каждом перемещении пользователь вводит вещественные координаты вектора (x, y), определяющего перемещение объекта на очередном шаге. После каждого шага выводите новые координаты объекта. Завершите работу программы, если после очередного перемещения объект оказался на расстоянии 1 или больше от начала координат.

10. Точка c комплексной плоскости $\mathbb{C}$ с координатами (x,y) принадлежит множеству Мандельброта, если последовательность $z_0 = (0, 0), z_{n+1} = z_n^2 + c$ остаётся ограниченной при $n \to \infty$. Известно, что если один из членов последовательности находится на расстоянии более 2 от начала координат, то последовательность не ограничена. Пользователь вводит координаты точки, а программа определяет, принадлежит ли (потенциально) заданная точка множеству. Считать, что если за 100 итераций последовательность не удалилась от начала координат дальше, чем на 2, то она, скорее всего, принадлежит множеству Мандельброта. (Возведение комплексного числа в квадрат: $(x, y)^2 = (x^2 - y^2, 2xy)$.)

11. Пользователь загадывает целое число из $[0; 100]$. Программа пытается отгадать его, выводя свои предположения. Пользователь отвечает вводом числа 0, если программа угадала (после чего работу следует завершить), числа меньше нуля, если загаданное число меньше догадки программы, и больше нуля — если больше.

12. Запросите у пользователя координаты слона на шахматной доске (и строки, и столбцы обозначьте 1–8). Выведите координаты всех полей, находящихся под его боем.

13. Запросите у пользователя вещественные размеры почтового конверта и письма (прямоугольников). Программа пытается поместить письмо в конверт. Помещать письмо в конверт можно только одним из двух вариантов, когда стороны письма и конверта параллельны без перекосов. Если письмо не влезает, его складывают пополам так, чтобы более длинная сторона укоротилась вдвое и пробуют снова. Выведите всю последовательность изменений размеров письма, пока оно, наконец, не влезло.

14. Запросите у пользователя два непробельных символа. Выведите таблицу соответствия символов их кодам в интервале между кодами введённых символов, включая концы, в виде `код - символ` по одному на строку. Используйте беззнаковые коды в интервале $[0; 255]$.

Глава 6

От описания до единиц трансляции

В данной главе будут рассмотрены фундаментальные свойства имён как сущностей, из которых состоит программа, и их описаний. Мы рассмотрим описания отдельных объектов и функций, а также их основные атрибуты. Будет рассмотрено устройство функций и их взаимодействия, а также необходимые средства для поддержания этого взаимодействия между разными единицами трансляции. Нашей итоговой целью будет рассмотрение разработки и функционирования программ из нескольких единиц трансляции, что позволит создавать проекты любой сложности и понять роль и местоположение стандартной библиотеки языка C++.

6.1. Описания и определения

В этом разделе будут рассмотрены характеристики имён в их связи с сущностями, которым они соответствуют. Мы уже знакомы с применением имён в виде идентификаторов для обозначения объектов, функций и меток.

Перед применением имени в программе, оно в большинстве случаев должно быть описано (declared), чтобы транслятор знал, какая сущность ей обозначается. Процесс сопоставления использованию имени в тексте единицы трансляции соответствующего ему описания называется поиском имён (name lookup).

Все имена вводятся в программу в той или иной её области, называемой ***областью описания (declarative region)***, которая является наибольшей потенциальной частью программы, в которой его использование ***допустимо (valid)*** для того, чтобы именовать соответствующую ему сущность. В общем случае область программы, в которой может использоваться имя, то есть его ***область видимости (scope)*** определяется как ***потенциальная область видимости (potential scope)***, из которой исключают вложенные области описания, где содержатся описания того же имени. Таким образом, область видимости имени может состоять из нескольких несмежных участков программы. Области видимости относятся к одному из нескольких видов, определяемых по расположению соответствующего описания в тексте программы относительно других её элементов.

6.1.1. Видимость идентификаторов меток

Опишем область видимости имён меток, используемые в известных нам конструкциях помеченного оператора и оператора `goto`, как наиболее простой пример.

Они — единственный элемент программы, имеющий областью видимости ***функцию (function)*** — независимо от вложенности блока, в котором расположен помеченный оператор, соответствующий идентификатор может использоваться в операторах `goto` в любом месте функции, как до, так и после него. Само использование идентификатора в конструкции помеченного оператора является его неявным описанием, это также один из редких случаев, когда описание имени может быть найдено поиском имён по тексту программы после его использования. Ввод в программу меток никак не влияет на описания имён других сущностей, а поиск имён меток в операторе `goto` игнорирует все описания, не являющиеся метками.

Метки особого вида `case` и `default`, используемые в теле оператора `switch` не имеют в своём составе имён, так что к ним поиск имён не применим.

6.1.2. Описания и определения объектов и функций.

Основной интерес представляют атрибуты имён объектов и функций.

Имена объектов и функций вводятся в программу с помощью ***описаний (declaration)***. Одно описание может задавать атрибуты сразу нескольких имён или конструкций, производных от них, каждая из которых называется ***описателем (declarator)***. Синтаксически описание состоит из одного или нескольких ***спецификаторов описания (declaration specifier)***, среди которых должен присутствовать хотя бы ***спецификатор типа (type specifier)***, задающий тип, соответствующий описателям. За ними могут следовать описатели, перечисленные через запятую, описание завершается точкой с запятой. Простейшим описателем является идентификатор, а спецификатором типа — последовательность ключевых слов, соответствующая имени фундаментального типа. Например:

```
int x;
```

является описанием. Использованный в данном случае спецификатор типа `int` — имя типа, который становится атрибутом перечисленных в описании в виде описателей конструкций, в данном случае, идентификатора `x`. С примером, где описание без описателей имеет эффект, мы познакомимся значительно позднее. Описания задают и другие атрибуты идентификаторов, многие из которых задаются не явно, а в зависимости от расположения описания в тексте программы. Потенциальная область видимости имён, входящих в описатели, начинается сразу после окончания соответствующего описателя.

Последовательность спецификаторов описания и ключевых слов в именах фундаментальных типов не имеет значения, например, `long unsigned long` — то же, что и канонически названный тип `unsigned long long`.

Описания объектов и функций могут являться ***определениями (definition)***, которые, помимо введения имени в область описания и задания его атрибутов, определяют содержание описываемой сущности. ***Определение идентификатора объекта приводит к выделению памяти под хранение соответствующего объекта, а определение функции содержит тело, т.е. задаёт последовательность операторов, ей соответствующую.*** Неформально, описание, не являющееся определением, говорит, что «где-то существует элемент программы с такими-то свойствами, именуемый следующим образом», а определение говорит, что «существует элемент программы с такими-то свойствами, именуемый следующим образом, ***и вот его содержимое***». Правила, определяющие, является ли описание определением, будут рассмотрены ниже.

Объекты, в том числе создаваемые в результате их определений в программе, имеют атрибут, называемый ***временем хранения (storage duration)***. Он определяет, когда происходят выделение и освобождение памяти под хранение представления значения, соответствующего объекту. В течение этого времени объект существует и обладает своим основным свойством — сохранять записанное в него значение и воспроизводить его при чтении.

Имена сущностей могут обладать ***связанностью (linkage)*** — свойством, позволяющим нескольким описаниям одного и того же имени (в одной или разных областях видимости) соответствовать одной и той же сущности. Если имя имеет связанность, то она является либо ***внешней (external)***, либо ***внутренней (internal)***. Все описания имени с внутренней связанностью именуют одну и ту же сущность в рамках той единицы трансляции, в которой содержатся, а с внешней — во всей программе. Наличие описаний одного имени с разной связанностью в одной области видимости не допустимо. ***Внешняя связанность является механизмом связи разных единиц трансляции и их сущностей.*** Связанность возможна только между описаниями одного и того же имени, имеющими ***совместимые (compatible)*** типы. Нам известен пока только один случай совместимости типов — их идентичность.

В одной области видимости в общем случае не может быть более одного описания одного и того же имени объекта или функции, соответствующего разным сущностям.

Время хранения и связанность определяются в первую очередь исходя из области видимости, а также наличия дополнительных спецификаторов описания, называемых спецификаторами ***класса памяти (storage class)***, из которых описания может содержать максимум один. Два основных спецификатора класса памяти — ключевые слова `static` и `extern`.

6.1.3. Производные типы. Категория типов «функция».

Помимо имеющихся в языке базовых типов, множество которых конечно, система типов языка C позволяет создавать новые типы за счёт применения конструкций создания ***производных типов (derived type)***. Многие производные типы имеют некоторый тип, о котором говорят, что другой тип является ***производным (derived)*** от него, исходный тип при этом называют ***базовым (base)***. Конструкции создания производных типов можно (в рамках имеющихся ограничений) применять снова к другим производным типам в качестве базовых, и таким образом строить конструкции произвольной сложности. ***Категорией типа (type category)*** называют последнюю применённую конструкцию создания производного типа или сам тип, если он не является производным.

Функция (function) — использованная нами конструкция создания производного типа, соответствующая частям алгоритма программы, и потому не относящаяся к типам ***данных***. Как нам уже известно, функция характеризуется типом возвращаемого значения, а также количеством и типами своих параметров. Типы категории «функция» считаются производными от типа их возвращаемого значения. Для задания такого произведения типа описатель должен содержать конструкцию, синтаксически похожую на операцию вызова функции: пару круглых скобок с описаниями параметров функции через запятую. Каждое такое описание должно содержать ровно один описатель, даже если несколько подряд идущих параметров функции имеют один тип. Имена параметров, задаваемые в этих описаниях, не влияют на тип описываемой сущности, и, если не важны, могут быть исключены из этих описаний, спецификаторы описания и прочие синтаксические части описателя при этом остаются на месте. Например,

```
unsigned f(int,double);
```

есть описание имени `f`. Полный описатель `f(int,double)` задаёт часть типа `f` — функция с двумя параметрами типа `int` и `double` соответственно. Имена этих параметров не указаны. Тип возвращаемого значения `unsigned` задаётся спецификатором типа, общим для всех описателей в этом описании. Таким образом,

```
int f1(char),obj,f2();
```

есть описание трёх идентификаторов, входящих в три описателя, каждый из которых является производным от `int`. `f1` — функция, возвращающая `int` и имеющая один параметр типа `char`. `obj` — объект типа `int` (нет конструкции создания производного типа, спецификатор типа относится непосредственно к идентификатору). `f2` — функция, возвращающая `int` и не имеющая параметров.

Описания функций обычно дают отдельными описаниями из одного описателя, чтобы не вносить путаницы, так что предыдущий пример, скорее, исключение из традиционной практики.

Ещё один пример описания функции:

```
int f1(char c1,char c2);
```

В этом примере имена параметров указаны, и показано, что несмотря на одинаковый тип двух подряд идущих параметров, каждому соответствует отдельное описание.

Сложные описатели в языке C следует читать, начиная с идентификатора (или места, где он мог бы быть для случаев, когда он отсутствует) и далее по приоритету операций, синтаксически соответствующим конструкциям в описателе. Это лишь мнемонический приём — несмотря на синтаксическую схожесть, эти конструкции операциями не являются. В последнем примере начиная с `f1` далее видна «операция», похожая на вызов функции — пара круглых скобок с некоторым содержимым. После неё описатель заканчивается и остаётся рассмотреть спецификаторы описания. Неформально, это можно прочесть как: «если при использовании `f1` в выражении применить к нему операцию вызова функции с одним аргументом типа `char`, то в результате получится `int`», т.е. `f1` — функция, возвращающая `int` и имеющая один параметр типа `char`. Данный приём можно применять последовательно несколько раз для прочтения любых более сложных описателей, с которыми мы познакомимся позднее.

Определения функций отличаются от описаний, не являющихся определениями, наличием тела функции. Они должны также удовлетворять следующим требованиям:

- Определение функции должно содержать единственный описатель.

- Вместо точки с запятой в конце определения функции записывают составной оператор, задающий её тело.
- Определение функции не должно содержаться в другом определении функции — вложенных функций в языке C++ не существует.

Описания функций, не являющиеся определениями, называются ***прототипами (prototype)***, а описания её параметров в этом случае имеют областью видимости ***прототип (prototype)***. Она заканчивается с описателем функции, параметрами которой они являются. Идентификаторы, имеющие областью видимости прототип, не являются обязательными и не влияют на смысл программы, их область видимости служит только для запрета дублирования имён параметров одной функции. Их в большинстве случаев указывают, чтобы при чтении описания функции был ясен смысл её параметров.

Таким образом:

```
1  // Определение функции f, возвращающей void,
2  // и имеющей два параметра типа int: x и y.
3  void f(int x,int y)
4  {
5      // ...
6      // ОШИБКА: определение функции внутри
7      //         тела другой функции.
8      void g()
9      {
10     }
11 }
12
13 // ОШИБКА: каждый параметр - отдельное описание,
14 //         y b не указан тип.
15 void h(int a,b)
16 {
17 }
18
19 // Описание функции, имена параметров не имеют значения,
20 // но несут прагматическую нагрузку.
21 void i(int a,double b);
```

6.1.4. Блочная область видимости и автоматическое время хранения. Рекурсия.

Описания в виде операторов в блоках, а также описания параметров функции в определениях функций вводят в программу имена с ***блочной (block)*** областью видимости. Помимо указания вида видимости, как блочной, можно указать конкретный блок программы, который для имён в этом описании является областью описания. Параметрам функции соответствует блок-тело функции таким образом, словно их определения даны первым оператором тела функции, хотя синтаксически они в него не входят. Описания с блочной областью видимости также называют ***локальными (local)***.

Одно из основных времён хранения объектов — ***автоматическое (automatic)***. Описания объектов с блочной областью видимости без спецификаторов класса памяти являются определениями и имеют именно это время хранения. Память под объекты с автоматическим временем хранения выделяется при каждом входе в блок, с которым связано их описание, и освобождается при выходе из этого блока. Память под такие объекты может выделяться и освобождаться несколько (включая ноль) раз в процессе выполнения программы.

Объекты с автоматическим временем хранения связанности не имеют, каждое такое описание соответствует новой сущности, и потому является определением.

```
1  // Определение объекта-параметра x в определении функции.
2  // Блочная область видимости с блоком-телом функции f.
3  void f(int x)
4  {
5      // Выделение памяти под объекты x,y.
6
7      // Определение объекта с именем y
8      // с блочной областью видимости, соответствующей
9      // телу функции f.
```

```
10      int y;
11
12      // Составной оператор как очередной оператор тела функции f.
13      {
14          // Выделение памяти под объект z.
15
16          // Определение объекта с именем z с блочной областью видимости,
17          // соответствующей блоку на строках 10-14.
18          int z;
19
20          // Освобождение памяти объекта z.
21      }
22
23      // ОШИБКА: другая сущность с тем же именем, что уже описанный в этом блоке int y.
24      double y;
25
26      // ОШИБКА: другая сущность с тем же именем, что уже описанный в этом блоке int y,
27      // пускай и того же типа.
28      int y;
29
30      // Освобождение памяти объектов x и y.
31  }
```

Большинство объектов, которыми оперируют программы, имеют именно эти атрибуты.

Объекты с автоматическим временем хранения являются одним из технических средств обеспечения ***рекурсивных (recursive)*** вызовов функций — ситуаций, когда функция вызывает саму себя непосредственно (***прямая рекурсия (direct recursion)***) или через цепочку других функций (***косвенная рекурсия (indirect recursion)***). Поскольку вызов функции не вызывает завершения, а только приостановку выполнения блока, в котором находится операция вызова функции, когда управление повторно будет передано той же функции, произойдёт повторный вход в блок, являющийся её телом, без выхода из первого. Так как под объекты с автоматическим временем хранения память выделяется при входе в блок, память будет выделена повторно, и в программе будут существовать несколько экземпляров объектов, соответствующих одним и тем же именам объектов с автоматическим временем хранения в рекурсивно выполняемых блоках. Рассмотрим пример:

```
1   #include <iostream>
2
3   void count_down(int x)
4   {
5       std::cout << "Count: " << x << '\n';
6       if(--x>0)
7           count_down(x);
8   }
9
10  int main()
11  {
12      count_down(3);
13      return 0;
14  }
```

Данная программа выводит

```
Count: 3
Count: 2
Count: 1
```

В данном пример имеется прямая рекурсия в функции `count_down`. Первый раз она вызывается из функции `main` со значением аргумента, равным 3. Далее функция вызывает саму себя с аргументов 2, а затем со значением 1. В последнем случае проверка условия обходит рекурсивный вызов функции и управление доходит до конца тела этой функции первый раз за выполнение программы. Стек вызова при этом имеет следующий вид:

```
count_down(x=1)
count_down(x=2)
count_down(x=3)
main()
```

В этот момент в программе существуют три объекта, соответствующих имени `x`, в каждом из вызовов функции, поскольку параметр функции имеет автоматическое время хранения. Далее все три вызова функции завершатся, что приведёт к высвобождению памяти, хранящей все три объекта, в порядке, обратном порядку выделения.

В то время как рекурсия является одной из основных конструкций функциональных языков программирования, в преимущественно процедурных языках, таких как C++, прямая рекурсия применяется редко. Косвенная рекурсия нередко возникает в сложных программах естественным образом, не являясь принципиальным элементом её построения. Хотя некоторые алгоритмы достаточно лаконично записываются в такой форме, она не является оптимальной с точки зрения производительности для языков процедурной парадигмы. Кроме того, среда выполнения обычно имеет лимит на объём памяти, занимаемой автоматическими объектами и механизмом вызова функций, при превышении которого программа завершается аварийно без возможности обработать подобную ситуацию. Таким образом, если преимущества рекурсии не велики, и нельзя дать строгого ограничения на её глубину, следует заменить её соответствующим циклом.

Из-за особого статуса функции `main`, вызывать её самому программисту запрещено.

6.1.5. Аппаратный стек как реализация механизма вызова функций и автоматического времени хранения

Рассмотрим механизм реализации операции вызова функций и автоматического времени хранения. Мы уже столкнулись с упоминанием стека вызова. В общем случае ***стек (stack)*** — структура данных, хранящая элементы, к которой применимы две операции:

- Добавление элемента (push) — указанный элемент добавляется в стек.
- Извлечение элемента (pop) — извлечение элемента из стека с возвращением его значения. Элементы извлекаются в порядке, обратном порядку их добавления. Извлекать элементы из пустого стека нельзя.

Элемент, который подлежит удалению следующей операцией извлечения, называется ***вершиной стека (top of stack)***. Стек можно представить как стопку некоторых предметов, сверху которой можно класть новые или забирать старые. Стек также называют структурой данных типа LIFO (Last-In-First-Out).

Данное определение стека является минимальным, отражающим его структуру, но на практике над ним допускают и другие операции. Часто используется операция ***«просмотр вершины» (peek)***, которая возвращает значение в вершине стека, не извлекая его. В наиболее общем случае возможен доступ ко всем элементам стека без их извлечения, нумеруя их относительно его вершины.

Архитектура x86 поддерживает работу с одним стеком на уровне своего набора инструкций. Чтобы отличить его от других стеков, которые может использовать программа при хранении своих данных, его часто называют ***аппаратным стеком (hardware stack)***. Элементы в нём формально имеют фиксированный размер, равный разрядности архитектуры и хранятся в памяти последовательно, для чего операционная система выделяет в адресном пространстве процесса необходимое место, однако в общем случае программа может работать с ним как стеком элементов произвольного размера. В данной архитектуре стек растёт «вниз», т.е. при добавлении в стек новых элементов они хранятся по меньшим численно адресам памяти, чем те, что были добавлены до них. Адрес элемента в вершине стека хранится в регистре `ESP`, который называют ***указателем стека (stack pointer)***. Для добавления или извлечения элементов имеются инструкции `push` и `pop` соответственно. Каждая из них имеет один операнд, определяющий записываемое значение или место, куда записывается считанный элемент. Таким образом

```
push value
```

эквивалентно

```
sub esp,4
mov [esp],value
```

а

```
pop location
```

эквивалентно

```
mov location,[esp]
add esp,4
```

где `value` — помещаемое в стек значение, а `location` — место для записи извлекаемого значения.

Поскольку вызов функции может осуществляться из произвольного количества различных мест в программе, перед передачей ей управления необходимо сохранить адрес, куда следует вернуть управление после её завершения. Передачу управления функции осуществляет инструкция `call`, параметром которой является адрес первой инструкции вызываемой функции. Она сохраняет добавляет адрес возврата (текущее содержимое регистра `EIP`) в стек, после чего осуществляет безусловный переход по указанному адресу. Обратную операцию — возврат из функции — осуществляет инструкция `ret` (RETurn), которая извлекает из стека значение и сохраняет его в регистр `EIP`.

Место для хранения параметров функции и используемых ей объектов с автоматическим временем хранения также выделяется на стеке. Вместе с адресами возврата из функций эти данные образуют на стеке для каждого вызова функции структуру данных, называемую ***кадром стека (stack frame)***. Для работы с ней частио используется ещё один регистр процессора `EBP`. Дальнейшее рассмотрение будем вести на следующем примере. Скажем перед этим, что в языке ассемблера NASM однострочные комментарии начинаются с точки с запятой, а любая инструкция может быть помечена синтаксисом, аналогичным меткам языка C++, после чего имя метки можно использовать в качестве адреса помеченной инструкции.

```
1  int f(int a,int b)
2  {
3      int c,d,e;
4      c = a+b;
5      // ...
6      return d;
7  }
8
9  void g()
10 {
11     // ...
12     f(5,6);
13     // ...
14 }
```

Рассмотрим сначала вызов функции `f`, находящийся в функции `g`:

```
1      ; ...
2      push 6
3      push 5
4      call f
5      add esp,8
6      ; ...
```

Аргументы функции помещаются в стек в порядке справа налево, после чего осуществляется вызов функции инструкцией `call`. Эта инструкция аналогично безусловногому переходу `jmp`, но перед ним записывает на стек текущее значение `EIP`, то есть адрес следующей за ней инструкции. Вызов функции устроен таким образом, что после возврата из неё с точки зрения вызывающей функции не изменилось ничего, кроме записи в установленное место возвращаемого значения функции и возможного изменения некоторого набора регистров. Поскольку функция `g` отбрасывает возвращаемое значение функции `f`, остаётся только убрать со стека параметры функции, что можно сделать одной инструкцией увеличения регистра указателя стека на их суммарный размер. Рассмотрим теперь реализацию функции `f`:

```
1       ; Начало функции f
2   f:
3       ; Пролог
4       push ebp
5       mov ebp,esp
6       sub esp,12
7       ; Тело функции
8       ; c = a+b;
9       mov eax,[ebp+8]
10      add eax,[ebp+12]
11      mov [ebp-4],eax
12      ; ...
13      ; return d;
14      mov eax,[ebp-8]
15      ; Эпилог
16      mov esp,ebp
17      pop ebp
18      ; Возврат из функции
19      ret
```

Код функции содержит в начале и конце фрагменты, создающие и уничтожающие соответствующий текущему вызову функции кадр стека, называемые ***пролог (prolog)*** и ***эпилог (epilog)***. Пролог выполняет сохранение значения регистра кадра стека `EBP`, используемого в вызывающей функции в стек, затем переписывает в него текущее значение указателя стека, после чего сдвигает указатель стека в сторону его расширения на значение, необходимое для хранения всех объектов с автоматическим временем хранения, определённых в функции (в данном случае 3 объекта `c`, `d` и `e` по 4 байта каждый = 12 байт). Кадр стека функции `f` после исполнения её пролога показан на рис. 6.1.

⋮	
`b`	← `[ebp+12]`
`a`	← `[ebp+8]`
адрес возврата в `g`	← `[ebp+4]`
сохранённое значение `ebp` из `g`	← `[ebp]`
`c`	← `[ebp-4]`
`d`	← `[ebp-8]`
`e`	← `[ebp-12]`, `[esp]`

Рис. 6.1: Кадр стека функции `f`

Кадр стека устроен таким образом, что доступ к параметрам функции осуществляется по положительным, а к другим объектам с автоматическим временем хранения — по отрицательным смещениям относительно адреса, хранящегося в регистре `EBP`.

Значения, умещающиеся в 32 бита, возвращаются из функции в регистре `EAX`. Эпилог проделывает обратные прологу операции: восстанавливает положение указателя стека из регистра `EBP`, после чего восстанавливает старое значение `EBP` из стека. Инструкция `ret` (RETurn) осуществляет снятие со стека значения и запись его в регистр `EIP`, тем самым выполнения программы продолжается с запомненного места возврата.

Теперь реализация показанных частей функции `f` должна быть понятной.

Все упомянутые здесь правила о порядке вызова функции называются ***соглашениями о вызовах (calling conventions)***. Это не единственный их вариант. Перечислим теперь формально основные положения соглашений о вызовах функций, принятые в языке C++ на 32-битной платформе x86 (также называемые `cdecl` по их унаследованию от языка C):

- Параметры передаются через стек, куда помещаются по порядку, начиная с последнего.
- Возвращаемое функцией значение передаётся вызывающей через регистр `EAX`.
- Содержимое всех остальных регистров, кроме `ECX` и `EDX` должно сохраняться вызываемой функцией, если ей необходимо их использовать, она отвечает за сохранение и восстановление их значений.

- Освобождение места на стеке, выделенного для хранения параметров функции, осуществляет вызывающая функция.

Это не полные соглашения о вызовах, которые даже в рамках языка C++ на платформе x86 различаются между операционными системами и трансляторами, но они дают первое представление о том, какие варианты в организации взаимодействия функций возможны.

В целом аппаратный стек содержит последовательность кадров стека всех функций, образующих текущий стек вызовов в каждый момент выполнения программы. По адресу, хранящемуся в регистре `EBP`, расположено сохранённое значение этого регистра, используемое в функции, вызвавшей текущую, таким образом расположенные на стеке значения этого регистра во всех кадрах образуют цепочку. Рядом с ними хранятся адреса возврата в вызвавшие функции, по цепочке которых и карте соответствия адресов памяти строкам программы, предоставленной транслятором, отладчик может восстановить всю последовательность вызова функций и отобразить её в виде стека вызовов, имея доступ к памяти приостановленной программы и текущему содержанию регистров процессора в ней.

Последовательности инструкций, соответствующие прологу и эпилогу, обычно встречаются в каждой функции, и их действия могут быть выполнены за одну инструкцию `enter` и `leave` соответственно — в данном случае действия показаны явно для простоты объяснения. Можно заметить, что зависимость между значениями регистров `EBP` и `ESP` внутри функции полностью предсказуема, таким образом, достаточно только последнего, а первый может быть освобождён для использования в других целях — эта оптимизация называется ***опускание указателя кадра (frame pointer omission, FPO)***. В настоящее время она не играет большого значения, тем не менее компилятор может нарушать соглашения о вызовах с целью оптимизации там, где это возможно. Например, ***листовая (leaf)*** функция — функция, не вызывающая других функций, которая не требует места на стеке для автоматических объектов (когда все промежуточные результаты удаётся уместить в регистры), может полностью обойтись без пролога и эпилога.

6.1.6. Область видимости пространства имён и статическое время хранения

Описания, размещённые непосредственно в самой единице трансляции вне других языковых конструкций, расположены в так называемом ***глобальном пространстве имён (global namespace)***. Их иногда также называют ***глобальными (global)***.

Чтобы внести порядок и структуру в множество имён, которые используются в программах, помимо одной области описания в виде глобального пространство имён, программистом может быть введено произвольное количество поименованных пространств имён. Итак, ***пространство имён (namespace)*** — именованная область описания. Они вводятся в программу конструкцией, называемой определением пространства имён:

`namespace` namespace-name$_{\text{opt}}$ `{` declaration-seq$_{\text{opt}}$ `}`

Таким образом, определение пространства имён есть ключевое слово `namespace`, идентификатор, являющийся именем пространства имён, и последовательность описаний, относящихся к определяемому пространству имён, заключённая в фигурные скобки. Оно является описанием имени пространства имён.

Определение пространства имён считается описанием, которое может быть дано в любом другом пространстве имён, включая глобальное, но не в блоке или другой области описания. За счёт вложения определений пространств имён друг в друга можно строить произвольные иерархии, подобные дереву каталогов файловой системы. Приведём пример:

```
1  // Описание переменной в глобальном пространстве имён
2  int x;
3
4  // Определение пространства имён A
5  namespace A
6  {
7      // Описание переменной в пространстве имён A
8      double y;
9
10     // Определение вложенного пространства имён B
11     namespace B
```

```
12      {
13          // Определение функции в пространстве имён B, вложенном в A.
14          bool always_true()
15          {
16              return true;
17          }
18      }
19  }
```

Определения функций могут быть даны только в области видимости пространства имён, что и определяет невозможность их вложения.

Описания объектов в областях видимости пространств имён без спецификаторов класса памяти являются определениями со ***статическим (static)*** временем хранения. Память под объекты со статическим временем хранения выделяется до начала выполнения программы, то есть до передачи управления функции `main`, а освобождается после выхода из неё в процессе завершения работы программы. Другими словами, время хранения таких объектов — всё время выполнения программы.

```
1  // Определение глобального объекта, хранящегося всё время работы программы.
2  int x;
3
4  namespace A
5  {
6      // Определение объекта в пространстве имён A
7      // со статическим временем хранения.
8      double y;
9  }
```

Использования объектов со статическим временем хранения, особенно если они ещё и обладают областью видимости пространства имён, следует по возможности избегать. Их применение обычно является признаком плохой структуры программы, например, когда программист хочет «сэкономить» на использовании параметров функций для передачи значений между частями программы. Объекты со статическим временем хранения соответствуют по смыслу сущностям, существующим всегда в единственном числе, что часто противоречит модульной структуре любой нетривиальной программы.

Глобальное пространство имён, а также пространства имён с именем, указанным пользователем, имеют внешнюю связанность — о случаях, когда имя не указывается будет рассказано ниже. Все описания внутри определения пространства имён имеют ту же связанность, что и оно само, если не указано иного.

6.1.7. Поиск имён

До сих пор мы использовали только имена в виде отдельных идентификаторов, однако мы уже упомянули, что пространствам имён, а, следовательно, и описанным в них именам, соответствует последовательность имён окружающих пространств имён. Имена, которые содержат в себе только имя, данное непосредственно в описании сущности, называется ***неквалифицированным (unqualified)***.

Рассмотрим процесс поиска неквалифицированных имён. Область описания, в которой заключена данная, называют ***окружающей (enclosing)***, а ею саму — ***вложенной (enclosed)*** по отношению к окружающей. В общем случае его схема такова:

- Исходя из места использования имени, определяется последовательность просматриваемых при поиске пространств имён, для каждого из них определяется, осуществляется ли поиск только до точки использования имени, или нет.
- Указанные области видимости просматриваются по порядку, первое нахождение требуемого описания останавливает поиск.
- Если все области видимости просмотрены, но описания имени не найдено, программа не согласована — неизвестно, о какой сущности идёт речь.

В известных нам случаях, поиск имён осуществляется только до использования имени. Соответствующие области видимости таковы:

- Для имени, использованного в области видимости пространства имён, просматривается пространство имён, в котором оно использовано, а затем последовательность окружающих пространств имён вплоть до и включая глобальное.

- Для имени, использованном в блоке, просматривается блок, в котором оно использовано, затем последовательность окружающих блоков вплоть до и включая тело функции, а затем окружающие пространства имён вплоть до и включая глобальное.
- При использовании имени в прототипе, просматривается только список параметров этого прототипа.

```
1  int x;
2
3  namespace A
4  {
5      int x;
6
7      void f()
8      {
9          // Поиск имени x:
10         // Просматривается тело функции f до использования x,
11         // затем пространство имён A, в котором обнаруживается описание x.
12         // Если бы оно там не было найдено, дальше поиск осуществлялся
13         // бы в глобальном пространстве имён, где нашлось бы определение другого
14         // объекта с тем же неквалифицированным именем.
15         x = 5;
16
17         // ОШИБКА: t не найден, так как описан после места использования.
18         t = 6;
19     }
20
21     int t;
22 }
```

Из-за указанного алгоритма поиска имён описание имени во вложенной области видимости с тем же именем, что и в окружающей, приводит к тому, что во вложенной области описания неквалифицированный поиск имён находит второе описание вместо первого, которое недоступно. Такое явление называют ***скрытием имён (name hiding)***. Его обычно следует избегать, чтобы не внести путаницу в программу. Формально, из потенциальной области видимости первого описания `x` исключается потенциальная область описания второго, чтобы определить его настоящую область видимости, которая в данном примере состоит из двух несмежных областей: от конца описателя в первом описании до конца описателя во втором, плюс от конца определения пространства имён A до конца единицы трансляции — фактически пустая, но формально присутствующая в тексте область.

Чтобы иметь возможность именовать сущности, находящиеся вне окружающих место использования пространств имён, могут применяться ***квалифицированные (qualified)*** имена. Они имеют вид последовательности неквалифицированных имён, разделённых операцией ***разрешения области видимости (scope resolution)*** `::`. Последний элемент этой последовательности есть имя искомой сущности, а остальные указывают имена областей видимости, которые необходимо пройти для его нахождения. При поиске имён в пути к последнему имени рассматриваются только имена областей видимости. Первый элемент в списке ищется как и при неквалифицированном поиске. Все остальные члены пути, включая само искомое имя, ищутся только в той области видимости, которая указана предшествующей частью пути.

Конструкция квалифицированного имени может начинаться с операции `::`, в таком случае следующее за ней имя ищется только в глобальном пространстве имён — такие полные имена, содержащие всю цепочку пространств имён, начиная с глобального, называют ***полностью квалифицированными (fully qualified)***. Такие имена однозначно идентифицируют сущности независимо от того, где используются.

Покажем использование квалифицированных имён на примерах:

```
1  namespace A
2  {
3      // Полное квалифицированное имя этого
4      // пространства имён - ::A::B
5      namespace B
6      {
7          // Полное квалифицированное имя этого объекта - ::A::B::x.
8          int x;
9      }
```

```
}

namespace C
{
    void f()
    {
        // Квалифицированное имя.
        // Первый элемент пути A ищется
        // как неквалифицированное имя сначала
        // в теле функции f и не находится, затем
        // в окружающем пространстве имён C
        // (там такого нет), затем в глобальном,
        // где находится ::A. В нём ищется
        // B и находится. Внутри ::A::B ищется
        // x и успешно находится.
        A::B::x = 17;
    }
}

int y;

void g()
{
    // Запись в объект в глобальном пространстве имён.
    y = 45;

    // Это описание скрывает одноимённое в глобальном
    // пространстве имён.
    int y;

    // Запись в локальную переменную т.к. не квалифицированный поиск
    // находит "ближайшее" описание, скрывающее остальные.
    y = 38;

    // Выводит 38.
    std::cout << "y = " << y << '\n';

    // Выводит 45, т.к. y ищется
    // в глобальном (и только) пространстве имён.
    std::cout << "y = " << ::y << '\n';
}
```

Приведённый пример демонстрирует другую возможность использования квалифицированных имён — доступ к описаниям, скрытым с точки зрения неквалифицированного поиска.

Последовательность описаний членов того или иного пространства имён может быть ***расширена (extended)*** даже после окончания определения этого пространства имён. Для этого достаточно дать новое определение пространства имён с тем же полным квалифицированным именем. При этом потенциальная область видимости описаний в области видимости пространства имён состоит не только из области описания между фигурными скобками определения, в которых они содержаться, но и области описания всех последующих определений того же пространства имён. Пример:

```
// Определение пространства имён ::A
namespace A
{
    // Описание переменной с полным именем ::A::y.
    double y;

    // Определение пространства имён ::A::B
    namespace B
    {
        // Функция с полным именем ::A::B::always_true.
        bool always_true()
        {
            return true;
        }
```

```
15      }
16  }
17
18  // Ещё одно определение пространства имён A.
19  // Поскольку уже есть определение
20  // пространства имён A в глобальном пространстве имён,
21  // прошлые определения видны в нём --- это расширение.
22  namespace A
23  {
24      void f()
25      {
26          // y видно из прошлого определения A.
27          y = 5.6;
28      }
29
30      // B, окружённое A, окружённое глобальным,
31      // уже было и объединяется с ним.
32      namespace B
33      {
34          void f()
35          {
36              // always_true видно при поиске
37              // имени в ::A::B.
38              if(always_true()){
39                  // ...
40              }
41          }
42      }
43  }
44
45  namespace C
46  {
47      // A, окружённого C не было,
48      // это определение нового пространства имён.
49      namespace A
50      {
51          void f()
52          {
53              // ОШИБКА: не видно описания y, были
54              // просмотрены тело ::C::A::f, ::C::A, ::C и ::.
55              y = 4.2;
56          }
57      }
58  }
```

Теперь ясно, что, например, использованное нами в программах имя `std::cout` — квалифицированное имя объекта с именем `cout` из пространства имён `std`, точнее, `::std`.

Если часто нужно имя, которое требует квалификации, можно сократить её или совсем избавиться от неё разными способами.

Рассмотрим так называемое ***описание `using`***. Это специальная форма описания, допустимая в области видимости пространства имён или блока. Его синтаксис — ключевое слово using, квалифицированное имя и точка с запятой. Оно является ещё одним описанием сущности, именованной указанным квалифицированным именем, размещённым в той области видимости, где дано описание `using`. Описание `using` никогда не является определением. Имя, вводимое этим описанием, то же, что и в исходном. Приведём пример:

```
1   namespace A
2   {
3       int x;
4   }
5
6   namespace B
7   {
8       // Описание имени x, соответствующего
9       // описанию ::A::x в ::B.
10      // Теперь ::A::x и ::B::x --- описания,
```

```
11      // соответствующие одному и тому же объекту.
12      using A::x;
13  }
14
15  namespace long_name
16  {
17      void f()
18      {
19      }
20  }
21
22  void g()
23  {
24      // Описание имени f, соответствующее ::long_name::f.
25      using long_name::f;
26
27      // f находится неквалифицированным поиском как описание
28      // using в этом же блоке, так что считается, что найдено ::long_name::f.
29      f();
30  }
```

Описание `using` не применимо к пространствам имён, но похожего эффекта можно достичь с помощью определения ***псевдонима пространства имён (namespace alias)***. Его синтаксис — ключевое слово `namespace`, идентификатор, пунктуатор `=`, квалифицированное имя пространства имён и точка с запятой. Это определение вводит указанный идентификатор в виде ещё одного имени пространства имён, указанного справа от знака равенства и также может быть размещено в блоке или пространстве имён. Пример:

```
1   namespace long_name
2   {
3       namespace other_long_name
4       {
5           int x;
6       }
7   }
8
9   void f()
10  {
11      long_name::other_long_name::x = 1;
12
13      // Псевдоним пространства имён:
14      namespace lo = long_name::other_long_name;
15
16      // Теперь в этом блоке можно писать короче:
17      lo::x = 1;
18  }
```

> Псевдоним пространства имён может быть переопределён в той же области видимости, но только если фактическое пространство имён, которое ему соответствует (неважно как найденное) при этом не изменится.

Эта конструкция применяется нечасто, так как есть решение, которое упрощает именование ещё больше. Когда в одной области видимости используется много имён из другого, требующих квалификации, использование множества описаний `using` остаётся громоздким. Вместо этого можно воспользоваться ***директивой `using`*** (не путать с описанием `using`!). Её синтаксис — ключевые слова `using namespace`, квалифицированное имя пространства имён и точка с запятой. Вновь, эта директива также может быть расположена в области видимости блока или пространства имён и имеет следующие эффекты на поиск имён:

- При неквалифицированном поиске имён, имена из указанного в директиве пространства имён видны, словно описаны в самом глубоком пространстве имён, одновременно являющимся окружающим как для пространства имён, указанного в директиве, так и для того, в котором она сама находится (напрямую или в теле функции). При использовании этой директивы может оказаться так, что будет найдено сразу несколько описаний разных сущностей — это является ошибкой, так как поиск имён должен давать однозначный результат

(случай, когда все найденные описания относятся к одной и той же сущности, допустим). Пример:

```
1  namespace A
2  {
3      namespace B
4      {
5          namespace C
6          {
7              int i;
8          }
9      }
10
11     // Ближайшее общее окружающее пространство имён
12     // для ::A::B::C, найденного по имени в директиве,
13     // и ::A, в котором она сама содержится - это
14     // ::A, т.е. теперь в ::A при неквалифицированном
15     // поиске после этой директивы видны имена из ::A::B::C.
16     using namespace B::C;
17
18     void f1()
19     {
20         // Находит не квалифицированным поиском ::A::B::C::i
21         // в ::A за счёт директивы using.
22         i = 5;
23     }
24
25     // Это описание видно только в следующих примерах,
26     // т.к. пока известные нам случаи поиска имён осуществляют
27     // поиск только до точки использования.
28     int i;
29
30     namespace D
31     {
32         // Находит ::A::B и делает здесь доступными его
33         // описания, словно они в ::A - ближайшем общем
34         // окружающем пространстве имён для ::A::B и ::A::D.
35         using namespace B;
36
37         // Находит ::A::B::C, увидев его в ::A за счёт
38         // предыдущей директивы, и делает его имена
39         // доступными через ::A здесь.
40         using namespace C; // ::A::C
41
42         void f2()
43         {
44             // ОШИБКА: неоднозначно найденное имя.
45             // Дойдя поиском до ::A обнаруживается два i,
46             // соответствующих разным сущностям:
47             // - описанное напрямую в ::A.
48             // - видимое в ::A::D описание из ::A::B::C
49             //   за счёт директивы using.
50             i = 5;
51         }
52     }
53 }
```

Хотя в большинстве случаев эффект этой директивы можно считать более простым — «сделать доступными здесь имена из другого пространства имён», можно привести контр-пример:

```
1  namespace A
2  {
3      namespace B
4      {
5          int x;
```

```
6       }
7
8       namespace C
9       {
10          int x;
11
12          namespace D
13          {
14              using namespace ::A::B;
15
16              void f()
17              {
18                  // Меняет ::A::C::x, поскольку он первым
19                  // находится поиском имён. Директива using
20                  // делает имена из ::A::B доступными в ближайшем
21                  // общем окружающем пространстве имён ::A,
22                  // до которого поиск просто не доходит.
23                  x = 30;
24              }
25          }
26      }
27  }
```

При не квалифицированном поиске имён директивы `using` транзитивны:

```
1   namespace A
2   {
3       int x = 10;
4   }
5
6   namespace B
7   {
8       using namespace A;
9   }
10
11  // За счёт транзитивности директив using,
12  // указание в директиве using пространства имён B включает
13  // заодно все директивы using в нём содержащиеся, в том
14  // числе и для A.
15  using namespace B;
16
17  // Находит ::A::x.
18  x = 20;
```

- При квалифицированном поиске имён для каждого компонента пути, включая последний, если пройденная часть пути задаёт поиск в конкретном пространстве имён, и искомого описания в нём нет, осуществляется повторная попытка найти нужное имя в объединении пространств имён, которые указаны директивами `using` в текущем. Если и в этих пространствах имён нет нужного описания, берутся директивы `using` из них, и осуществляется следующая итерация поиска, и так пока не будет найдено имя или кончатся транзитивные директивы `using`.

```
1   namespace A
2   {
3       int x;
4       int y;
5   }
6
7   namespace B
8   {
9       int x;
10      using namespace A;
11  }
12
13  void f()
14  {
```

```
15      // B ::B есть x, находится он.
16      B::x = 15;
17      // B ::B нет y, но есть директивы using,
18      // которые "номинируют" ::A. Поиск в ::A
19      // находит ::A::y, который и меняется.
20      B::y = 35;
21  }
22
23  namespace A1
24  {
25      using namespace A;
26  }
27
28  namespace B1
29  {
30      using namespace B;
31  }
32
33  namespace C
34  {
35      using namespace A1;
36      using namespace B1;
37  }
38
39  void g()
40  {
41      // B ::C нет y, дальше просматриваются
42      // объединение указанных директивами using
43      // ::A1 и ::B1. В них тоже нет y, поэтому вместо
44      // них просматривается объединение ::A и ::B.
45      // В нём (через ::A) находится y, который и есть результат поиска.
46      C::y = 3;
47
48      // ОШИБКА: в объединении ::A и ::B есть два описания x,
49      //         и это описания разных сущностей.
50      C::x = 5;
51
52      // ОШИБКА: весь просмотр транзитивных пространства имён не находит
53      //         ни одного описания z.
54      C::z = 15;
55  }
```

Если пространство имён расширено после директивы `using`, использующей его, новые члены также становятся видны в общем окружающем пространстве имён:

```
1   namespace A
2   {
3       // Пока ничего.
4   }
5
6   using namespace A;
7
8   // Расширение ::A
9   namespace A
10  {
11      int x;
12  }
13
14  void f()
15  {
16      // x из расширения после директивы using виден.
17      x = 40;
18  }
```

Автор рекомендует не злоупотреблять директивами `using`. Иерархия пространств имён позволяет обособить схожие имена, требуемые разными частями программы, и исключить

их скрытие и конфликты. Эта проблема хорошо видна в языке C, где они отсутствуют. Привычка использовать эту директиву, особенно в глобальном пространства имён, возвращает эту проблему, которую пространства имён были призваны решить.

6.1.8. Инициализация

Рассмотрим вопрос о начальном значении объектов. Эти значения могут быть заданы в определении путём записи после описателя имени, соответствующего объекту, ***инициализатора (initializer)***. Все объекты ***инициализируются (initialize)*** одним или несколькими способами. Рассмотрим эти способы, имеющиеся в языке C++ и условия их применения. Многие из этих способов для нас пока не различаются по эффектам, мы приводим полную классификацию, поскольку она потребуется далее. Некоторые способы включают в себя другие.

Многие из этих способов включают в себя ***список инициализаторов (initializer list)***: список выражений (в том числе пустой или из одного элемента), разделённых запятыми и заключённый в фигурные скобки. Список выражений в некоторых случаях может использоваться на месте, где обычно применяются отдельные выражения, но это возможно только в строго оговоренных случаях, поскольку сам по себе список инициализации выражением не является и, в том числе, не обладает типом. Способы инициализации, содержащие в себе синтаксис списка инициализаторов, называют ***инициализацией списком (list initialization)***.

- ***Инициализация нулём (zero initialization)***. Инициализация нулём записывает во все байты представления объекта значение 0. Для всех фундаментальных типов это соответствует нулевым значениям (числам разных типов, ложному логическому значению). Эта инициализация применяется к объектам со статическим временем хранения перед любой другой инициализацией.
- ***Инициализация по умолчанию (default initialization)***. Смысл инициализации по умолчанию — совершение минимально возможного набора действий по инициализации объекта. Все пока известные нам типы данных в таком случае не инициализируются вовсе и содержат неопределённые значения, чтение которых запрещено. Синтаксически отсутствие инициализатора в определении задаёт инициализацию по умолчанию.

```
1  // Инициализируется сначала нулём, как со статическим временем хранения,
2  // затем по умолчанию (нет эффекта). Начальное значение --- 0.
3  int x;
4
5  void f()
6  {
7      // Объект с автоматическим временем хранения без инициализатора
8      // инициализируется только по умолчанию ---
9      // начальное значение неопределённое, считывать его нельзя.
10     int a;
11 }
```

- ***Инициализация копированием (copy initialization)***. Инициализация копированием задаёт начальное значение объекта как копию значения указанного выражения. К инициализации копированием относят инициализацию параметров функции аргументами, возвращаемого значения функции в виде фиктивного объекта оператором `return`, а также использование синтаксиса инициализатора в виде пунктуатора `=` и выражения, задающего начальное значение. При этом могут происходить неявные преобразования типов.

  Вместо выражения во всех указанных выше конструкциях может записываться список инициализаторов, содержащий требуемые начальные значения: на месте аргумента в операции вызова функции, выражения в операторе `return` и выражения справа от `=` в инициализаторе описателя. Поскольку мы пока знакомы только со скалярными объектами, элементов в таком списке может быть не более одного.

  Отметим, что несмотря на схожесть синтаксиса с операцией простого присваивания, инициализация и присваивание — принципиально разные явления в языке, хотя пока для нас похожи. Идеологическая разница в том, что первое есть задание начального значения объекта, тесно связанное с выделением под него памяти, а второе — изменение значения уже существующего объекта. В этой конструкции токен `=` является пунктуатором, а не операцией.

```
1  // Объект x получает копию указанного значения 15, инициализация копированием.
2  int x = 15;
```

```
3
4  // Инициализация копированием из списка.
5  int y = {23};
6
7  int f(int v)
8  {
9      // Инициализация копированием
10     // возвращаемого значения функции.
11     return v;
12 }
13
14 void g()
15 {
16     // Инициализация параметра v копированием
17     // (с неявным преобразованием double->int).
18     f(15.3);
19 }
20
21 void h()
22 {
23     // Функция f вызывается с инициализацией её параметра
24     // копированием из списка. Само выражение вызова
25     // функции используется в инициализаторе объекта t
26     // в определении, это тоже инициализация копированием.
27     int t = f({5});
28 }
29
30 double i()
31 {
32     // Инициализация копированием из списка возвращаемого значения функции.
33     return {3.5};
34 }
```

- ***Прямая инициализация (direct initialization)***. Прямая инициализация концептуально создаёт новый объект *напрямую* по заданным для его создания значениям без осуществления копирования. Различие между прямой инициализацией и инициализацией копированием во многом только формальное, как может показаться сейчас, т.к. для фундаментальных типов они не различаются по результату. Разницу мы покажем в дальнейшем, когда будем рассматривать возможности классовых типов по контролю своего жизненного цикла. Синтаксически прямая инициализация задаётся парой круглых или фигурных скобок, в которых через запятую указаны значения, из которых производится инициализация. В случае использования круглых скобок также может (но не всегда) возникать двусмысленность описания. Известные нам случаи применения операции `static_cast` формально описаны в стандарте языка как прямая инициализация фиктивного объекта нового типа значением старого.

```
1  void f()
2  {
3      // Определение s типа double с прямой инициализацией
4      // значением 2. Как описание функции трактоваться
5      // не может, т.к. литерал 2 не может быть описанием параметра.
6      double s(2);
7
8      // Вариант с фигурными скобками.
9      double t{3.5};
10 }
```

- ***Инициализация по значению (value initialization)***. Идея инициализации по значению — создание самого «простого», но детерминированного значения объекта. Она состоит из инициализации нулём с последующей инициализацией по умолчанию. К сожалению, ни английский термин, ни его дословный перевод не раскрывают смысла этого вида инициализации.

  Форма инициализатора, соответствующая инициализации по значению — пара круглых или фигурных скобок (пустой список инициализаторов). Она может использоваться как при прямой инициализации, так и при инициализации копированием. Однако на практике пара

круглых скобок с нулём выражений внутри не применима, так как в описателе синтаксически идентична конструкции создания производного типа «функция» без параметров, а по правилам языка C++ такая двусмысленность в описании всегда трактуется в пользу функции. Эта проблема с двойным смыслом печально известна и получила неформальное имя «most vexing parse». Наличие такого «бесполезного» синтаксиса объясняется тем, что аналогичный синтаксис применяется не только в описателях, но и других, пока не известных нам конструкциях языка, где двусмысленности нет.

```
1  void f()
2      // Инициализация по значению: нулём, затем по умолчанию (нет эффекта).
3      // Начальное значение - ноль.
4      int x{};
5
6      // Трактуется как описание функции без параметров, возвращающей int,
7      // а не как определение объекта с инициализацией по значению.
8      int y();
9  }
10
11 double g(bool)
12 {
13     // Инициализация копированием по значению возвращаемого значения функции (0.).
14     return {};
15 }
16
17 void h()
18 {
19     // Инициализация копированием по значению параметра функции g (false).
20     g({});
21 }
```

Практически любая форма инициализации допускает использование списков инициализаторов, поэтому их применение также называют ***унифицированной инициализацией (unified initialization)***. По сравнению с вариантами с круглыми скобками, этот вид не допускает двусмысленности с описанием функции.

Для инициализации объектов арифметических типов, с которыми мы пока знакомы, достаточно синтаксиса инициализации копированием — он является наиболее привычным для большинства программистов, и его предлагается применять для задания всех начальных значений объектов арифметических типов, включая ноль для случаев, когда отсутствие инициализатора не даст его, то есть для локальных переменных.

При использовании списков инициализаторов для исключения случайной потери данных запрещены ***сужающие преобразования (narrowing conversions)***:

- Преобразования из вещественных типов в целочисленные — подавляющее большинство вещественных значений не представимо в целых типах без потери информации.

```
1  // ОШИБКА: сужающее преобразование,
2  //         начальное значение объекта не соответствовало бы
3  //         заданному в тексте программы значению, если бы было
4  //         разрешено неявное преобразование.
5  int x{3.5};
```

- Преобразования между вещественными типами в строну менее точного, за исключением случая, когда происходит преобразование константного выражения, которое в точности представимо в целевом типе — то есть когда есть гарантия отсутствия потери информации.

```
1  // Скорее всего, не ошибка, т.к. 3.5 - константное выражение,
2  // и его значение типа double может быть представлено
3  // в точности типом float на большинстве архитектур, включая x86.
4  float x = {3.5};
5
6  double f()
7  {
```

```
8       return 0.;
9   }
10
11  // ОШИБКА: сужающее преобразование из double в float
12  //         не константного выражения f().
13  float y = {f()};
```

- Преобразования от целого типа к целому, когда множество значений исходного не входит полностью в множество значений второго, за исключением представимых значений константных выражений.

```
1   int x = 5;
2
3   // ОШИБКА: значение не констатного выражения x не может быть
4   //         сужено из int в unsigned, так как множество значений
5   //         первого не является подмножеством второго.
6   unsigned y{x};
```

Эти ограничения не распространяются на синтаксисы инициализации без использования списоков инициализаторов, поскольку они были введены после их закрепления в языке, и их повсеместное введение привело бы к некорректности многих ранее корректных программ:

```
1  // Нет списка инициализации, возможные потери информации в сужающих
2  // преобразованиях не рассматриваются, формальной ошибки нет.
3  // Начальное значение объекта - 3, что сбивает с толку -- в тексте записано
4  // другое значение.
5  int x = 3.5;
```

В языке C++ процесс инициализации объектов со статическим временем хранения разбит на три фазы, чтобы учесть возможность вычисления их начальных значений на этапе выполнения программы, а не трансляции:

1. Вначале все объекты со статическим временем хранения инициализируются нулём.
2. Все объекты, инициализаторы которых являются константными выражениями, получают свои начальные значения. Этот шаг вместе с предыдущим вместе называют ***статической инициализацией (static initialization)***, поскольку она выполняется при трансляции программы и результаты сохраняются в её образе, готовые к использованию запущенной программой без совершения дополнительных действий.
3. Инициализация объектов со статическим временем хранения, имеющих инициализаторы, которые не являются константными выражениями, называется ***динамической (dynamic initialization)***. Для большинства объектов в области видимости пространства имён динамическая инициализация выполняется в порядке их определения в единице трансляции. Компилятор может в некоторых случаях заменять динамическую инициализацию статической, поэтому ссылка в инициализаторе на объект, определённый далее в той же единице трансляции может давать зависящие от реализации результаты. Относительно объектов со статическим временем хранения в других единицах трансляции порядок инициализации также не определён — эта ещё одна причина, по которой использование объектов с видимостью пространства имён следует сводить к минимуму и не использовать в их инициализаторах другие подобные объекты. Проблема инициализации зависящих друг от друга объектов со статическим временем хранения является известной проблемой без хорошего решения. Динамическая инициализация может выполняться кодом инициализации программы до вызова функции `main` или даже после начала её выполнения до первого использования сущностей единицы трансляции, в которой она находится. Все эти правила дают возможность для оптимизаций и не имеют видимого различия, если не использовать сложные зависимости между инициализаторами. Инициализация локальных переменных со статическим временем хранения может быть выполнена по тем же правилам или при выполнении её оператора-определения.

Приведём пример:

```
// Инициализируется нулём как объект со
// статическим временем хранения в
// статическую фазу инициализации -
// инициализация по умолчанию не предписывает
// никаких действий для типа int.
int x;

// Инициализируется в статическую фазу сначала
// нулём, затем его константным инициализатором.
int y = 42;

int f()
{
    return 42;
}

// В статическую фазу инициализируется нулём.
// При выполнении программы до первого использования
// любой сущности этой единицы трансляции его инициализатор
// вычисляется и используется для динамической инициализации.
// Особо умный компилятор может доказать, что этот
// формально не константный инициализатор можно вычислить
// и на этапе компиляции и заменить динамическую инициализацию
// статической.
int z = f();

// Поскольку b определено после a в той же единице трансляции,
// гарантируется, что инициализатор b использует полностью
// инициализированное a.
int a = f();
int b = a;

// c может быть инициализировано:
// - статически из нуль-инициализированого d,
//   который ещё не инициализирован динамически.
// - динамически из нуль-инициализированного d,
//   который ещё не инициализирован динамически
// - динамически из статически инициализированного d.
// Выбор варианта определяется реализацией и влияет
// на начальное значение - 0 или 42, так что такие
// конструкции использовать не следует.
int c = d;
int d = f();
```

Список инициализации из одного элемента может быть использован в качестве правого операнда операции присваивания объекту скалярного типа:

```
int x;

// То же самое, что следующая строка.
x = {3};
x = 3;
```

Этот синтаксис разрешён для общности.

6.1.9. Связанность описаний функций

Поскольку функции могут быть определены только в области видимости пространства имён, а такие описания имеют ту же область видимости, что и само пространство имён, определения функций по умолчанию имеют внешнюю связанность. Это свойство позволяет вызывать функции, определённые в одной единице трансляции из других:

```
// file1.cpp

namespace A
{
```

```
5       // Определение ::A::f с внешней связанностью.
6       int f()
7       {
8           return 42;
9       }
10  }
```

```
1   // file2.cpp
2   #include <iostream>
3
4   namespace A
5   {
6       // Описание ::A::f того же типа что и в file1.cpp.
7       // Теперь оба описания (одно из которых является определением) связаны
8       // и именуют одну функцию в масштабах программы.
9       int f();
10  }
11
12  int main()
13  {
14      // Вызывает ::A::f, определённую в другой единице трансляции.
15      std::cout << A::f() << '\n';
16  }
```

Данная программа содержит две единицы трансляции, которые могут взаимодействовать друг с другом с помощью внешней связанности. ***У функции с внешней связанностью не может быть более одного определения в программе, и должно быть ровно одно, если в программе есть хотя бы одно потенциально вычисляемое использование её имени.***

Хотя в блочной области нельзя дать определение функции, описания функций в ней допустимы. Полные имена сущностей, описанных в них считаются так же, словно это описание было дано в окружающем текущую функцию пространстве имён. Такое имя тоже имеет внешнюю связанность, так что можно добавить к прошлой программе третью единицу трансляции:

```
1   // file3.cpp
2
3   namespace A
4   {
5       void g()
6       {
7           // Это описание видно только в блоке, в котором
8           // содержится, но полное имя описываемой сущности - ::A::f,
9           // т.к. void g() определена в ::A. Это описание связывается с
10          // другими двумя в программе.
11          int f();
12
13          // Вызов ::A::f, определённой в другой единице трансляции.
14          f();
15      }
16  }
```

Пока мы не знали ответа на вопрос, как написать две функции, вызывающие друг друга — любой порядок их расположения приводит к тому, что использование имени в одной из них будет до его описания. С использованием связанности описаний функций решение легко найти — достаточно ввести дополнительный прототип:

```
1   #include <iostream>
2
3   // Прототип tac с внешней связанностью.
4   void tac(int x);
5
6   void tic(int x)
7   {
8       std::cout << "tic\n";
9       if(x)
10          // Имя так уже описано (хоть и не определено), ошибки нет.
```

```
11          tac(x-1);
12  }
13
14  void tac(int x)
15  {
16      std::cout << "tac\n";
17      if(x)
18          // Имя уже определено.
19          tic(x-1);
20  }
21
22  int main()
23  {
24      tic(10);
25      return 0;
26  }
```

Часто для удобства вначале дают описания вообще всех функций, а затем связанные с этими описаниями определения, которые можно уже располагать в любом удобном порядке независимо от использования в них имён друг друга.

6.1.10. Спецификаторы класса памяти static и extern. Анонимные пространства имён.

Чтобы функции и объекты, описанные в области видимости пространства имён, не были доступны из других единиц трансляции, в первую очередь, чтобы не конфликтовать с ними, когда такой доступ не требуется, их внешнюю по умолчанию связанность можно заменить на внутреннюю. Для этого можно использовать спецификатор класса памяти `static`, который в указанных случаях задаёт описываемым именам внутреннюю связанность. После того, как оно дано на первом описании функции, его можно не указывать на остальных — они тоже будут его иметь. Давать его на следующих описаниях после того, как уже дано описание с внешней связанностью нельзя, т.к. сущность не может иметь две разных связанности сразу.

```
1   // file1.cpp
2
3   // Описание ::f с внутренней связанностью.
4   // Это вспомогательная функция, которая нужна только
5   // в этой единице трансляции.
6   static int f();
7
8   int g()
9   {
10     return f();
11  }
12
13  // Определение ::f, static не обязательно,
14  // связанность всё равно внутренняя.
15  /*static*/ int f()
16  {
17      return 42;
18  }
19
20  // ОШИБКА: внутренняя связанность после уже имеющейся внешней.
21  static int g();
```

```
1   // file2.cpp
2   #include <iostream>
3
4   // Определение ::f с внутренней связанностью.
5   // Это определение не может быть связано с f
6   // из file1.cpp, поскольку то описание имело
7   // внутреннюю связанность, которая не позволяет
8   // связываться с описаниями из других единиц трансляции.
9   // Связанность этого описания роли не играет.
10  static int f()
```

```
11  {
12      return 43;
13  }
14
15  int main()
16  {
17      // Вызывает ::f, определённую в этой единице трансляции.
18      std::cout << A::f() << '\n';
19
20      // Можно указать static для функции и в блочной области видимости,
21      // если это иначе допустимо.
22      static int f();
23  }
```

В современном C++ имеется более удобный способ придания сразу многим описаниям внутренней связанности — ***анонимное пространство имён (anonymous namespace)***. Определение анонимного пространства имён — это определение пространства имён, в котором имя не указано. Такое пространство имён всё равно имеет имя, но оно определяется транслятором, уникально во всей программе и его нельзя узнать. Чтобы дать возможность именования описаний такого пространства имён вне него, такое определение ведёт себя так, словно сразу после него дана директива `using` с его именем, так что описания в нём становятся видны в окружающем пространстве имён. Хотя это может показаться равносильным не использованию дополнительного пространства имён вовсе, у такого использования есть смысл, закреплённый в правилах языка: поскольку полное имя анонимного пространства имён и имена сущностей, описанных в нём, содержат неизвестные и неповторимые компоненты, их имена невозможно указать в других единицах трансляции, поэтому все они имеют внутреннюю связанность. Таким образом, вместо указания на каждом описании внутренней связанности явно, можно заключить их последовательность в анонимное пространство имён:

```
1   // Вспомогательные функции только для этой единицы трансляции,
2   // внутреннюю связанность за счёт анонимного пространства имён.
3   namespace
4   {
5       // Полное имя - ::неизвестное_имя::f
6       void f()
7       {
8           // ...
9       }
10
11      void g()
12      {
13          // ...
14      }
15
16      void h()
17      {
18          // ...
19      }
20
21      // Тоже внутренняя связанность.
22      namespace A
23      {
24          // Тоже внутренняя связанность - полное имя
25          // ::неизвестное_имя::A::k.
26          void k()
27          {
28              // ...
29          }
30      }
31  }
32  // Здесь как будто указано
33  // using namespace неизвестное_имя;
34
35  int main()
36  {
37      // Найдено в анонимном пространстве имён за счёт
```

```
38      // неявной директивы using.
39      f();
40      return 0;
41  }
```

В блочной области видимости спецификатор класса памяти `static` имеет для объектов совсем другой эффект — он меняет автоматическое время хранения объекта на статическое. Это можно использовать для создания функций, которые «помнят» некоторую информацию между своими вызовами и которая нужна только в них. Для таких целей это лучше, чем создавать объект со статическим временем хранения в области видимости пространства имён, поскольку он будет виден не только в этой функции:

```
1   #include <iostream>
2
3   void count()
4   {
5       // Блочная область видимости, нет связанности, но статическое время хранения,
6       // потому инициализировано нулём.
7       static int c;
8       std::cout << "Call " << ++c << '\n';
9   }
10
11  int main()
12  {
13      count();
14      count();
15      count();
16  }
```

Спецификатор класса памяти `static` для определений объектов в области видимости пространства имён, также как и для функций, изменяет связанность на внутреннюю.

Чтобы связаться с определением объекта, которое имеет связанность, потребуется дать описание объекта, не являющееся определением. Для этого служит спецификатор класса памяти `extern`, который превращает определение объекта в описание и придаёт ему связанность. При использовании в блочной области видимости, он также обеспечивает статическое время хранения, вместо обычного для описания в блоке автоматического. Такое описание подчиняется тем же правилам по взаимодействию с другими описаниями с внутренней связанностью и определению полного имени сущности, что и описания функций без спецификаторов класса памяти. Наличие инициализатора в описании «сильнее», чем `extern` — такое описание остаётся определением, и, кроме того, недопустимо в блочной области видимости, так что такую форму лучше не использовать. `extern` можно использовать и в описаниях функций, где он ничего не меняет.

```
1   // file1.cpp
2
3   // Определение объекта с внутренней связанностью и статическим
4   // временем хранения.
5   static int x;
6
7   // ОШИБКА: из-за инициализатора это --- определение, не взирая
8   //         на extern, являющееся вторым описанием сущности с внешней связанностью.
9   extern int x = 43;
10
11  void f()
12  {
13      // Описание x, связанное с предыдущим, те же атрибуты, кроме области видимости.
14      extern int x;
15  }
16
17  namespace
18  {
19      // Внутренняя связанность от анонимного пространства имён.
20      int y;
21  }
22
```

```
23  namespace A
24  {
25      // Внешняя связанность, описание.
26      // Связано с определением в file1.cpp.
27      extern int z;
28
29      void f()
30      {
31          z = 42;
32
33          // ОШИБКА: extern + инициализатор
34          extern int q = 15;
35      }
36  }
```

```
1   // file2.cpp
2
3   // Внешняя связанность, не связано с
4   // ::x из file1.cpp, т.к. у того описания
5   // связанность была внутренняя.
6   int x;
7
8   namespace A
9   {
10      int z;
11  }
```

Ещё раз напомним, что объекты со статическим временем хранения и областью видимости пространства имён следует применять только по необходимости.

После того, как элементу программы, у которого может быть описание со связанностью, не являющееся определением, дано такое описание в области видимости пространства имён, его последующие описания, включая определения, могут быть даны не только в том же пространстве имён, но и в любом окружающем. При этом вместо неквалифицированного имени в таком описании используется квалифицированное. Такие описания называют ***отделёнными (out-of-line)***:

```
1   namespace A
2   {
3       // Описания объекта и функции со связанностью
4       // в области видимости пространства имён.
5       void f();
6       extern int x;
7   }
8
9   // Отделённое определение A::f.
10  void A::f()
11  {
12      // ...
13  }
14
15  // Отделённое описание A::x.
16  extern int ::A::x;
17
18  // ОШИБКА: в ::A нет описания имени g.
19  void A::g();
```

6.1.11. Описания в операторах

На месте контролирующего выражения оператора `if` может быть дано определение с одним описателем, содержащим инициализатор, без точки с запятой в конце. В таком случае, в качестве условия проверяется значение определённого объекта после его инициализации, неявно приведённое к `bool`. Имя, введённое таким определением, видно во всём операторе `if` включая обе его ветви, и считается описанным в той же области описания, что они, если они являются блоками:

```
int f()
{
    // ...
}

void g()
{
    // Вычисление значения f(), определение объекта x с тем же значением,
    // инициализированным копированием, и использование его в качестве
    // управляющего выражения. x виден в обеих ветвях, но не далее.
    if(int x = f()){
        std::cout << "Not zero: " << x << '\n';
    }else{
        // Можно записать вместо x 0, других вариантов нет.
        std::cout << "Zero: " << x << '\n';

        // ОШИБКА: два описания разных объектов в одной области видимости.
        int x;
    }

    // ОШИБКА: x уже не видно.
    x = 1;
}
```

Подобная замена возможна и для контролирующего выражения цикла `while`. При этом описанный объект виден в теле цикла и имеет связанное с ним автоматическое время хранения: на каждой итерации этот объект создаётся, инициализируется и уничтожается после конца итерации или при выходе из цикла, когда его начальное значение после преобразования к булевскому типу ложно.

```
// Продолжение предыдущего примера

void h()
{
    // Перед каждой итерацией создаётся объект x,
    // инициализируется напрямую списком значением выражения f(),
    // и, если оно не нулевое, происходит итерация цикла...
    while(int x{f()})
        ; // ... после которой он уничтожается и тут же создаётся новый для следующей.
}
```

Нетрудно догадаться, что для оператора цикла с пост-условием такой замены не предусмотрено, т.к. в нём условие расположено после его тела.

Наиболее полезный вариант замены подобного рода допускает цикл `for`: первые выражение и точка с запятой могут быть заменены произвольным описанием. Это позволяет определить необходимую циклу переменную прямо в его заголовке.

```
#include <iostream>

int main()
{
    // i видно в выражениях 2 и 3 заголовка
    // цикла for и его теле.
    for(int i=0;i<10;++i)
        std::cout << '-';
    std::cout << '\n';

    // Для цикла for нет ограничения на число описателей.
    for(int i=0,j=1;i<10;++i,j*=3)
        std::cout << "3 ^ " << i << " = " << j << '\n';
    return 0;
}
```

В отличие от остальных операторов, в которых использование определений в заголовке нечасто и даже запрещается некоторыми стилями, в цикле `for` такая конструкция используется повсеместно.

6.2. Псевдонимы типов

Для задания других имён существующим типам применяются ***псевдонимы типов (typedef-name)***. Их вводят с помощью следующей специальной формы определения:

alias-declaration:
`using` identifier `=` type-id `;`

Эта конструкция с ключевым словом `using` определяет имя-псевдоним типа, указанное слева от пунктуатора `=`, в качестве другого имени типа, указанного справа. Описания псевдонимов типов определениями, а не просто описаниями, являются всегда. Пока это определение видно, указанное имя может использоваться в качестве спецификатора типа. Эта конструкция может применяться для укорачивания длинных имён, которые часто используются, или просто для задания более понятных имён. Видимость имени, введённого данным определением, определяется как обычно, связанности оно не имеет. Приведём примеры:

```
1  // ull - другое имя unsigned long long
2  using ull = unsigned long long;
3
4  // Объект типа unsigned long long
5  ull x = 1;
6
7  void f()
8  {
9      // Более понятное имя.
10     using byte = unsigned char;
11
12     // Тип, указываемый в операции приведения типов,
13     // тоже включает спецификатор типа, поэтому псевдоним применим и здесь.
14     byte b = static_cast<byte>(1);
15
16     {
17         // Имя псевдонима типа может быть скрыто как любое имя.
18         int byte = 3;
19     }
20 }
21
22 // ОШИБКА: byte имеет блочную видимость.
23 byte b2;
```

В определении функции конструкция создания производного типа «функция», соответствующая определяемой функции, должна быть дана явно без использования псевдонима типа:

```
1  // func - псевдоним типа "функция без параметров, возвращающая void".
2  using func = void ();
3
4  // Описание функции f без параметров, возвращающей void.
5  func f;
6
7  // ОШИБКА: определение функции, где создание соответствующего производного
8  //         типа не давно явно, а скрыто в псевдониме.
9  func f
10 {
11     // ...
12 }
```

До сих пор распространён и является единственным допустимым в языке C старый синтаксис определения псевдонимов типов. Для него в описании в числе спецификаторов описания указывается ключевое слово `typedef`. Оно меняет смысл описания следующим образом: вместо присвоения соответствующих типов именам в описателях, эти описатели становятся псевдонимами соответствующих типов. Кроме спецификаторов типа и самого `typedef` другие спецификаторы описания в этом описании недопустимы. Перепишем псевдонимы из примера выше в этой форме:

```
// Если бы не typedef, ull был бы объектом типа unsigned long long,
// а с ним он становится псевдонимом этого типа, а не объектом.
typedef unsigned long long ull;

typedef unsigned char byte;
```

Автор рекомендует пользоваться новой формой, поскольку в ней имя псевдонима всегда расположено в начале и отделено пунктуатором, а в старой форме оно может быть где угодно в середине сложного описателя, которые часто используются в задании псевдонимов для упрощения их использования. Кроме этого, только новый синтаксис может быть использован с дополнительными конструкциями обобщённой парадигмы языка C++.

6.3. Возможности предварительной обработки

Предварительная обработка (preprocessing) — обработка последовательности символов из базового набора исходных символов, которую представляет собой единица трансляции после чтения файла, в котором она содержится. Задачей предварительной обработки, в конечном итоге, является преобразование этой последовательности символов в последовательность токенов, поступающих на основную фазу трансляции, производя над исходным текстом требуемые изменения. Исторически, предварительная обработка выполнялась отдельным инструментальным средством — ***препроцессором (preprocessor)***. В настоящее время он часто не является отдельной программой, а входит в состав транслятора, но может быть вызван отдельно с помощью соответствующей опции.

Язык C++ унаследовал свой препроцессор от языка C без изменений, и следует иметь в виду, что задачи, для которых он в своё время создавался, в настоящее время могут решаться более выгодными способами. Основная проблема с препроцессором в том, что он оперирует ещё даже не токенами, а некоторой другой классификацией неделимых частей программы — ***токенами препроцессора (preprocessing token)***. Он из-за этого «не понимает» большинство конструкций языка C++: понятия «описание», «атрибуты идентификаторов», «оператор», «блок» и почти любые другие, если не указано обратное, на этапе предварительной обработки не применяются. При этом он модифицирует текст программы и, следовательно, легко может создать некорректные для основного этапа работы транслятора конструкции, которые с точки зрения программиста обычно не видимы, так как для этого нужно специально запускать фазу предварительной обработки отдельно, чтобы увидеть её влияние на текст программы. Поэтому мы опишем минимум необходимых сведений о его возможностях и работе, а в процессе дальнейшего изучения отметим средства, которые позволят минимизировать его использование.

Следующие действия, выполняемые до основной фазы трансляции, относят к обязанностям препроцессора:

- Пара подряд идущих символов \ и «перевод строки» удаляются из текста, что позволяет записывать в несколько строк файла программы то, что будет далее считаться одной строкой. Это бывает удобно использовать вместе с другими возможностями препроцессора, но может пригодиться также и со строковыми литералами:

```
// Выводит ABCDEF в одну строку, символов \ и конца строки с точки
// зрения транслятора в строковом литерале нет.
std::cout << "ABC\
DEF\n";
```

- Если в конце файла нет символа конца последней строки, он добавляется, даже если последний — обратный слеш.
- Символы разбираются на токены препроцессора и последовательности пробельных символов, к которым в данном случае относится всё содержимое комментариев.
- Выполняются все ***директивы препроцессора (preprocessing directive)*** — команды для препроцессора по модификации текста единицы трансляции.
- Все символы в символьных и строковых литералах, а также escape-последовательности заменяются на соответствующие им значения в наборе символов среды выполнения.
- Смежные строковые литералы объединяются в один — мы уже пользовались этой возможностью в предыдущих примерах.

- Последовательности символов, соответствующие токенам препроцессора, преобразуются в токены основной фазы компиляции, а все пробельные символы отбрасываются.

Основные отличия токенов препроцессора от настоящих токенов в том, что препроцессор не отличает ключевые слова, считая их идентификаторами, и все виды целочисленных и вещественных литералов считает одним видом токенов — ***числами препроцессора (pp-number)***, поскольку понятия «тип» для него не существует.

Директивы препроцессора — строки программы, первый непробельный символ которых — `#`. Отметим, что в отличие от конструкций основной фазы трансляции, синтаксис директивы препроцессора включает в себя пробельные символы, поскольку ограничителем директив препроцессора являются символы перевода строк. После их обработки они удаляются из текста программы. Если это единственный символ на строке, то это — пустая директива, которая ничего не делает. Остальные директивы должны иметь следующим токеном один из специальных идентификаторов, определяющих конкретную директиву. Директивы препроцессора выполняют три основные функции: включение в единицу трансляции других текстов, макроподстановки и условную трансляцию.

6.3.1. Включение других текстов

Включение других текстов выполняется директивой `#include`, которой мы уже пользовались. После неё должна следовать последовательность символов, заключённая в угловые скобки или двойные кавычки, идентифицирующая требуемый текст. В результате выполнения директивы она заменяется текстом с указанным именем, если он найден, иначе трансляция останавливается. Форма с угловыми скобками ищет текст в определяемой реализацией последовательности мест, включая его из первого, где он будет найден. Вторая форма должна именовать файл для включения, если он отсутствует, производится попытка интерпретировать это имя текста как в первой форме. Хотя интерпретация имени текстов оставлена стандартом языка за реализацией, на практике это тоже имена файлов, которые транслятор просматривает в некоторой последовательности каталогов. Она включает в себя каталоги, определяемые самим транслятором и системой, на которой он установлен, кроме этого в неё обычно можно добавить свои пути с помощью соответствующих опций. Все известные автору трансляторы понимают прямые слеши `/` в качестве разделителей каталогов в этой директиве, поэтому этой формой следует пользоваться в любых ОС, включая Windows, где традиционно эту роль играет обратный слеш \:

```
1  // Теперь ясно: вместо этой директивы включается содержимое
2  // файла iostream из одного из путей поиска заголовочных файлов.
3  #include <iostream>
4
5  // Пути в форме с двойными кавычками задаются относительно
6  // того каталога, где находится текущий обрабатываемый файл.
7  // Если такого файла нет, будет произведена попытка найти
8  // файл file1.h в подкаталогах dir всех путей поиска включаемых
9  // файлов транслятором.
10 #include "dir/file1.h"
```

Включаемые файлы также подлежат предварительной обработке и могут сами содержать директивы включения других текстов. Таким образом, одна директива включения может привносить в единицу трансляции тысячи строк текста. Циклические включения ограничены некоторым устанавливаемым транслятором лимитом, чтобы предотвратить бесконечную рекурсию включения.

6.3.2. Макроподстановки

Препроцессор может, подобно текстовому редактору, заменять вхождения указанного идентификатора любой последовательностью токенов, для чего служит директива `#define`. Первый токен после её названия становится именем ***макроса (macro)***, а остальные — его значением (включая пустую последовательность токенов). В большинстве стилей принято записывать имена макросов заглавными буквами. После этого макрос считается ***определённым (defined)***, и все вхождения имени макроса в тексте программы заменяются его содержимым. Чтобы отменить определение макроса до конца единицы трансляции, может применяться директива `#undef` (UNDEFine) с его именем, после чего указанный идентификатор перестаёт

быть макросом. Чтобы заменить значение макроса, его сначала нужно отменить, новые определения макроса допускаются, только если он в этом месте программы не определён, или, в качестве исключения, если новая последовательность токенов совпадает со старой. Отмена не существующего определения макроса допускается и игнорируется.

```
1  // Определение макроса с именем N и значением из одного токена 10.
2  #define N 10
3
4  void f()
5  {
6      // К основной фазе трансляции эта строка превратится в последовательность
7      // токенов int, x, = и 10.
8      int x = N;
9
10 // Отмена определения макроса.
11 #undef N
12
13     // ОШИБКА: строка доходит в исходном виде, имя N не описано.
14     int y = N;
15 }
```

Подобный подход очень распространён в языке C для задания констант, но в C++ есть более безопасные средства, которые мы рассмотрим далее.

Некоторые макросы уже определены на момент начала обработки единицы трансляции. Это макросы, заданные опциями транслятора, а также предопределённые компилятором, в том числе согласно стандарту языка. Некоторые стандартные предопределённые макросы приведены в таблице 6.3.2.

Идентификатор	Значение	Описание
`__cplusplus`	201402L для C++14	Стандарт языка, поддерживаемый транслятором
`__DATE__`	"Mmm dd yyyy"	Дата трансляции
`__TIME__`	"hh:mm:ss"	Время трансляции
`__FILE__`	строковый литерал	Имя транслируемого файла
`__LINE__`	целое число	Номер текущей строки

Таблица 6.1: Предопределённые макросы

Макросы могут содержать параметры, при этом в директиве `#define` после имени макроса в круглых скобках через запятую перечисляются их имена-идентификаторы, которые затем могут быть использованы в списке токенов, соответвутющих значению макроса. После имени такого макроса в тексте программы должна идти пара скобок, подобно операции вызова функции, внутри которой через запятую даны последовательности токенов — аргументы макроса. Вся такая конструкция заменяется указанной в определении макроса последовательностью токенов, где вместо токенов, соответствующих параметрам макроса, подставляются соответствующие последовательности из списка аргументов. Все эти замены являются текстовыми подстановками практически в чистом виде и не имеют никакого отношения к механизму вызова функций, хотя такие макросы и называют иногда ***макросами-функциями (function macro)***. Это очень грубое описание приведено здесь только потому, что они могут встретится читателем в имеющихся библиотеках, включая стандартную библиотеку C++, содержащую стандартную библиотеку C, где подобное использование нередко. Автор настоятельно не рекомендует использовать макросы, особенно с параметрами, без серьёзной необходимости, и в качестве единственного примера предлагает типичную проблему, возникающую при их использовании:

```
1  // Определение макроса с параметрами.
2  #define MUL(x,y) x*y
3
4  // Текстовая замена: параметру x соответствуют токену
5  // 2, +, 3, а y - 4, +, 5. К основной фазе трансляции
6  // доходит последовательность 2, +, 3, *, 4, +, 5 -
7  // выражение с результатом 19, а не 45, как, возможно, желал программист.
8  int a = MUL(2+3,4+5);
```

По этой причине при использовании параметров в значении макроса их каждый раз окружают скобками. Из-за текстовой замены аргументы, содержащие побочные эффекты, могут после замены продублироваться и вызвать их несколько раз — автор надеется, что означенных проблем достаточно, чтобы отговорить читателя от их преждевременного использования.

6.3.3. Условная компиляция

Условная компиляция позволяет исключать из текста единицы трансляции указанные фрагменты в зависимости от условий, задаваемых константными целочисленными выражениями из литералов и значений макросов.

Директива `#if` отмечает начало фрагмента программы, подлежащего условной трансляции. После неё должно идти константное целочисленное выражение, определяющее условие. Только в этом выражении разрешена специальная операция `defined`, за которой следует (опционально, в круглых скобках) идентификатор — результат этой операции 1, если макрос с таким именем на данный момент определён, иначе 0. В этом выражении осуществляется макроподстановка, как и в другом тексте программы, после неё в выражении оставшиеся идентификаторы препроцессора, кроме `true` и `false` заменяются нулём, поскольку ни о каких именах, переменных, и прочих сущностях основного этапа трансляции препроцессор знать не может. Так как информацией о системе типов препроцессор тоже не обладает, все целочисленные вычисления производятся с максимальной доступной реализации шириной, а вычисления с плавающей точкой запрещены полностью. Полученное выражение трактуется как логическое по обычным правилам.

Конец условно включаемого фрагмента программы обозначается директивой `#endif`. Если выражение в директиве `#if` истинно, то в результате работы препроцессора удаляются только сами директивы. Если оно ложно, также удаляется весь текст между ними. Между этими директивами может располагаться директива `#else`, в таком случае транслируется только один из двух фрагментов, на которые она разбивает часть программы между `#if` и `#endif`. Пары директив условной компиляции могут быть вложены друг в друга, при этом каждая `#else` и `#endif` относится к ближайшей `#if`. Допустимы следующие сокращения:

`#ifdef` идентификатор	↔	`#if defined` идентификатор
`#ifndef` идентификатор	↔	`#if !defined` идентификатор
`#else`	↔	`#elif` выражение

`#if` выражение

Покажем, как может использоваться условная трансляция:

```
1  void f(int x)
2  {
3  // Оставить в транслируемом тексте вывод
4  // значения x только если определён макрос TRACE_PARAMS,
5  // например, опцией компилятора.
6  #ifdef TRACE_PARAMS
7      std::cout << "Value of x: " << x << '\n';
8  #endif
9
10 // Оставить в тексте программы только блок для нужной
11 // операционной системы. Проверяемые макросы определяются
12 // трансляторами в соответствующих ОС.
13 #ifdef _WIN32
14     std::cout << "Windows\n";
15 // Сокращения elifdef нет.
16 #elif defined __linux__
17     std::cout << "Linux\n";
18 #else
19 #error Unknown operating system!
20 // Один #endif Заканчивает все группы #elif/#else,
21 // относящиеся к ближайшему #if.
22 #endif
23 }
```

В приведённом выше примере также использовалась ***директива `#error`*** — она вызывает ошибку трансляции с сообщением, содержащим токены препроцессора, следующие за ней. Она применяется редко, но в данном примере позволяет остановить компиляцию в среде

операционной системы, для которой в программе не предусмотрено реализации, например, не входящего в стандарт языка средства.

Известным приёмом является применение директив условной компиляции для преодоления недостатка многострочных комментариев в части невозможности вкладывать их друг в друга. Для комментирования фрагмента кода из нескольких полных строк его заключают между директивами `#if 0` и `#endif`. Такой фрагмент, так же как и комментарий, будет безусловно исключён из единицы трансляции в процессе предварительной обработки. В отличие от настоящих многострочных комментариев, такая конструкция может быть заключена внутри другой такой же с желаемым результатом — каждая директива `#endif` будет относиться к своей парной `#if 0`.

6.4. Заголовочные файлы

Хотя прямых средств поддержки модульной парадигмы программирования в языке C++ нет, единицу трансляции принято рассматривать как контейнер для средств обеспечения некоторой общей функциональности. Мы уже упомянули о скрытии необходимых только в реализации данной единицы трансляции функций и объектов со статическим временем жизни от других путём применения к ним внутренней связанности.

Интерфейс (interface) — средства и правила взаимодействия двух отдельных систем. Применительно к частям программ, интерфейсом являются языковые средства, используемые одной из частей программы, чтобы воспользоваться возможностями, предоставляемыми другой. Поскольку в языке C++ единственной границей, которую можно использовать для разделения программы на части является единица трансляции, они и рассматриваются в качестве ***чёрных ящиков (blackbox)***, осуществляющих взаимодействие между собой за счёт предоставленных интерфейсов: известны их внешние характеристики, но не внутреннее устройство.

Для описания интерфейса единицы трансляции в языке C++ используется ***заголовочный файл (header file)***, который содержит текст на языке C++, но использует по традиции расширение `.h` или `.hpp`. Автор предлагает пользоваться вторым вариантом, чтобы отличить заголовочные файлы единиц трансляции на языке C++ от C, поскольку они могут встречаться в одной программе одновременно. Заголовочный файл обычно содержит:

- Описания функций и объектов, которые составляют интерфейс модуля и предоставляют доступ к его функциональности.
- Описания используемых в интерфейсе производных типов данных.
- Вспомогательные макроопределения.

Заголовочный файл включается директивой `#include` в единицы трансляции, которым требуется доступ к этому интерфейсу, включая саму единицу трансляции, которая содержит его реализацию — помимо требуемых определений типов данных, это позволяет давать определения функций интерфейса в произвольном порядке, независимо от того, какая из них использует имена других. Таким образом решается обозначенная нами ранее проблема синхронизации описаний в нескольких единицах трансляции — описания даются один раз в заголовочном файле, который затем подключается везде, где это требуется. Чтобы устранить возможные конфликты с именами в глобальном пространстве имён, содержимое заголовочных файлов и, следовательно, саму реализацию интерфейса единицы трансляции, следует заключить в именованное пространство имён или несколько, отражающих иерархию данного модуля в структуре программы или библиотеки. Такое выделение общей темы описаний часто также позволяет сократить их имена, сохраняя контекст смысла этих имён. Хороший заголовочный файл содержит в комментариях краткую документацию описаний, данных в нём, таким образом он же является первичной документацией интерфейса единицы трансляции. ***Задача компиляции и последующей компоновки нескольких единиц трансляции — обязанность системы сборки. Не включайте сами единицы трансляции в другие директивой `#include` — это обычная ошибка новичков.***

Заголовочный файл может быть включён в файл с расширением `.cpp`, если требуется в реализации этой единицы трансляции. Если же интерфейс одной единицы трансляции использует возможности другой, то возможно включение одного заголовочного файла из другого. При этом возникает дерево зависимостей единиц трансляции и их интерфейсов от других интерфейсов. Сам термин «дерево» подразумевает отсутствие циклических зависимостей, которых требуется избегать. При использовании многих интерфейсов часто возникает ситуация, что один и тот же заголовочный файл включается косвенно в одну единицу трансляции несколько раз. Легко представить, например, что одна единица трансляции включает в себя два никак

не связанных заголовочных файла, не подозревая, что каждому из них требуется один и тот же третий заголовочный файл. При этом не все элементы, входящие в заголовочный файл допускают повторное описание (даже идентичное). Для предотвращения повторного включения заголовочных файлов используется приём на основе условной компиляции, называемый ***защита подключения (include (header) guards)***.

Единицы трансляции могут содержать и другие вспомогательные определения объектов и функций со связанностью, не входящие в интерфейс единицы трансляции, поскольку они нужны только в данной единице трансляции — их относят к ***деталям реализации (implementation details)***. Чтобы избежать случайных конфликтов, им придают внутреннюю связанность, в заголовочный файл описания при этом, разумеется, не включают.

Продемонстрируем эти все описанные понятия на примере программы из нескольких единиц трансляции.

```
// Файл cards.hpp

#ifndef CARDS_H
#define CARDS_H

namespace cards
{
    // Проверка и классификация номеров банковских карт.

    // Псевдоним типа для хранения номеров карт.
    using card_t = unsigned long long;

    // Возвращает истину, если номер карты является 16-значным
    // и контрольный разряд имеет правильное значение.
    bool is_valid(card_t cn);

    // Возвращает тип карты по её номеру.
    // Не проверяет контрольный раряд на корректность.
    // Распознаются следующие типы карт:
    // 'A' - American Express,
    // 'M' - Mastercard,
    // 'm' - Maestro,
    // 'V' - Visa/Visa Electron.
    // Для номеров карт остальных эмиттентов, включая
    // ошибочные, возвращается нулевой символ.
    char type(card_t cn);
}
#endif
```

```
// Файл cards.cpp

#include "cards.hpp"

namespace cards
{
    namespace
    {
        bool number_in_range(card_t cn)
        {
            return cn>=1000000000000000u&&cn<=9999999999999999u;
        }
    }

    bool is_valid(card_t cn)
    {
        if(!number_in_range(cn))
            return false;
        // Алгоритм Луна, стандарт ISO/IEC 7812.
        unsigned sum = 0;
        for(int i=0;i<16;++i){
            unsigned digit = cn%10;
            cn /= 10;
            if(i%2){
```

```
                digit *= 2;
                if(digit>9)
                    digit -= 9;
            }
            sum += digit;
        }
        return !(sum%10);
    }

    char type(card_t cn)
    {
        if(number_in_range(cn)){
            card_t first2 = cn/100000000000000;
            if(first2==34||first2==37)
                return 'A';
            if(first2>=51&&first2<=55)
                return 'M';
            if(first2==50||(first2>=56&&first2<=69))
                return 'm';
            if(first2>=40&&first2<=49)
                return 'V';
        }
        return '\0';
    }
}
```

```
// Файл main.cpp

#include <iostream>

#include "cards.hpp"

int main()
{
    std::cout << "Enter card number: ";
    cards::card_t cn;
    if(!(std::cin >> cn)){
        std::cerr << "Failed to read card number!\n";
        return 1;
    }
    if(!cards::is_valid(cn)){
        std::cout << "Invalid card number!\n";
        return 2;
    }
    switch(cards::type(cn)){
        case 'A':
            std::cout << "This is an American Express card.\n";
            break;
        case 'M':
            std::cout << "This is a MasterCard card.\n";
            break;
        case 'm':
            std::cout << "This is a Maestro card.\n";
            break;
        case 'V':
            std::cout << "This is a VISA card.\n";
            break;
        default:
            std::cout << "This is an unknown card that appears valid.\n";
            return 3;
    }
    return 0;
}
```

Файл `cards.cpp` содержит реализацию проверки и классификации номеров банковских карт, а `cards.hpp` является соответствующим ему заголовочным файлом — заголовочные

файлы отличаются расширением, чтобы совпадать по имени с соответствующим файлом реализации. Файл `main.cpp` содержит функцию `main` простой программы, которая запрашивает у пользователя номер карты и выводит её тип.

Рассмотрим заголовочный файл. Пара директив препроцессора в его начале и парная к ним в конце и являются упомянутой защитой заголовка: всё содержимое заголовочного файла компилируется только при условии, что некоторой макрос не определён, при этом первой же строкой внутри этого условного блока этот макрос и определяется (последовательность токенов, соответствующая макросу может быть пустой, он всё равно при этом становится определённым). Таким образом, при возможных повторных включениях (отсутствующих в данном примере для этого заголовочного файла) все, кроме первого включения заголовочного файла в каждой единице трансляции не будут включать его содержимого. Отметим также, что включение заголовочного файла, являющегося частью программы, обычно производится указанием его имени в двойных кавычках, чтобы его поиск осуществлялся в том же, каталоге, что и остальные файлы программы, если система сборки не настроена так, чтобы включать отдельные каталоги проекта в путь поиска включаемых файлов транслятором. Имя макроса, используемое в этой конструкции не несёт смысловой нагрузки и может быть произвольным. Обычно выбирается псевдослучайное имя или имя, производное от имени самого файла. В любом случае оно должно быть уникальным во всей программе (а лучше — глобально), при использовании имён, производных от имени файла, следует быть особенно внимательным, если файлы проекта хранятся в нескольких каталогах и в них имеются файлы с одинаковыми именами — тогда обычно в имени макроса отражают весь путь относительно некоторого базового каталога.

Основное содержимое этого заголовочного файла полностью прокомментировано на уровне документации, достаточной для использования этого интерфейса независимо от его реализации — в этом и состоит его основное свойство. Клиенту этого интерфейса не нужно знать, каков алгоритм проверки банковских номеров и как их хранить — все эти детали скрыты за псевдонимами типов и реализациями функций. В том числе они могут быть изменены в достаточно широких пределах без надобности серьёзного (или вообще) изменения кода, использующего её — это основное достоинство модульной структуры программы, позволяющее разрабатывать огромные по объёму программы многими людьми совместно так, что никто не знает устройства всех деталей всех модулей (что для достаточно больших проектов невозможно).

Отметим один из наиболее важных фактов: ***заголовочные файлы содержат только описания объектов и функций, таким образом трансляция самой реализации описанного в нём интерфейса происходит только один раз, независимо от числа использований этого интерфейса***. Теоретически можно, скажем, просто выделить определения часто используемых функций в отдельный файл, который включать в каждую другую единицу трансляции, где они нужны. При этом потребуется придать им внутреннюю связанность, поскольку иначе будет иметься несколько определений сущностей с внешней связанностью, что недопустимо. Даже в этом случае потребуется многократное транслирование одного и того же кода, что, очевидно, неэффективно, а на практике ведёт к множеству других разнообразных проблем. Именно поэтому эти файлы называют заголовочными — они содержат только описания, а не определения большинства сущностей.

Иногда заголовочные файлы самодостаточно, например, содержат только определения типов, в таком случае они могут не иметь парной единицы трансляции. В начале заголовочного файла или единицы трансляции последовательность включений обычно располагают в отсортированном по именам порядке с разбиением на группы: заголовок данной единицы трансляции, другие заголовки данной программы, заголовки сторонних библиотек и заголовки стандартной библиотеки языка.

6.5. Программы из нескольких единиц трансляции

Рассмотрим работу с программами из нескольких единиц трансляции в среде Qt Creator. Простой проект на языке C++ по умолчанию состоит из одной единицы трансляции `main.cpp`.

Попробуем воспроизвести пример из предыдущего раздела, для этого начнём с замены содержимого этой единицы трансляции соответствующим кодом из примера. После этого в окне «Проекты» в контекстном меню проекта выберем пункт «Добавить новый... (Add New...)» и выберем тип файла C++ — Файл исходных текстов C++ (C++ — C++ Source File). На следующей странице можно убедится, что он будет добавлен к нужному проекту, после чего подтвердить создание — новый файл появится в разделе «Исходники (Sources)»

окна проектов и будет открыт, он изначально пуст. Аналогичным образом через добавление нового файла типа C++ — Заголовочный файл C++ (C++ Header File) можно создать парный заголовочный файл, он появится в новом разделе «Заголовочные (Headers)» окна «Проекты» и будет содержать защиту заголовка с макросом, имеющим имя, производное от имени файла. С помощью команды из того же меню «Добавить существующие файлы... (Add Existing Files...)» можно добавить к проекту уже имеющиеся файлы, они будут размещены по категориям в соответствие со своими расширениями. Перед этим настоятельно рекомендуется переместить их в каталог проекта, чтобы избежать путаницы. Если требуется удалить файл из проекта, можно воспользоваться пунктом контекстного меню соответствующего файла «Удалить файл... (Remove File...)». Если в открывшемся диалоговом окне подтверждения установить флажок «Удалить файл навсегда (Delete file permanently)», он не только будет исключён из проекта, но и удалён с диска. После добавления нужных файлов в проект и записи в них содержимого, проект можно собирать и запускать как обычно — наличие в нём нескольких единиц трансляции принципиально ничего не меняет.

Если теперь открыть файл проекта, в нём можно обнаружить следующее:

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp      cards.cpp

HEADERS +=      cards.hpp

QMAKE_CXXFLAGS += -std=c++14 -Wall -pedantic-errors
```

Как уже упоминалось, он содержит информацию для системы сборки. Для простых проектов на языке C++ используется система сборки `qmake`, рассмотрим смысл всех заданных здесь опций. Поскольку проект создаётся на переносимом языке, его файл проекта содержит в том числе опции, которые могут не иметь смысла в конкретной операционной системе, с которой вы работает в данный момент.

- `TEMPLATE = app` — устанавливает тип проекта как самостоятельную программу (в отличие, например, от библиотек).
- `CONFIG += console` — указывает, что программа будет использовать в качестве основного консольный интерфейс (критично для ОС Windows).
- `CONFIG -= app_bundle` — необходимое для работы в ОС Mac OS X консольного приложения отключение создания пакета приложения.
- `CONFIG -= qt` — отключает использование библиотеки Qt, которую эта система сборки (частью которой она сама является) подключает по умолчанию.
- `SOURCES += …` — перечисляет входящие в проект единицы трансляции.
- `HEADERS += …` — перечисляет входящие в проект заголовочные файлы. Хотя с ними в процессе сборки для используемого нами типа проекта и не нужно ничего делать, этот список используется, например, самой средой разработки для отображения списка файлов, входящих в проект.
- `QMAKE_CXXFLAGS += …` — выполненная в начале работы с проектом настройка компилятора согласно разделу 4.6.1.

Символ `\` обеспечивает склейку строк, как и в языке C++. Как мы видим, список файлов, входящих в проект, можно редактировать и напрямую, не пользуясь графическим интерфейсом интегрированной среды разработки.

`qmake` не является самостоятельной системой сборки, вместо этого по файлу проекта она создаёт файл правил для настоящей системы сборки, которая может быть разной. При работе с инструментальными средствами по умолчанию обычно применяется традиционная система сборки `make`, файл правил которой называется `Makefile`. По умолчанию в среде Qt Creator все файлы, генерируемые в ходе сборки проекта, размещаются в отдельном от каталога самого проекта месте — такая сборка называется ***теневой (shadow)*** и является правилом хорошего тона, позволяющим не засорять каталог проекта с файлами, редактируемыми напрямую программистом, продуктами сборки для этого не предназначенными. По умолчанию этот каталог расположен на одном уровне с каталогом проекта и

носит длинное имя вида `build-имя_проекта-имя_набора_инструментов-имя_конфигурации`, например `build-testproj-Desktop_Qt_5_4_1_MinGW_32bit-Debug`. В этом каталоге можно найти несколько файлов `Makefile`, содержимое которых может быть весьма объёмным — именно поэтому `qmake` берёт все нюансы создания точных правил сборки проекта для различных ОС, трансляторов и конфигураций на себя.

Можно открыть окно «Консоль сборки» и понаблюдать, например, за инструментами, вызываемыми при выборе команды «Сборка — Пересобрать всё (Build — Rebuild All)» (указаны только имена вызываемых программ и ключевые параметры):

- `make clean` — вызов системы сборки `make` с командой удалить все оставшиеся с прошлой сборки файлы — «Пересобрать всё» = «Очистить всё (Clean All)» + «Собрать всё (Build All)».
- `qmake имя_проекта.pro` — вызов `qmake` для генерации файла правил сборки.
- `make` — вызов системы сборки `make` в обычном режиме сборки. Она в свою очередь для этого запускает (показаны только опции, отвечающие за режимы работы транслятора):
  - `clang++ -c cards.cpp -o cards.o` — компиляция файла `cards.cpp` в объектный файл `cards.o`.
  - `clang++ -c main.cpp -o main.o` — компиляция файла `main.cpp`.
  - `clang++ -o имя_проекта main.o cards.o` — компоновка образа программы с именем, совпадающим с именем проекта, из объектных файлов, соответствующих входящим в неё единицам трансляции.

Сборка с предварительной очисткой была выбрана, чтобы показать полный список команд по сборке проекта, в обычном режиме сборки, как нам известно, выполняется только необходимый минимум команд, соответствующий необходимым командам для актуализации изменений, внесённых после прошлой сборки.

6.6. Этапы трансляции программы на практике

Рассмотрим этапы трансляции программы и её представление на каждом из них на примере компилятора `clang` в ОС Linux и простого примера. Этот транслятор поддерживает несколько языков, чтобы транслировать программы на C++ его следует вызывать по имени `clang++`. Чтобы не усложнять, обойдёмся без надлежащего в настоящей программе заголовочного файла:

```
1  // file1.cpp
2
3  #include <iostream>
4
5  extern int x;
6  int f();
7
8  #define OK 0
9
10 int main()
11 {
12     std::cout << "result = " << x+f() << '\n';
13     return OK;
14 }
```

```
1  // file2.cpp
2
3  int x = 2;
4
5  static int y = 3;
6
7  static int g()
8  {
9      return y;
10 }
11
```

```
12  int f()
13  {
14      return g();
15  }
```

Чтобы в простейшем случае собрать эту программу, достаточно в командной строке (приглашение которой обозначим здесь `$`) вызвать сам транслятор, указав ему имена единиц трансляции, при этом будет создан файл программы с именем по умолчанию `a.out`:

```
$ clang++ file1.cpp file2.cpp↵
$ ./a.out↵
5
```

Программа `clang++` на самом деле является оболочкой над большим числом инструментов, которые она вызывает для выполнения фактической работы в зависимости от выбранного режима — в данном случае выполнения всех этапов трансляции сразу. Попробуем запросить выполнение только этапа предварительной обработки опцией `-E` для первой единицы трансляции:

```
$ clang++ -E file1.cpp↵
# 1 "file1.cpp"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 320 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "file1.cpp" 2

# 1 "/usr/bin/../include/c++/v1/iostream" 1 3
⋮
namespace std { inline namespace __1 {

extern __attribute__ ((__visibility__("default"))) istream cin;
extern __attribute__ ((__visibility__("default"))) ostream cout;
extern __attribute__ ((__visibility__("default"))) ostream cerr;
extern __attribute__ ((__visibility__("default"))) ostream clog;
extern __attribute__ ((__visibility__("default"))) wistream wcin;
extern __attribute__ ((__visibility__("default"))) wostream wcout;
extern __attribute__ ((__visibility__("default"))) wostream wcerr;
extern __attribute__ ((__visibility__("default"))) wostream wclog;

} }
# 4 "file1.cpp" 2

extern int x;
int f();

int main()
{
    std::cout << "result = " << x+f() << '\n';
    return 0;
}
```

Хотя результат этапа предварительной обработки формально уже не является текстом, а последовательностью токенов, многие трансляторы позволяют вернуться к текстовому представлению по запросу. Результат в этом режиме выводится по умолчанию непосредственно на экран и является весьма объёмным (бóльшая часть здесь опущена), поскольку включает содержимое заголовочного файла стандартной библиотеки, включённого явно, а также множества других, включённых через них косвенно. Строки, начинающихся с символа `#`, в данном случае являются не директивами препроцессора, а маркерами, показывающими, из каких файлов и каких их строк образовался этот текст. В нём можно обнаружить, например, описание используемого нами объекта `std::cout`. Также

можно заметить, что использованный в операторе `return` макрос `OK` текстово заменён соответствующей ему последовательностью токенов — числовой константой 0.

Результат генерации кода на языке ассемблера в синтаксисе Intel можно запросить с помощью опций `-S -masm=intel`, по умолчанию он будет помещён в файл с именем, совпадающим с именем единицы трансляции, и расширением `.s`. Мы не будем приводить здесь пример этого файла, поскольку очень похожее содержимое обнаружится на следующем этапе.

Компилятор `clang` является частью программной инфраструктуры LLVM, которая имеет свой внутренний язык для представления текстов программ. Затребовать этот текст, сходный с языком ассемблера можно с помощью опций `-S -emit-llvm`, он будет размещён по умолчанию в файле с расширением `.ll`.

Сделаем шаг вперёд: выполним компиляцию одной единицы трансляции (опция `-c`), начав со второй:

```
%$ clang++ -c file2.c↵
$ file file2.o↵
file2.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (GNU/Linux), not stripped
```

При компиляции создаётся ***объектный файл (object file)***, по умолчанию имеющий имя файла единицы трансляции и расширение `.o`. Объектный файл — файл, содержащий часть ***образа программы (program image)*** — информации, составляющей программу в памяти на момент начала её выполнения. Утилита `file` показывает, что этот файл является файлом формата ELF (Executable and Linkable Format), используемого в ОС Linux для представления объектных файлов. Он уже не является текстовым, и для представления его содержимого в читаемом человеком виде потребуются дополнительные средства. Воспользуемся утилитой `objdump` с требуемыми опциями (не относящиеся к текущему обсуждению части её вывода удалены):

```
$ objdump -xds -M intel file1.o↵
file2.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000001a  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  00000000  00000000  0000005c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000064  2**2
                  ALLOC
⋮
SYMBOL TABLE:
00000010 l     F .text        0000000a _ZL1gv
00000004 l     O .data        00000004 _ZL1y
00000000 g     F .text        00000010 _Z1fv
00000000 g     O .data        00000004 x%
⋮
Contents of section .data:
 0000 02000000 03000000                    ........
⋮
Disassembly of section .text:

00000000 <_Z1fv>:
   0:        55                           push   ebp
   1:        89 e5                        mov    ebp,esp
   3:        83 ec 08                     sub    esp,0x8
   6:        e8 05 00 00 00               call   10 <_ZL1gv>
   b:        83 c4 08                     add    esp,0x8
   e:        5d                           pop    ebp
   f:        c3                           ret

00000010 <_ZL1gv>:
  10:        55                           push   ebp
```

```
11:       89 e5                         mov    ebp,esp
13:       a1 04 00 00 00                mov    eax,ds:0x4
                     14: R_386_32          .data
18:       5d                            pop    ebp
19:       c3                            ret
```

Объектный файл хранит часть информации, составляющей программу на момент начала её выполнения — это код функций программы и данные, соответствующие объектам со статическим временем хранения. Дальнейшее обсуждение несколько упрощено, чтобы остановиться на существенных моментах. Информация в объектном файле разбита на части — ***секции (section)***, которые отличаются некоторыми характеристиками, в первую очередь ограничениями доступа к памяти, которые должны быть установлены на эти данные операционной системой после загрузки образа программы в его адресное пространство. Различают доступ на чтение, запись и исполнение считанных инструкций (не всё, что можно прочесть, можно выполнять). Ограничения по допустимым действиям позволяют ОС выполнять различные оптимизации использования памяти, а также отвечают за обнаружение недопустимых операций, попытки совершения которых являются следствием наличия в программе ошибок. Например, секции, доступные только на чтение и потому гарантировано имеющие в этой программе одно и то же значение всегда, могут быть загружены в оперативную память единожды и отображены в адресные пространства любого числа копий программы. Таким образом экономится оперативная память — независимо от числа параллельно работающих экземпляров одной программы данные секций только для чтения загружены в память в единственном экземпляре. Перечислим основные секции и их назначение:

- `.text` — секция текста. Эта секция хранит машинный код, соответствующий функциям. Для неё разрешены чтение и исполнение содержащихся в ней инструкций.
- `.data` — секция данных. Содержит начальные значения объектов со статическим временем хранения, отличные от нулевых байтов. Разрешено чтение и запись.
- `.bss` — секция нулевых данных. Соответствует объектам со статическим временем хранения, представлением начального значения которых являются исключительно нулевые байты. Эта секция присутствует только в оглавлении объектного файла с указанием своего размера — сами нулевые байты не хранятся для сохранения места. В процессе загрузки образа программы операционная система выделяет указанный объём памяти и заполняет его нулями. Это техническая сторона реализации инициализации нулём в первую очередь объектов со статическим временем хранения. Как и для секции данных возможен доступ на чтение и запись.
- `.rodata` — секция данных только для чтения. Содержит данные объектов со статическим временем хранения, которые не подлежат изменению во время выполнения программы. Доступна только на чтение.

Секция нулевых данных в данном объектном файле имеет нулевой размер, что ожидаемо — в этой единице трансляции не определено объектов со статическим временем хранения с нулевыми начальными значениями. Секции данных только для чтения в этом объектном файле нет.

В объектных файлах используется термин, которым можно назвать как функцию, так и объект: ***символ (symbol)*** — это некий фрагмент информации одной из секций, о котором может быть известен его адрес, размер, тип и некоторые другие атрибуты. Объектный файл содержит ***таблицу символов (symbol table)***, в которой содержится информация о символах содержащихся в нём, в том числе тех, которые доступны для использования другими единицами трансляции, и символах, которые требуются этому объектному файлу, но в нём отсутствуют. Символы идентифицируются по именам, которые в простейшем случае совпадают с соответствующими идентификаторами. К языку C++, правда, этот простейший случай не применим, поскольку ему требуется кодировать в именах символов информацию о их полных квалифицированных именах, а для функций — о типе и количестве их аргументов. Такие имена называют ***декорированными (decorated)***. Причину сохранения типа функций в их именах мы рассмотрим позднее.

Рассмотрим секцию текста, представленную в виде дизассемблированного кода с аннотациями (синтаксис несколько отличается от NASM, но довольно близок, поскольку

тоже является производным от синтаксиса Intel). Символы `_Z1fv` и `_ZL1gv` — декорированные имена функций `f` и `g` соответственно. В них нетрудно обнаружить пролог и эпилог практически в стандартном виде. В функции `f` содержится вызов функции `g`, а в `g` — чтение объекта `y`, адрес которого задан как смещение в 4 байта относительно секции данных. Действительно, выше в разделе «Contents of section .data» можно обнаружить шестнадцатеричный дамп на 8 байт, которые и составляют представлению начальных значений объектов `x` и `y` со статическим временем хранения.

Таблица символов данной единицы трансляции содержит все четыре определённых в ней символа с указанием их типов — `F` (Function) или `O` (Object). Также указаны их смещения относительно начала секций, их содержащих и их размер (для функций это объём кода). Связанность этих символов отражена в соседнем столбце бувами `l` (Local) и `g` (Global) — глобальными являются символы с внешней связанностью, доступные для других единиц трансляции, а локальные — нет, это символы с внутренней связанностью.

Скомпилируем оставшуюся единицу трансляции и рассмотрим полученный объектный файл. Он содержит значительный объём кода и данных, которые оказались его частью за счёт включения стандартных средств языка, приведём только интересующие нас фрагменты:

```
SYMBOL TABLE:
00000000 l     O .rodata.str1.1        0000000a .L.str
00000000 g     F .text        0000006e main
00000000         *UND*        00000000 _Z1fv
00000000         *UND*        00000000 _ZNSt3__14coutE
00000000         *UND*        00000000 x
⋮
Contents of section .rodata.str1.1:
 0000 72657375 6c74203d 2000               result = .
⋮
Disassembly of section .text:

00000000 <main>:
   0:        55                           push   ebp
   1:        89 e5                        mov    ebp,esp
   3:        83 ec 28                     sub    esp,0x28
   6:        8d 05 00 00 00 00            lea    eax,ds:0x0
                     8: R_386_32           _ZNSt3__14coutE
   c:        8d 0d 00 00 00 00            lea    ecx,ds:0x0
                     e: R_386_32           .L.str
  12:        c7 45 fc 00 00 00 00         mov    DWORD PTR [ebp-0x4],0x0
  19:        89 04 24                     mov    DWORD PTR [esp],eax
  1c:        89 4c 24 04                  mov    DWORD PTR [esp+0x4],ecx
  20:        e8 fc ff ff ff               call   21 <main+0x21>
                     21: R_386_PC32            _ZNSt3__1lsINS_11char_traitsIcEEEERNS_13ba
  25:        8b 0d 00 00 00 00            mov    ecx,DWORD PTR ds:0x0
                     27: R_386_32          x
  2b:        89 45 f8                     mov    DWORD PTR [ebp-0x8],eax
  2e:        89 4d f4                     mov    DWORD PTR [ebp-0xc],ecx
  31:        e8 fc ff ff ff               call   32 <main+0x32>
                     32: R_386_PC32            _Z1fv
  36:        8b 4d f4                     mov    ecx,DWORD PTR [ebp-0xc]
  39:        01 c1                        add    ecx,eax
  3b:        8b 45 f8                     mov    eax,DWORD PTR [ebp-0x8]
  3e:        89 04 24                     mov    DWORD PTR [esp],eax
  41:        89 4c 24 04                  mov    DWORD PTR [esp+0x4],ecx
  45:        e8 fc ff ff ff               call   46 <main+0x46>
                     46: R_386_PC32            _ZNSt3__113basic_ostreamIcNS_11char_traits
  4a:        b9 0a 00 00 00               mov    ecx,0xa
  4f:        89 04 24                     mov    DWORD PTR [esp],eax
  52:        c7 44 24 04 0a 00 00         mov    DWORD PTR [esp+0x4],0xa
  59:        00
  5a:        89 4d f0                     mov    DWORD PTR [ebp-0x10],ecx
  5d:        e8 fc ff ff ff               call   5e <main+0x5e>
                     5e: R_386_PC32            _ZNSt3__1lsINS_11char_traitsIcEEEERNS_13ba
```

```
62:       31 c9                    xor    ecx,ecx
64:       89 45 ec                 mov    DWORD PTR [ebp-0x14],eax
67:       89 c8                    mov    eax,ecx
69:       83 c4 28                 add    esp,0x28
6c:       5d                       pop    ebp
6d:       c3                       ret
```

Поскольку в этой единице трансляции также есть описания сущностей с внешней связанностью `f` и `x`, они присутствуют в таблице символов, но их размеры и адреса нулевые и помечены маркером `*UND*` — это неопределённый (UNDefined) символы, которые, тем не менее, используются этой единицей трансляции, хотя их местоположение транслятору не известно. Это ожидаемый эффект описаний, не являющихся определениями, сущностей со связанностью.

По адресу `31` видна инструкция вызова функции, которая должна в этом месте вызывать функцию `f`, определённую в другой единице трансляции. Аннотация под этой инструкцией показывает, что адрес в этой инструкции надлежит исправить на адрес функции `f`, когда он станет известен. Байты, которые сейчас соответствуют адресу этой функции, содержат фиктивные значения и просто резервируют место под настоящее значение. Инструкция по адресу `25` содержит аналогичную ссылку на адрес объекта `x`. Наконец, по адресу `6` то же происходит с объектом `std::cout` — `_ZNSt3__14coutE` есть его декорированное имя. Полноценное определение этого объекта дано в стандартной библиотеке языка C++.

Функция `main` определена в этой единице трансляции и её одноимённый символ находится в таблице символов с известными характеристиками. Он имеет внешнюю связанность, чтобы она могла быть вызвана кодом инициализации стандартной библиотеки языка. Также можно заметить, что содержимое строкового литерала попало в подсекцию данных только для чтения, адрес этих данных загружается в регистр процессора инструкцией по адресу `c`.

Наконец, рассмотрим работу компоновщика. Как и отражено в его названии, редактор связей получает на вход набор объектных файлов, составляющих программу и выполняет две операции:

- Объединение одноимённых секций объектных файлов. Бóльшая часть информации при этом оказывается расположенной по адресам, отличным от таковых в их исходных объектных файлах — адреса в каждом файле начинаются с нуля и при объединении в одну секцию нулевой начальный адрес остаётся только у содержимого одной единицы трансляции. Если код не является ***позиционно-независимым (position-independent code, PIC)***, компоновщик исправляет содержащиеся в секциях адреса по вспомогательной информации, содержащейся в объектном файле — ***таблице перемещений (relocation table)***. Её содержимое утилита `objdump` отображает параллельно с ассемблерным дампом, например, после инструкции по адресу `13` в секции текста первой рассмотренной нами единицы трансляции указано `14: R_386_32 .data` — «адрес, расположенный по смещению `14` задан относительно секции данных в том виде, в котором она представлена в этом объектном файле, и если эта информация переместится, то этот адрес требует коррекции». Другие подобные записи соответствуют указаниям, где в объектном файле оставлены «дырки» для адресов объектов, которые в данной единице трансляции не известны, с указанием имён соответствующих символов.
- Для каждого символа, который требовался, но отсутствовал в конкретном объектном файле, осуществляется поиск его как предоставляемого для внешнего использования в остальных. При нахождении единственной кандидатуры происходит вписывание реального адреса символа в оставленные для этого пустые места. Если найдено ни одного или более одного символа с искомым именем, компоновщик выдаёт ошибку. Ситуация с несколькими определениями одного символа с внешней связанностью является ошибочной, поскольку неизвестно, какое из них следует использовать.

Полученный набор объединённых секций с исправленными адресами самодостаточен и является образом программы, готовым для загрузки и исполнения операционной системой. Таким образом, ***механизм внешней связанности имён реализуется компоновщиком***.

Два полученных объектных файла можно скомпоновать и запустить полученную программу на выполнение:

```
$ clang++ -o program file1.o file2.o↵
$ file program↵
program: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
  interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, not stripped
$ ./program↵
5
```

В данном случае опцией `-o` задано желаемое имя выходного файла, она может применяться в любом режиме трансляции. Утилита `file` сообщает, что образ программы также хранится в формате `ELF`, но в отличие от объектных файлов является исполняемым. Заинтересованный читатель может попытаться применить утилиту `objdump` и к нему — результат не будет здесь приведён, поскольку черезчур громоздок.

Если механизм заголовочных файлов вместе с компоновкой позволяют избежать многократной компиляции кода в рамках одной программы, то под ***библиотекой (library)*** подразумевается хранилище символов для многократного использования во многих программах. Имена необходимых для сборки программы библиотек передаются компоновщику вместе со списком объектных файлов, составляющих программу. Стандартная библиотека языка C++ обычно учитывается компоновщиком автоматически. Различают статические и динамические библиотеки.

Статическая библиотека (static library) — архив, содержащий объектные файлы. Эти объектные файлы просматриваются наряду с составляющими программу при поиске необходимых единицам трансляции программы символов и добавляются к её образу по мере надобности. В рассматриваемой нами цепочке инструментов статические библиотеки хранятся в файлах с расширением `.a`.

Динамические библиотеки (shared library) не включаются в образ программы, когда она использует их символы. Вместо этого информация о требуемых программе динамических библиотеках и символах из них остаётся в образе программы в виде ссылок.

Во время загрузки программы, использующей динамические библиотеки, компонент операционной системы, называемый ***динамическим компоновщиком (dynamic linker)*** отображает указанные в образе программы динамические библиотеки в адресное пространство программы и «доделавывает» работу, которую не сделал обычный компоновщик — записывает в отмеченные места действительные адреса требуемых программе символов из динамической библиотеки. Динамические библиотеки сами могут использовать динамические библиотеки, образуя дерево зависимостей. Динамические библиотеки могут иметь любое расширение, но в подавляющем большинстве случаев используется принятый в конкретной операционной системе стандарт — `.so` для Linux, `.dll` для Windows.

Перечислим основные преимущества и недостатки этих типов библиотек. Статические библиотеки за счёт более тесной связи дают больше возможностей оптимизации и позволяют создавать самодостаточные программы. Динамические библиотеки позволяют разделять общий код между разными программами, экономя место на диске и в оперативной памяти, но программы, их использующие, запускаются и работают несколько медленнее. Динамические библиотеки также являются средством построения сложных расширяемых приложений. С точки зрения исправления ошибок в программах динамические библиотеки выгоднее, поскольку, исправив саму библиотеку, автоматически исправляются и все программы, её использующие, без их перекомпоновки. Это важно при применении обновлений безопасности, чтобы не упустить старую версию библиотеки, которая могла быть включена в саму программу, если бы была статической. По этой и многим другим причинам, большинство операционных систем предпочитают динамические библиотеки для большинства компонентов.

Укажем шаги, которые необходимо выполнить операционной системе для запуска программы по её образу:

1. Выделить ресурсы, обеспечивающие программе собственное адресное пространство и независимое состояние процессора.
2. Отобразить в адресное пространство программы содержимое её образа, установив надлежащие атрибуты для соответствующих секций. Помимо секций кода и данных в образе программы содержиться информация по требуемому программе объёму стека, которое ОС выделяет в адресном пространстве процесса. На архитектуре x86 при этом задаётся начальное значение регистра `ESP`.

3. Если программа использует динамические библиотеки, динамический компоновщик должен отобразить и связать с программой все требуемые динамические библиотеки.
4. Начать выполнение программы с указанной в её образе инструкции. На архитектуре x86 требуемый адрес заносится в регистр `EIP`.

6.7. Встраиваемые функции

Нередко возникает ситуация, что очень небольшой фрагмент кода выделяется для удобства в отдельную функцию, например:

```
1  int max(int x,int y)
2  {
3      return x>y?x:y;
4  }
```

Для таких функций особенно характерно, что их вызывают часто, в том числе в глубоких вложенных циклах. Обидным в этой ситуации является то, что код пролога и эпилога данной функциий может занимать чуть ли не больше места, чем её полезная функциональность, настолько она простая. Даже если это не так, для многих малых функций может быть полезно избавиться от накладных расходов, связанных с соглашениями о вызовах.

Среди спецификаторов описания в описаниях функций могут употребляться ***спецификаторы функций (function specifier)***, одним из которых является спецификатор `inline`, помечающий функцию как ***встраиваемую (inline)***. Формально, стандарт языка трактует это как просьбу транслятору максимально ускорить вызов этой функции.

Механизмом реализации встраиваемых функций является копирование кода, соответствующего телу такой функции непосредственно в точку её вызова так, словно соответствующий текст программы из тела функции был скопирован туда (все остальные правила языка при этом, разумеется, не нарушаются). За это часто приходится платить увеличением размера программы из-за дублирования кода встраиваемой функции, хотя для совсем тривиальных функций можно даже выиграть и в объёме кода.

Данный спецификатор является лишь просьбой компилятору, которую он вправе игнорировать, например, на нулевом уровне оптимизации. Современные компиляторы значительно лучше программиста могут решать, какие функции выгодны к встраиванию и в каком конкретном случае их вызова, но чтобы облегчить транслятору этот процесс, можно пометить функцию встраиваемой, что накладывает на неё следующие ограничения:

- Определение встраиваемой функции должно иметься во всех единицах трансляции, где она используется, чтобы у транслятора был доступ к её коду для встраивания. Встраиваемые функции не редки в интерфейсах единиц трансляции, в таком случае в заголовочный файл помещаются именно их определения. Проблемы с множественными определениями одной функции в нескольких единицах трансляции при компоновке в таком случае не происходит, исходя из стандарта языка. Реализуется это обычно введением на уровне объектного файла специального атрибута, который разрешает компоновщику выбрать из всех определений встраиваемой функции одну любую — по правилу одного определения все они должны быть идентичны, что и обеспечивается единственной копией текста определения встраиваемой функции в заголовочном файле.
- Первое описание функции со спецификатором `inline` должно предшествовать её определению в единице трансляции или быть им.

Например:

```
1  #ifndef UTIL_HPP
2  #define UTIL_HPP
3
4  // Определение встраиваемой функции в заголовочном файле
5  // интерфейса единицы трансляции, которой в данном случае
6  // может не быть вообще, так как в ней нечего определять.
7  inline bool is_odd(int x)
8  {
9      return x%2;
10 }
11
12 #endif
```

Глава 7

Указатели и массивы

В этой главе мы рассмотрим понятие указателя, наличие и возможности которого являются «визитной карточкой» языков C и C++. Автор просит читателей обратить особое внимание на этот материал, так как он имеет особую важность. Для демонстрации возможностей арифметики указателей в этой главе также будут рассмотрены первые представители не скалярных типов — массивы, а также другие конструкции, раскрывающие систему типов языка C++.

7.1. Указатели на отдельные объекты

7.1.1. Категория типов «указатель»

Указатель (pointer) — конструкция создания производных типов, значениями которой являются адреса в памяти объектов базового типа.

Согласно стандарту, указатель — тип объекта, значением которого является «ссылка» на объект. Поэтому скажем, что указатель хранит адрес объекта, которым считается адрес байта с наименьшим адресом из всей непрерывной последовательности байтов памяти, которая хранит представление объекта. Представление указателей стандартом не оговаривается, но для плоской модели памяти в рассматриваемой нами архитектуре x86 указатель является просто 32-битным номером байта в адресном пространстве процесса, что является простым и наиболее часто встречающимся случаем представления указателей. Указатели относятся к скалярным типам, но арифметическими они не являются. Как производный тип, указатель имеет базовый тип — тип объекта, адрес которого он содержит. Имя типа категории «указатель» читается как «указатель на ТИП». Хотя по смыслу представление указателей на объекты любых типов обычно одинаково, информация о базовом типе указателя требуется для большинства полезных операций, которые с этим значением можно делать. Самые простые из них — получение указателя на идентифицированный объект и наоборот — доступ к объекту по указателю на него. Для выполнения этих операций в языке C++ имеются две соответствующие операции.

Операция взятия адреса & — унарная префиксная операция, возвращающая адрес своего операнда. Операнд должен быть леводопустимым выражением — адрес есть атрибут объекта, значения, не связанные с объектами, адреса не имеют. Если в этом случае операнд имеет тип «ТИП», то результатом операции является значение типа «указатель на ТИП».

Операция разыменования (dereference) * — унарная префиксная операция, выполняющая действие, обратное взятию адреса: она имеет *леводопустимое* значение, идентифицирующее объект, указателем на который является её операнд. Её операнд должен иметь категорию типа «указатель», при этом если имя его типа «указатель на тип», то результат имеет тип просто «ТИП». Эта операция обозначается тем же токеном, что и умножение, но отличается от него по арности и синтаксису записи.

Продолжим аналогию между синтаксисом использования идентификаторов и их описаниями. Для описания объекта типа указатель, его описатель должен содержать конструкцию, синтаксически сходную с операцией разыменования, продемонстрируем это на примере:

```
int *x;
```

Это описание идентификатора `x`, которому соответствует объект типа «указатель на `int`». Объекты этой категории типа для краткости сами называют «указателями». Следуя мнемоническому правилу «синтаксис описателя повторяет синтаксис выражения, при вычислении

которого получается значение указанного в описании типа», эту строку можно прочесть как «если взять `x` и применить к нему операцию разыменования, то получится `int`». Освежим в памяти, что конструкции, входящие в описатели, относятся только к конкретному описателю, если их в описании несколько:

```
int* x,y;
```

Здесь программист, видимо, хотел показать пробелами, что желает сделать оба идентификатора указателями, но * — элемент описателя и относится только к `x`, `y` в данном случае имеет тип просто `int`. Чтобы описать оба идентификатора как указатели следует написать:

```
int *x,*y
```

Напомним, что ***когда говорят о типе со сложным описателем отдельно от какого либо идентификатора, его записывают в виде описания одного идентификатора, где сам он опущен.*** Например, имя типа «указатель на `char`» можно записать как `char *`.

Рассмотрим работу с указателями на примере.

```
1  int x = 15;
2  int *px = &x;
```

Здесь в качестве инициализатора объекта типа указатель использовано выражение, значением которого является адрес объекта, идентифицируемого `x`. Единственное, что мы пока можем сделать со значением, хранящимся в `px` — разыменовать его обратно. Посмотрим, что при этом происходит:

```
3  // Выводит *px = 15
4  std::cout << "*px = " << *px << '\n';
5
6  *px = 42;
7
8  // Выводит *px = 42, x = 42
9  std::cout << "*px = " << *px << ", x = " << x << '\n';
```

Применение обратной операции в первом операторе предсказуемо даёт то же значение, что и содержится в `x` — мы получили доступ к содержимому объекта, идентифицируемого `x` без использования этого имени. Это один из первых примеров того, что между объектами и выражениями, идентифицирующими их, нет отношения один-к-одному. Поскольку результат операции разыменования идентифицирует объект, т.е. является леводопустимым, его можно использовать в том числе в качестве левого операнда операций присваивания и тем самым осуществлять запись по адресу, являющемуся значением этого объекта. Последний оператор показывает, что запись действительно изменила содержимое объекта, идентифицируемого `x`. Таким образом, после присвоения объекту `px` адреса объекта `x` выражения `x` и `(*px)` идентифицирует один и тот же объект и полностью равноправны.

Разумеется, эта связь не вечна — указателю, как и любому объекту, можно присвоить новое значение в любой момент:

```
1  int a,b;
2  int *p = &a;
3  *p = 1;
4  p = &b;
5  *p = 2;
6
7  // Выводит a = 1, b = 2
8  std::cout << "a = " << a << ", b = " << b << '\n';
```

Отметим, что время жизни объекта-указателя и того объекта, адрес которого в данный момент он содержит, никак не связаны. Если объект, адрес которого хранится в указателе, перестаёт существовать, поскольку заканчивается его время хранения, разыменовывать такой указатель нельзя до присвоения ему нового значения — это ведёт к неопределённому поведению, поскольку происходит чтение или запись памяти, которая не соответствует никакому существующему объекту. Указатель в таком состоянии неформально называют ***висячим (dangling)***. Неопределённое поведение при доступе к памяти по адресам, не относящимся

к существующим в данный момент с точки зрения программы объектам — одна из самых часто встречаемых ошибок в программах на языке C++, которая, как и любое неопределённое поведение, к сожалению, может не проявлять себя внешне. Покажем, как такая ошибка может возникнуть:

```
int *p;
{
    int x;
    p = &x;
}
// ОШИБКА, запись по адресу, не относящемуся
// к объекту, undefined behavior.
*p = 0;
```

Рассмотрим реализацию этих операций в машинном коде на примере:

```
void f()
{
    int x,y,*px;
    // ...
    px = &x;
    // ...
    *px += y;
    // ...
}
```

Приведём вариант реализации этой функции:

```
    ; Функция f
f:
    ; Пролог, выделение памяти на стеке под 3 4-байтных объекта
    enter 12,0
    ; ...
    lea eax,[ebp-4]   ; EAX = &x
    mov [ebp-12],eax  ; px = EAX
    ; ...
    mov ebx,[ebp-12]  ; EBX = px
    mov eax,[ebx]     ; EAX = *EBX
    add eax,[ebp-8]   ; EAX += y
    mov [ebx],eax
    ; ...
    leave
    ret
```

Соответствующий псевдокод на C++, рассматривающий регистры как объекты, приведён в построчных комментариях. В этой реализации использована не встречавшаяся нам ранее инструкция `lea` (Load Effective Address), которая заносит в регистр назначения, являющийся её первым операндом, адрес, по которому бы обратилась конструкция, задаваемая её вторым операндом, будь она аргументом инструкции `mov`. В данном случае она эквивалентна последовательности

```
    mov eax,ebp
    sub eax,4
```

Однако инструкция `lea` короче и позволяет выполнить несколько операций сразу, задействуя вместо арифметических блоков процессора узлы, отвечающие за режимы адресации.

Таким образом, операция взятия адреса в машинном коде в явном виде не встречается (если не считать инструкцию `lea`, что несколько некорректно) — процессор оперирует адресами так же, как и другими числами. Операция разыменования присутствует в виде ***косвенной адресации***, выражаемой на ассемблере заключением выражений в квадратные скобки.

При отображении указателей в списке локальных объектов или заданных выражений, Qt Creator показывает связанную с ними информацию в виде дочернего элемента. Само значение

указателя — адрес, который в нём хранится, отображается в шестнадцатеричном виде. Внутри узла значения `expr` типа «указатель на `T`» будет находится дочерний элемент `*expr` типа `T`, соответствующий объекту, адрес которого в данный момент содержит указатель. Если указатель разыменовать нельзя, например, если он висячий, этот дочерний элемент может не отображаться или содержать мусор. ***Обратите внимание*** — если ни у одного из указателей нет дочерних элементов, а вместо их типа и значения сразу отображается тип и значения объекта, на который они указывают, откройте контекстное меню окна объектов и снимите выделение с пункта «Dereference Pointers Automatically» («Автоматически разыменовывать указатели»). Автор настоятельно не рекомендует пользоваться этой опцией, поскольку она очень сильно сбивает с толку и скрывает истинную связь между объектами и указателями. Почему-то при отключении этой опции становятся недоступны пункты «Open Memory Editor — Open Memory View/Editor at Pointer Address» («Открыть редактор памяти — Открыть вид/редактор памяти по адресу указателя») для объектов-указателей, но вместо их использование можно открыть редактор памяти по адресу объекта, соответствующего их дочернему узлу.

7.1.2. Использование указателей с функциями

Покажем, какие возможности предоставляют указатели, не доступные без их использования. Напишем функцию, которая находит простой делитель натурального числа с максимальной кратностью:

```
1  unsigned max_order_factor(unsigned n)
2  {
3      if(n<2)
4          // Тривиальные случаи:
5          // 1 = 1, 0 = 0.
6          return n;
7      unsigned fac = 2,
8               max_ord = 0,
9               max_ord_fac = 0;
10     do{
11         unsigned ord = 0;
12         while(!(n%fac)){
13             n /= fac;
14             ++ord;
15         }
16         if(ord>max_ord){
17             max_ord = ord;
18             max_ord_fac = fac;
19         }
20         // Первый шаг на 1 (2->3), остальные по 2.
21         fac += 1+(fac>2);
22     }while(n>1);
23     return max_ord_fac;
24 }
```

Вероятнее всего, для делителя максимальной кратности само значение кратности тоже интересно вызывающему данную функцию, однако функция всегда имеет ровно одно возвращаемое значение (включая вариант с `void`). Однако с помощью указателей можно сколько угодно значений, вычисленных в функции, сделать доступными для вызывающей её:

```
1  // Добавлен параметр max_order_factor.
2  unsigned max_order_factor(unsigned n,unsigned *max_ord)
3  {
4      if(n<2){
5          *max_ord = 1;
6          return n;
7      }
8      unsigned fac = 2,
9               max_ord_fac = 0;
10     // Определение с инициализацией заменено присваиванием.
11     *max_ord = 0;
12     do{
13         unsigned ord = 0;
```

```
14          while(!(n%fac)){
15              n /= fac;
16              ++ord;
17          }
18          // При использовании совершается разыменование.
19          if(ord>*max_ord){
20              *max_ord = ord;
21              max_ord_fac = fac;
22          }
23          fac += 1+(fac>2);
24      }while(n>1);
25      return max_ord_fac;
26  }
```

Используется этот вариант следующим образом:

```
1  unsigned mo,mof = max_order_factor(42,&mo);
```

В данном случае функции передаётся адрес объекта, в который требуется записать второе интересное вызывающей стороне значение. В теле функции этот указатель везде используется с операцией разыменования в качестве выражения, идентифицирующего объект, в котором в результате работы алгоритма оказывается требуемое значение. Таким образом функция может иметь сколько угодно «результатов». В данном случае первый результат мы по прежнему делаем доступным вызывающей функции через возвращаемое значение, но для однообразия функция, которая по смыслу возвращает более одного значения, часто записывает их все по переданным ей адресам, а её возвращаемое значение в терминах языка имеет тип `void`.

В некоторых языках и системах существует классификация параметров функции на ***входные (in)***, ***выходные (out)*** и ***двунаправленные (in/out или ref)***. Речь в данном случае идёт о направлении передачи значений между вызывающей и вызываемой функцией. С этой точки зрения возвращаемое функцией значение является специальным безымянным выходным параметром. Исходя из механизма вызова функций в языке C++, все параметры функций являются строго входными параметрами, поскольку никакой информации об источнике значений, являющихся аргументами функции, таковая не получает и не может что либо «вернуть» в него, даже если это было бы удобно.

Однако эти термины вполне пригодны и в языке C++, если рассматривать их не технически, а по смыслу. В рассмотренном выше примере параметр `max_order_factor` можно охарактеризовать как выходной, поскольку его значение используется с единственной целью — передать информацию вызывающей функции. В этом случае значение объекта, адрес которого передаётся через такой параметр, на момент вызова функции не важно, т.к. всё равно будет перезаписано — в данном случае объект `mof` перед вызовом функции даже не инициализируется.

Рассмотрим простой пример:

```
1  void increment(int *x)
2  {
3      ++*x;
4  }
```

В этой функции осуществляется как чтение, так и запись объекта, на который указывает параметр `x`, поэтому на момент вызова функции содержимое объекта, адрес которого передан через него, должно быть определено и осмыслено. Таким образом в данном случае этот параметр можно концептуально считать двунаправленным.

7.2. Операция sizeof

Стандарт языка не определяет точных размеров большинства типов, но информация об объёме памяти, занимаемой объектом, часто оказывается необходимой. Получить её позволяет операция `sizeof`. Это одна из нескольких операций в языке C++, обозначаемая ключевым словом. Она имеет две формы:

primary-expression:
…
`sizeof` unary-expression
`sizeof (` type-id `)`

Из-за высокого приоритета этой операции выражение в первой форме обычно приходится тоже заключать в скобки, хотя эта форма используется редко.

Операция sizeof возвращает размер в байтах типа указанного его именем или соответствующего её операнду в зависимости от используемой формы. В случае вычисления размера значения выражения, оно не вычисляется, поскольку тип выражения определяется только типами входящих в него операндов и операциями. Многие программисты пишут операнд `sizeof` всегда в скобках, даже там, где он является выражением, и скобки по приоритету операций не обязательны, что, разумеется, на результат не влияет.

Тип возвращаемого значения операции `sizeof` является беззнаковым целым типом, в котором представим размер любого допустимого объекта в используемой реализации. Этот тип носит имя `std::size_t` и является псевдонимом, определённом в заголовочном файле `cstddef`. Обычно этот файл включать явно нет надобности, т.к. его включает большинство других заголовочных файлов стандартной библиотеки, включая `iostream`.

Приведём примеры использования этой операции:

```
1  // Все следующие выражения имеют истинное значение.
2
3  // Размер типа char всегда один байт по определению.
4  sizeof(char)==1;
5
6  // Платформа x86 использует представление IEC 559 для
7  // типов с плавающей точкой.
8  sizeof(double)==8;
9
10 // Выражение не вычисляется, возвращается размер
11 // типа int, определяемого по структуре выражения
12 // безотносительно конкретных значений операндов.
13 // Ошибки нет.
14 sizeof(0/0)==4;
```

7.3. Простое использование массивов

С указателями тесно связана ещё одна конструкция создания производного типа — массивы.

Массив (array) — конструкция создания производного типа данных, представлением которого в памяти является непустая последовательность расположенных подряд объектов базового типа, называемых его ***элементами (element)***. Помимо базового типа, который в случае массива называется ***типом элемента (element type)***, типы категории массив характеризуются числом элементов в них. Название такого типа читается как «массив из N объектов типа ТИП». В языке C++ элементы в массивах нумеруются с нуля, что объясняется реализацией работы с ними, которая будет рассмотрена ниже. Порядковый номер, соответствующий элементу в массиве, часто называют ***индексом (index)***.

В отличие от скалярных типов, значение которых является по смыслу неделимым, массив содержит отдельные элементы, каждый из которых является полноценным объектом. Объекты, включающие в себя другие объекты, называются ***агрегатными (aggregate)***. Объекты, являющиеся частью другого, в общем случае называются ***подобъектами (subobject)***. Таким образом, элементы массива — его подобъекты.

Над значениями типа категории массив не определено никаких операций, кроме `sizeof`, возвращающей размер всего массива, то есть размер элемента умноженный на их количество. В том числе выражения категории типа «массив» не являются модифицируемыми, даже когда леводопустимы, то есть не могут использоваться в качестве левого операнда операции присваивания. К типам категории массив применимо следующее стандартное преобразование, называемое преобразованием ***массива в указатель (array-to-pointer)***, которое и позволяет осуществлять с ними требуемые действия: ***значение категории типа «массив объектов типа ТИП» может быть преобразовано в чисто праводопустимое значение типа «указатель на ТИП», являющееся адресом нулевого элемента этого массива***. Это чрезвычайно важное свойство является одной из частей механизма, обеспечивающих основную операцию, обычно совершаемую с массивом — доступ к его элементам. Чтобы объяснить, как он работает, вернёмся к рассмотрению операций над указателями.

7.3.1. Арифметика указателей

Поскольку значения указателей являются адресами в памяти, т.е. по сути числами, к ним применимы некоторые операции, обычно используемые с арифметическими типами, которые в таком случае называются ***арифметикой указателей***, хотя указатели и не являются полноценными арифметическими типами. В этом разделе мы будем считать, что операции над несколькими указателями выполняются над указателями одного типа, т.е. на объекты одного и того же типа — приведению типов в применении к указателям будет посвящён один из следующих разделов.

Операции `==` и `!=` считают указатели «равными», если они указывают на один и тот же объект. В данном случае значения указателей сравниваются численно. Операции отношения (`<`, `>`, `<=`, `>=`) применимы к указателям, содержащим адреса объектов, являющихся элементами одного и того же массива. Согласно стандарту, в этом случае они дают результат сравнения индексов рассматриваемых элементов в массиве. В машинном коде это тоже реализовано через численное сравнение адресов, что возможно в рамках одного массива, где элементы расположены последовательно по определению, и бессмысленно, если речь идёт об адресах не связанных объектов, т.к. расположение их в памяти среды выполнения является деталью реализации — в таком случае результат этих операций формально не уточняется.

Указатель `p` также может участвовать в операциях вида `p+n`, `n+p` и `p-n`, где `n` — целочисленное значение. ***Прибавление или вычитание из указателя целого числа даёт в результате указатель на элемент того же массива, что и исходный, индекс которого отличается на данное целое значение***. Например, если `p` — указатель на третий элемент некоторого массива, то `p+2` или `2+p` — указатель на пятый элемент, а `p-3` — на нулевой. В качестве результата арифметики указателей допускаются только указатели, соответствующие индексам существующих элементов массива: в массиве на N элементов элементы имеют индексы из $[0; N)$. Помните, что из-за нумерации с нуля, в массиве на N элементов элемента с индексом N нет — это очень частая ошибка. В качестве исключения разрешается формировать (но не разыменовывать!) указатель на несуществующий N-ый элемент массива из N элементов. Это исключение введено в первую очередь потому, что на практике для обозначения подпоследовательностей элементов внутри массива часто используется пара указателей, такая что интервал, задаваемый этими указателями, замкнут слева, но не справа, и потому при указании правой границы, совпадающей с концом массива, требуется указатель на следующий после него элемент. Такое представление позволяет представить в том числе пустое множество элементов в виде полуинтервала с одинаковыми началом и концом, что невозможно для полностью закрытого интервала.

Воспользовавшись арифметикой указателей, можно добраться до нужного элемента массива. Пусть `a` — значение категории типа массив. Выражение `a+n` соответствует указателю на n-ый элемент массива: по свойству значений категории типа массив, левый операнд сначала разлагается в указатель на свой нулевой элемент, после чего применение арифметики указателей сдвигает его на необходимый элемент. Осталось разыменовать полученный указатель. Таким образом, если `a` — значение типа массив, а `n` — индекс элемента в нём, леводопустимым выражением, идентифицирующим n-ый элемент массива, является `*(a+n)`. Поскольку доступ к элементам массива используется очень часто, для его записи в языке C++ есть вспомогательная операция.

Операция индексации (subscript) `[]` имеет синтаксис `E1[E2]`, где `E1` и `E2` — выражения, и эквивалентна записи `(*((E1)+(E2)))`. Таким образом, для обращения к элементу массива `a` с индексом `n` пишут `a[n]`. (Можно написать и `n[a]`, воспользовавшись коммутативностью сложения, но делать так не принято.) Заметим, что выражение `a[0]` эквивалентно `*a`.

К арифметике указателей относят ещё одну операцию. ***Разностью двух указателей на элементы одного массива является знаковое целочисленное значение типа `std::ptrdiff_t`, равное разнице их индексов — возможны отрицательные значения.*** Этот тип также является псевдонимом, описанным в заголовочном файле `cstddef`. Вычисление разности указателей на объекты, не являющиеся элементами одного массива, ведёт к неопределённому поведению.

Хотя в тексте стандарта вся арифметика указателей определена в терминах индексов элементов массива, машинный код оперирует значениями адресов, являющихся номерами байтов памяти. Чтобы сместить указатель на заданное число элементов, смещение перед прибавлением необходимо домножить на размер элемента. При вычитании указателей разность адресов напротив требуется поделить на размер элемента, чтобы она выражалась не в байтах, а в элементах. Таким образом, базовый тип указателя не только определяет тип значения, получаемого при разыменовании, но и предоставляет необходимую информацию для операций

арифметики указателей.

7.3.2. Описание и работа с массивами

Поскольку наиболее часто к массивам применяется операция индексации, аналогичный ей синтаксис используется и в описателях объектов типов категории «массив». Вместо индекса элемента в квадратных скобках записывают общее их количество в описываемом массиве. Например,

```
int a[10];
```

— описание идентификатора `a`, соответствующего объекту типа «массив из 10 объектов типа `int`». По мнемоническому правилу сопоставления синтаксиса использования идентификатора с синтаксисом объявления, это описание можно прочесть как «если к объекту `a` применить операцию индексации с индексом не более 10, то получится значение типа `int`».

Число элементов в массиве должно задаваться преобразованным константным выражением типа `std::size_t` — поскольку он предназначен для хранения размеров объектов, их количество тоже представимо в нём. Будьте бдительны при задании размеров массивов — популярные компиляторы, которые не настроены на строгое соответствие языку, примут и не константные выражения в этом качестве, но полученный тип данных далеко не всегда ведёт себя так, как обычные массивы. Альтернативный способ создания массивов вычисляемой программой по ходу выполнения длины будет показан при рассмотрении работы с динамическим временем хранения. Пока этот способ неизвестен, будем ограничиваться небольшими объёмами обрабатываемых данных и определять в программе массивы максимального объёма, который может понадобиться в рамках наложенных нами на себя ограничений.

Продемонстрируем объявление и использование массива на примере.

```
#include <iostream>

int main()
{
    int a[10];
    // Зададим начальные значения элементов
    // массива: каждый элемент хранит
    // собственный индекс.
    for(int i=0;i<10;++i)
        a[i] = i;
    // Цикл диалога с пользователем.
    for(;;){
        // Введём индекс массива.
        std::cout << "Enter index [0-10) or -1 when done: ";
        int index;
        if(!(std::cin >> index)){
            std::cerr << "Failed to input index!\n";
            return 1;
        }
        // Ввод -1 приводит к завершению программы.
        if(index==-1)
            break;
        // Проверим индекс на допустимость.
        if(index<0||index>=10){
            std::cerr << "Index must be in [0;10)!\n";
            continue;
        }
        // Выведем старое значение и приглашение к вводу нового.
        std::cout << "a[" << index << "] = " << a[index] << "\n"
                     "Enter new value: ";
        // Введём новое значение.
        if(!(std::cin >> a[index])){
            std::cerr << "Failed to input value!\n";
            return 2;
        }
    }
    // Перед выходом из программы выведем все элементы массива.
    std::cout << "Final array content:";
```

```
39      for(int i=0;i<10;++i)
40          std::cout << ' ' << a[i];
41      std::cout << '\n';
42      return 0;
43  }
```

Эта программа позволяет в диалоговом режиме менять значения указанных элементов массива. Интересующий нас элемент массива идентифицируется выражением `a[index]`, поскольку оно является леводопустимым, оно используется как для чтения, так и для записи элемента типа `int`.

Приведём пример сеанса работы с этой программой.

```
Enter index [0-10) or -1 when done: 2↵
a[2] = 2
Enter new value: 5↵
Enter index [0-10) or -1 when done: 111↵
Index must be in [0;10)!
Enter index [0-10) or -1 when done: 8↵
a[8] = 8
Enter new value: -3↵
Enter index [0-10) or -1 when done: -1↵
Final array content: 0 1 5 3 4 5 6 7 -3 9
```

Обратим внимание, что в этой программе размерность массива 10 пришлось многократно упомянуть в тексте программы. Если его потребуется изменить, придётся искать все его вхождения, что очевидно плохо. Этот недостаток в программах на языке C исправляется с использованием макросов препроцессора, но в языке C++ есть более безопасное решение, которое будет показано далее.

Приведём примеры реализации операций работы с массивами в машинном коде на следующем примере:

```
1   void f(int n)
2   {
3       int a[10];
4       a[n+2] = 3;
5       // ...
6   }
```

Его реализация:

```
1       ; Функция f
2   f:
3       ; Пролог. Под автоматические объекты выделяется
4       ; 40 байт памяти = 10 элементов по 4 байта массива a.
5       enter 40,0
6       lea ebx,[ebp-40]           ; EBX = &a[0]
7       mov eax,[ebp+8]            ; EAX = n
8       mov dword [ebx+eax*4+8],3 ; Запись по адресу (EBX+EAX*4+8)
9                                  ; 32-битного значения 3
10      ; ...
```

В этом фрагменте используется одна из наиболее сложных форм косвенной адресации. При обращении к памяти выражение, заключённое в круглые скобки, кодируется в байты инструкции, в которой оно содержится, при этом допустимыми являются далеко не все возможные выражения, а только те, которые могут быть вычислены блоком адресации центрального процессора — более сложные вычисления следует выполнять отдельными инструкциями непосредственно над значением адреса. Архитектура x86 в 32-битном варианте поддерживает выражения, являющиеся суммой одного или нескольких слагаемых из следующего списка (каждое слагаемое не более одного раза):

- Значение одного из 8 основных регистров.
- Значение одного из 8 основных регистров, кроме `ESP`, умноженное на 1, 2, 4 или 8.
- Явно заданная числовая величина, называемая ***смещением (displacement)*** (может быть отрицательной).

Например, в предыдущих примерах, где использовалось обращение по явно заданным адресам памяти, выражение адреса состояло только из смещения, а в этом примере присутствуют все три компонента:

- Значение регистра `EBX`, в который предварительно загружен адрес нулевого элемента массива — он известен транслятору, поскольку он сам выделил место на стеке и распределил его между автоматическими объектами.
- Значение регистра `EAX`, в который загружено значение параметра `n`, переданного в функцию, домноженное на размер элемента (4).
- Смещение 8 — это дополнительное смещение на два 4-байтных элемента, соответствующее прибавлению 2 к n в индексе массива.

Доступ на чтение осуществляется аналогично. Таким образом, разложение массива в указатель на его нулевой элемент, сама нумерация этих элементов с нуля и определение операции индексации в языке C++ вытекают из реализации этих операций в машинном коде.

Обсудив описание и использование массивов, автор хочет дать пояснение, почему пошёл столь длинным путём в объяснении работы операции индексации, вместо простого определения «операция индексации осуществляет доступ к элементу массива по индексу». Во-первых, использованное автором определение в точности соответствует стандарту — единственное упоминание этой операции в нём содержит её определение в виде эквивалентного выражения с использованием арифметики указателей и разыменования. Во-вторых, возможно вполне корректная формулировка применительно к некоторым другим языкам программирования оказывается некорректной в языке C++. Операция сложения, присутствующая в раскрытии операции индексации, требует либо двух арифметических операндов, либо целого числа и указателя. Поскольку индекс, обычно указываемый в квадратных скобках, является целым числом, выражение перед ними должно быть указателем, но тем не менее там часто указывают массив. Возможность такого указания обеспечивается свойством значений категории «массив» разлагаться в указатель на свой нулевой элемент в большинстве контекстов выражения. Если рассматривать операцию индексации только как доступ к элементу массива, невозможным кажется использование в ней отрицательных индексов, однако ничего странного в этом нет:

```
1  // Массив на 10 элементов.
2  int a[10];
3
4  // p указывает на четвёртый элемент.
5  int *p = a+4;
6
7  // Модификация элемента с индексом 4-2==2.
8  p[-2] = 42;
```

Преобразование массива в указатель может быть выполнено явно операцией `static_cast`:

```
1  int a[5] = {};
2  int *p = static_cast<int*>(a);
```

В большинстве случаев этот явный синтаксис не нужен, т.к. преобразования проходит неявно там, где требуется.

В отладчике Qt Creator при раскрытии узла, соответствующего объекту типа категории «массив», внутри будут показаны подузлы, соответствующие его элементам. В качестве значения узла самого массива обычно отображается его адрес, т.к. ничего другого о всём массиве в целом, что не описывается его типом, сказать нельзя.

7.3.3. Инициализаторы массивов

Массивы, как и другие объекты, могут иметь в определении инициализаторы. При инициализации агрегатов, включая массивы, можно применять все виды инициализации списком из требуемого числа элементов:

```
int x[3] = {1,2,3};
```

Указанные значения используются для инициализации копированием подобъектов инициализируемого объекта по порядку их расположения в памяти, то есть для массива — по порядку индексов элементов, начиная с 0. Если элементов списка инициализаторов меньше, чем подобъектов, оставшиеся подобъекты инициализируются по значению — инициализация пустым списком способна занулять агрегаты любого размера. Лишние элементы в списке инициализаторов не допускаются.

Этот синтаксис показывает, что скаляры в языке во многом трактуются как массивы из одного элемента. В то же время типы категории «массив» начинают показывать разницу между инициализацией и присваиванием:

```
1  // Инициализация.
2  // После последнего элемента разрешена лишняя запятая для общности.
3  int a[2] = {
4      1,
5      2,
6  };
7
8  // ОШИБКА: массив не является модифицируемым леводопустимым значением.
9  a = {1,2};
```

В определении массива, имеющего инициализатор, размер можно опустить, тогда он будет вычислен автоматически по количеству элементов списка инициализаторов:

```
1  // Массив из 3 элементов.
2  int array[] = {1,2,3};
3
4  // ОШИБКА: попытка определить массив на 0 элементов.
5  //         Принимается некоторыми не настроенными компиляторами
6  //         в качестве расширения.
7  int array[] = {};
```

Эта форма удобна в некоторых случаях, когда число инициализаторов велико или часто меняется (и всегда короче), но порождает проблему — если мы не стали указывать размер в определении массива, как вычислить его в остальных местах, где он понадобится? Ведь если в других местах потребуется задавать его явно, то теряется то преимущество, что при изменении длины инициализатора не нужно корректировать значение длины.

Число элементов массива может быть вычислено применением операции `sizeof` к массиву и его элементу:

```
1  int a[] = {1,5,87,3,7,54,4};
2
3  // Число элементов есть размер всего массива, делённый на размер элемента,
4  // например, нулевого, который есть всегда.
5  for(std::size_t i=0;i<sizeof a/sizeof *a;++i)
6      std::cout << a[i] << '\n';
```

Данный пример работает с массивом элементов, ни разу не упоминая их число — инициализатор можно менять произвольным образом и программа будет выводить значения всех элементов массива. В приведённой формуле размер массива в байтах делится на длину нулевого элемента в байтах, чтобы получить само число элементов. Подвыражение `sizeof *x` внешне похоже на умножение, но нужно помнить, что и `sizeof`, и * в данном случае — унарные префиксные операции.

То, что тип этого выражения тоже имеет тип `std::size_t`, подчёркивает, что это требуемый тип для хранения количеств объектов. Если вы будете по привычке использовать тип `int` для счёта объектов вместе со значениями типа `size_t` вы столкнётесь с двумя проблемами:

- Тип `int` на многих, например большинстве 64-битных платформах остаётся 32-битным, а `std::size_t` имеет размер, совпадающий с разрядностью архитектуры. В таком случае в программе возникнут проблемы при обработки очень больших объёмов данных, когда объект типа `int` не сможет хранить необходимые значения.
- `int` — знаковый тип, а `std::size_t` — нет, поэтому возможны соответствующие проблемы при применении операций отношения, если вы будете смешивать использование этих двух типов.

В простых случаях, когда число объектов мало и фиксировано, допустимо обходиться и фиксированным фундаментальным типом, но следует заранее привыкать к использованию `std::size_t`, что будет особенно важным при ручном управлении выделением произвольных объёмов памяти.

7.3.4. Массивы и функции

Основное свойство разложения массива имеет особые эффекты при использовании массивов вместе с функциями. Технически сами значения массивов, т.е. значения всех их элементов, никогда не передаются в или из функции, поскольку могут иметь произвольно большой размер, что не соответствует философии языка, стремящегося к максимальной производительности.

Возвращаемыми значениями функций массивы и вовсе быть не могут. Вместо этого можно вернуть указатель, но будьте внимательны: попытка возвращения адреса объекта с автоматическим временем хранения из функции неминуемо приведёт к тому, что при выходе из неё объект будет уничтожен, и возвращённое значение будет «висячим» указателем, который использовать нельзя:

```
1  int* f()
2  {
3      int a[10];
4      // ОШИБКА: возвращение адреса объекта
5      // с автоматическим временем хранения.
6      return a;
7  }
8
9  int* g()
10 {
11     static int b[10];
12     // Ошибки нет, b существует всё время
13     // выполнения программы и теперь его
14     // элементы могут быть изменены вне
15     // функции g, являющейся его областью
16     // видимости по возвращённому указателю.
17     return b;
18 }
```

Рассмотрим теперь, что произойдёт, если использовать массивы в качестве типов параметров функции. Все попытки определить параметр функции категории «массив» приводят к тому, что вместо этого определяется параметр типа, в который такой массив разлагался бы, т.е. типа указателя на базовый тип массива. Поскольку вторая характеристика массива — размер — в этом случае будет потеряна, указанный размер массива не играет роли и может быть опущен. Поясним на примерах:

```
1  // Описание функции, настоящий тип параметра --- int *
2  void f(int x[10]);
3
4  // Совместимое и потому допустимое после предыдущего
5  // описание функции --- тип параметра тот же, int *
6  void f(int x[20]);
7
8  // Ещё одно совместимое описание --- число элементов не важно.
9  void f(int x[]);
10
11 // Последнее совместимое описание, отражающее реальный
12 // тип параметра функции:
13 void f(int *x);
```

В этом примере показано, что не только идентичные описания могут быть совместимыми — в данном случае это просто разные формы записи одного и того же типа параметра.

Поскольку операция индексации применима в первую очередь именно к указателям, передача в функцию и работа с такими параметрами выглядит так, словно это действительно массивы:

```
1  void make_double(int x[])
2  {
```

```
3       for(int i=0;i<5;++i)
4           x[i] *= 2;
5   }
6
7   void f()
8   {
9       int array[] = {1,2,3,4,5};
10      make_double(array);
11      // Выводит 2 4 6 8 10
12      for(int i=0;i<5;++i)
13          std::cout << array[i] << ' ';
14      std::cout << '\n';
15  }
```

Но ни в коем случае не впадайте в это заблуждение! Вспомните, что настоящий тип параметра функции `make_double` — `int *`. Когда имя, соответствующее объекту типа массив из 5 объектов типа `int`, указывается в операции вызова функции, его значение типа категории «массив» разлагается в указатель на нулевой элемент этого массива. Теперь оно имеет тип `int *`, что совпадает с действительным типом параметра функции. С точки зрения функции, ей передаётся указатель на *некоторый* элемент некоторого массива — является ли он действительно нулевым с точки зрения функции определить невозможно. Информация об общем числе элементов в этом массиве функции тоже не доступна. Всё, что она может — обращаться к элементам массива, указатель на один из элементов которого ей передан, относительно него. В нашем случае она удваивает значение элемента, указатель на который ей передан, и четырёх следующих, к которым обращается за счёт использования арифметики указателей в операции индексации. Покажем, что её можно использовать для удвоения любой последовательности из 5 элементов типа `int`:

```
17  // Продолжение прошлого фрагмента
18
19  void g()
20  {
21      int array[10];
22      // Заполнить массив единицами.
23      for(int i=0;i<10;++i)
24          array[i] = 1;
25      // Удвоить элементы с индексами [0;0+5)
26      make_double(array); // array эквивалентно array+0
27      // Удвоить элементы с индексами [3;3+5)
28      make_double(array+3);
29      // Выводит 2 2 2 4 4 2 2 2 1 1
30      for(int i=0;i<10;++i)
31          std::cout << array[i] << ' ';
32      std::cout << '\n';
33  }
```

Число элементов, которые необходимо обработать, можно передать в функцию отдельным параметром, тем самым она сможет работать с последовательностями элементов любой длины:

```
1   int sum(int x[],std::size_t n)
2   {
3       int s = 0;
4       for(std::size_t i=0;i<n;++i)
5           s += x[i];
6       return s;
7   }
8
9   void f()
10  {
11      int x[] = {3,7,1,8,4,2,0,5,6,9};
12      // Сумма значений с индексами [3;3+3) --- 14.
13      std::cout << sum(x+3,3) << '\n'
14      // Сумма значений с индексами [5;5+4) --- 13.
15                << sum(x+5,4) << '\n'
16      // Сумма значений с индексами [0;0+10) (всего массива) --- 45.
17                << sum(x,10) << '\n';
18  }
```

Покажем ещё один пример того, что любой объект может рассматриваться как массив из одного элемента соответствующего типа:

```
20 // Продолжение прошлого фрагмента
21
22 void g()
23 {
24     // Сумма из одного слагаемого:
25     int x[] = {42};
26     std::cout << sum(x,1) << '\n'
27     // То же самое для скалярного объекта:
28     int y = 42;
29     std::cout << sum(&y,1) << '\n';
30 }
```

С помощью переданного функции указателя можно совершать операции как чтения, так и записи, поэтому, если рассматривать такую запись как «передачу массива» (или его части), т.е. последовательности элементов, то можно считать такой параметр потенциально двунаправленным — вызываемая функция может как прочесть значения этих элементов, которые могла записать вызывающая функция, так и изменить их, чтобы они были прочитаны после возврата. Из рассмотренных нами примеров функция `make_double` использует параметр как двунаправленный, а `sum` — как входной. Приведём пример использования указателя для записи функцией последовательности значений, доступных вызывающей, т.е. в роли out-параметра:

```
1  bool read_ints(int x[],std::size_t n)
2  {
3      for(std::size_t i=0;i<n;++i)
4          if(!(std::cin >> x[i]))
5              return false;
6      return true;
7  }
8
9  bool f()
10 {
11     int a[3],b[5];
12     std::cout << "Enter 3 integers for a: ";
13     if(!read_ints(a,3)){
14         std::cerr << "Input failed!\n";
15         return false;
16     }
17     std::cout << "Enter 5 integers for b: ";
18     if(!read_ints(b,5)){
19         std::cerr << "Input failed!\n";
20         return false;
21     }
22     // Вычисления с использованием a и b...
23     // ...
24     return true;
25 }
```

В этом примере показана функция ввода последовательности целых чисел заданной длины и её использование. Функция завершает работу и возвращает `false` при любой ошибке ввода, так что вызывающая может проверить успешность её работы. Видно, что массивы `a` и `b`, адреса элементов которых передаются функции для заполнения даже не инициализируются — в данном случае функция записывает в них результат своей работы, т.е. неформально «возвращает массив».

Для задания диапазона элементов помимо представления в виде указателя на начальный элемент и длины, можно использовать представление в виде двух указателей `begin` и `end`, которые задают диапазон вида $[begin; end)$, о котором уже говорилось. Перепишем предпоследний пример в таком виде:

```
1  int sum(int* begin,int* end)
2  {
3      int s = 0;
```

```
4       while(begin!=end)
5           s += *begin++;
6       return s;
7   }
8
9       int x[] = {3,7,1,8,4,2,0,5,6,9};
10      // Сумма значений с индексами [3;6) --- 14.
11      std::cout << sum(x+3,x+6) << '\n'
12      // Сумма значений с индексами [5;9) --- 13.
13                << sum(x+5,x+9) << '\n'
14      // Сумма значений с индексами [0;10) (всего массива) --- 45.
15                << sum(x,x+10) << '\n';
16
17  void g()
18  {
19      // Сумма из одного слагаемого:
20      int x[] = {42};
21      std::cout << sum(x,x+1) << '\n'
22      // То же самое для скалярного объекта:
23      int y = 42;
24                << sum(&y,&y+1) << '\n';
25  }
```

Для того, чтобы в этой форме можно было представить интервал, включающий последний элемент массива, и разрешено формирование указателя на элемент, следующий за последним в нём. В отличие от симметричной формы интервала, закрытого с обоих сторон $[begin; end]$ данная форма также позволяет представить пустой интервал (`begin==end`), кроме того вычисление числа элементов в таком диапазоне требует одной операции вычитания без прибавления единицы, что упрощает вычисления.

К использованию формы с конструкцией создания производного типа категории «массив» вместо действительно типа-указателя в параметрах функций есть два подхода. Первый предполагает никогда не использовать форму, не соответствующую сути происходящего и всегда использовать указатель. Второй заключается в использовании формы массива, когда речь идет о передачи адреса последовательности объектов в памяти, и формы указателя, когда речь идёт об адресе единственного объекта. Вторая форма несколько более выразительна, но на практике мало где закрепилась. Автор, тем не менее, будет придерживаться последнего стилистического варианта.

7.4. Комбинирование конструкций создания производных типов

В общем случае, в качестве базового типа для конструкции создания производного типа может выступать другая конструкция. С помощью такого комбинирования можно получить различные более сложные типы данных. Не все комбинации являются допустимыми:

- Как уже обсуждалось, массив не может быть возвращаемым значением функции.
- Тип категория «функция» не может быть базовым для конструкций создания производного типа «функция» или «массив», поскольку не является типом ***данных*** (функция возвращает данные, а массив — последовательность данных в памяти). Указатели на функции допустимы, и будут рассмотрены в разделе 7.5.
- Все остальные комбинации допустимы.

Для создания таких сложных типов нужно последовательно применить в описателе необходимые конструкции. Последовательность применения конструкций соответствует приоритетам операций разыменовывания, вызова функции и индексации, синтаксически схожие с которыми конструкции обозначают эти формы частей описателей. Для изменения этого порядка можно, как и в выражениях, применять круглые скобки. Приведём примеры всех 9 возможных вариантов, включая недопустимые — всё это различные и осмысленные (в допустимых случаях) типы:

```
1   // Указатель на указатель на int.
2   int **pp;
3
```

```
4  // Указатель на массив из 10 подобъектов типа int.
5  // Скобки нужны, поскольку приоритет индексации выше разыменования.
6  int (*pa)[10];
7
8  // Указатель на функцию без параметров, возвращающую int.
9  // Скобки нужны, поскольку приоритет вызова функции выше разыменования.
10 int (*pf)();
11
12 // Массив из 10 указателей на объекты типа int.
13 int *ap[10];
14
15 // Массив из 10 массивов из 20 объектов типа int каждый.
16 int aa[10][20];
17
18 // ОШИБКА: массив функций недопустим.
19 int af[10]();
20
21 // Функция без параметров, возвращающая указатель на int.
22 int *fp();
23
24 // ОШИБКА: функция не может возвращать массив.
25 int fm()[10];
26
27 // ОШИБКА: функция не может возвращать функцию.
28 int ff()();
```

На одном уровне вложенности конструкций создания производных типов можно не останавливаться и создавать типы любой сложности. В реальных программах, разумеется, созданием чрезмерно сложных типов из любви к искусству не занимаются, напротив — во многих случаях для упрощения чтения сложного вложенного типа, который действительно потребовался при решении задачи, его разбивают на несколько описаний с помощью псевдонимов типов. Попробуем прочесть сложный тип, содержащий псевдонимы типов.

```
using fptr =  int (*)(double);
fptr (*x[10])(fptr);
```

Вначале следует выделить тип, соответствующий всем описателем в описании и каждый описатель отдельно. В этом случае описателю `(*x[10])(fptr)` соответствует тип `fptr`. Идентификатор, описываемый этим единственным описателем — `x`, тип, соответствующий ему, мы и выясняем. Так же, как и в более простых случаях, воспользуемся мнемоническим правилом «начиная с идентификатора, применение к нему операций, синтаксически соответствующих конструкциям создания производного типа, даёт тип описания». Следуя этому правилу прочтём, что «если к `x` применить индексацию, затем разыменование, затем вызов функции, то получится `fptr`». Заменим теперь применяемые действия смыслом конструкций: то, что разыменовывается — указатель, индексируется — массив, а вызывается — функция: `x` — массив из 10 указателей на функции, принимающих и возвращающих `fptr`.

Теперь можно раскрыть псевдоним типа, прочтя его аналогично. В сложном имени типа бывает непросто найти место, откуда был удалён идентификатор, чтобы начать оттуда чтение, но если вы видите странную, на первый взгляд, конструкцию `(*)`, то единственный корректный вариант — идентификатор был между звездой и закрывающей скобкой. Таким образом, это псевдоним типа указателя на функцию, принимающую `double` и возвращающую `int`. Итого: `x` — массив из 10 указателей на функции, принимающих и возвращающих указатель на функцию с одним параметром типа `double` и возвращающую `int`.

Без использования псевдонимов это описание записывается совсем ужасно:

```
int (*(*x[10])(int (*)(double)))(double);
```

В данном случае выполнять коррекцию типов категории «массив», использованных в качестве параметров функции, до указателей делать не пришлось, но при возникновении такой ситуации сделать это при прочтении типа также придётся. Навык чтения сложных описаний вырабатывается так же, как и умение определить в математическом выражении последовательность операций и их операндов.

При работе в отладчике, дочерние элементы, соответствующие производным типам, могут быть раскрыты на любую требуемую глубину, образуя дерево.

7.4.1. Массивы массивов

Одним из наиболее часто используемых случаев вложенного использования конструкций создания производного типа являются произведения массивов от массивов. Хотя ничем более с точки зрения языка C++ эта конструкция не является, её часто называют неформально «многомерным массивом». Частота её использования обусловлена тем, что «двумерный» массив — массив массивов — является удобным представлением различных матриц и таблиц. Трёхмерные (массив массивов массивов) и более сложные конструкции также применяются, но значительно реже.

Рассмотрим устройство многомерного массива на примере

```
int a[2][4];
```

Поскольку массив по определению — непрерывная последовательность элементов в памяти, то массив, элементами которого является другой массив, есть одна непрерывная последовательность однотипных элементов. Таким образом, приведённое выше определение соответствует восьми объектам типа `int`, расположенным подряд, так же как и в определении

```
int b[8];
```

Что неудивительно — элементы любой матрицы можно перенумеровать одним индексом, но использование двух для нумерации по строкам и столбцам удобнее. Отметим, что с точки зрения языка C++ понятий «строка» и «столбец» не существует — это лишь наше соглашение об интерпретации порядка элементов в линейном адресном пространстве: мы можем считать этот объект матрицей из 2 строк и 4 столбцов, а можем считать, что в нём 4 столбца и 2 строки, при этом меняется порядок указания индексов к двум операциям индексации, которые понадобятся для доступа к отдельному элементу типа `int` в этом сложном агрегатном объекте:

```
// Может интерпретироваться как:
// - элемент строки 1 столбца 2
//   матрицы из 2 строк и 4 столбцов.
// - элемент столбца 1 строки 2
//   матрицы из 2 столбцов и 4 строк.
a[1][2] = 42;
```

Поскольку физически в памяти расположены подряд элементы последней конструкции создания производного типа «массив», то первый из этих способов интерпретации соответствует случаю, когда элементы одной строки расположены в памяти непосредственно друг за другом, а индексы пишутся так, как принято в математике: сначала строка, потом столбец. Такой порядок хранения элементов называют ***построчным (row-major)***, и в языке C++ принято пользоваться именно им. Обратный способ интерпретации (которым в языке C++ пользоваться ничуть не сложнее, но обычно не принято, если нет принципиальной разницы) называется ***постолбцовым (column-major)*** и более широко применяется в некоторых других языках программирования.

Обратим внимание на возможную ошибку, характерную особенно для знакомых с языком `C#`:

```
1  int c[2,4];
2  c[1,2] = 42;
```

Это синтаксически корректный код на языке C++, что, однако, является причиной недоразумений. Вспомним, что операция «запятая» в языке C вычисляет свой левый операнд, а затем правый и возвращает его, потому эта запись эквивалентна

```
1  int c[4];
2  c[2] = 42;
```

где, на самом деле, речь идёт всего лишь об одномерном массиве — левые операнды не имеют побочных эффектов и могут быть удалены из программы без изменения её смысла.

Итак, ***массив массивов является непрерывной линейной последовательностью объектов, структура типа которого позволяет использовать последовательность операций индексации для доступа к определённому элементу по его индексам в каждом «измерении»***. Покажем представление определённого выше массива `a` с помощью рис. 7.1.

Рассмотрим вычисление выражения `a[1][2]` для изображённого на рисунке случая, когда объект `a` расположен по адресу `0x7f000000`. Будем использовать ту же самую запись с операциями индексации, чтобы именовать элементы, о которых идёт речь.

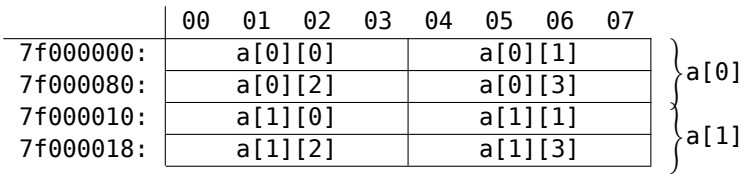


Рис. 7.1: Представление объекта `int a[2][4]`.

1. По определению операции индексации имеем `a[1][2] == *(*(a+1)+2)`.
2. Подвыражение `a` типа `int [2][4]` разлагается в указатель на свой первый элемент `a[0]` типа `int [4]`. Это значение имеет тип `int (*)[4]` (указатель на массив из 4 элементов типа `int`) и численно совпадает с адресом самого объекта `a`: `0x7f000000`.
3. Операция сложения этого указателя с числом 1 приводит к смещению этого указателя на 1 элемент того типа, на который он указывает, результатом является указатель того же типа `int (*)[4]` на объект `a[1]`, численно равный `0x7f000010`. Это значение получено путём прибавления к значению указателя смещения, умноженного на размер объекта, на который указывает указатель: `sizeof(int [4]) == sizeof(int)*4 == 4*4 == 0x10` (для архитектуры x86).
4. К последнему промежуточному значению применяется операция разыменования, результат — леводопустимое выражение, идентифицирующее объект `a[1]` типа `int [4]`.
5. Это значение опять преобразуется в указатель на нулевой элемент по требованию операции сложения, в которой оно является операндом. В результате получается значение типа `int *`, являющееся адресом объекта `a[1][0]` — `0x7f000010`.
6. Прибавление к нему числа 2 сдвигает указатель на объект `a[1][2]`.
7. Наконец, операция разыменования даёт леводопустимое значение, соответствующее объекту `a[1][2]` типа `int`.

Убедитесь, что вам понятна эта последовательность смены значений и типов! Покажем для сравнения значения и типы некоторых выражений, связанных с подобъектами объекта `a`, в таблице 7.1.

Выражения	Тип	Значение
`a`	`int [2][4]`	(весь массив массивов)
`a+0`	`int (*)[4]`	`0x7f000000`
`&a`	`int (*)[2][4]`	`0x7f000000`
`*a`,`a[0]`	`int *`	`0x7f000000`
`a+1`	`int (*)[4]`	`0x7f000010`
`&a+1`	`int (*)[2][4]`	`0x7f000020`
`*a+1`,`a[0]+1`	`int *`	`0x7f000004`

Таблица 7.1: Выражение, значения и типы для подобъектов массива массивов

Из таблицы видно, что численные значения адресов, соответствующие значениям некоторых выражений совпадают, хотя они и имеют разные типы. Дальнейшее прибавление к ним единицы даёт различные адреса, поскольку арифметика указателей оперирует индексами элементов, а соответствующие изменения адресов зависят также от размера элементов. Таким образом, мы ещё раз убедились, что базовый тип категории произведения типов «указатель» не просто определяет тип разыменованного объекта, но и определяет его размер, что необходимо для применения арифметики указателей, без которой работать с массивами невозможно.

Теперь поговорим об инициализации вложенных агрегатных объектов. Правило записи их инициализаторов таково: ***если инициализатор подобъекта в списке инициализации является списком инициализации, то он используется для задания содержимого объекта, иначе для инициализации этого подобъекта берутся элементы текущего списка инициализации, возвращаясь к подобъектам исходного объекта, если их достаточно***. Таким образом, с каждым списком инициализации связан свой инициализируемый объект. Поясним это правило на примерах:

```
1 // Один список инициализации для двух подобъектов объекта a:
2 int a[2][3] = {1,2,3,4,5,6};
```

```
3
4  // Аналогично, но каждый подобъект инициализируется
5  // своим списком инициализации:
6  int b[2][3] = {{1,2,3},{4,5,6}};
7
8  // Частичная инициализация:
9  int c[3][3] = {
10     {1}, // Инициализатор подобъекта c[0],
11          // c[0][0] = 1 явно, c[0][1] = c[0][2] = 0 неявно по значению.
12     {    // Инициализатор подобъекта c[1]
13         2, // c[1][0] = 2
14         3  // c[1][1] = 3
15     }      // c[1][2] = 0 (невявно)
16 }; // c[3][0] = c[3][1] = c[3][2] = 0 (неявно)
```

Первый пример использует минимальное необходимое число списков инициализации для агрегата любой глубины вложенности — один. Второй пример содержит явный список инициализации для каждого агрегата на всех уровнях вложенности — эта форма инициализатор называется ***полной скобочной (completely braced)***. Последний пример показывает, что нулевые значения получают все подобъекты на всех уровнях вложенности, не получившие явного инициализатора.

Как было показано ранее, функция легко может работать с внешними относительно неё массивами произвольного размера. Рассмотрим проблемы, которые возникают при попытке повторить этот приём с массивами массивов (более «многомерные» массивы ведут себя аналогично).

Попробуем описать функцию с параметром типа «массив массивов»:

```
void f(int x[10][20]);
```

Поскольку тип параметра функции имеет категорию «массив», он разлагается в указатель на свой базовый тип: это описание эквивалентно

```
void f(int (*x)[20]);
```

Обратите внимание, что последующего разложения не происходит — получившийся после первого разложения тип «указатель на массив из 20 объектов типа `int`» имеет категорию указатель и никаким преобразованиям, характерным для массивов, не подвержен. Таким образом, при передаче в функцию массива массивов варьироваться во время исполнения программы (и игнорироваться при задании в описании) будет только первое измерение. Объясняется это уже известными нам свойствами. Указатель на элемент самого «старшего» измерения, который по факту передаётся в функцию, позволяет осуществить доступ к произвольному числу элементов, но сама арифметика указателей с ним требует знания размера этого самого элемента, который сам является массивом, поэтому его размер существенен и входит в действительный тип параметра.

Решить проблему передачи «многомерных» массивов в функцию можно несколькими способами. Пока нам доступен только один, и не лучший — поступить аналогично одномерному случаю: задать размеры, исходя из естественных или искусственных ограничений задачи, и использовать только необходимую часть.

7.5. Указатели на функции

Указатель на функцию является допустимой конструкцией создания производного типа данных, несмотря на то, что с точки зрения языка C++ типы категории «функция» не считаются данными. Выражение, имеющее тип категории «функция», обычно является именем сущности соответствующего типа, т.е. соответствующим функции в результате её описания. К таким выражениям применимо неявное преобразование ***функции в указатель (function-to-pointer)***: значение типа категории функция может быть неявно преобразовано к типу указателя на такую функцию. Об этом преобразовании, как и для массива, также неформально говорят, как о разложении (decay). Такое значение уже имеет тип данных и является сутью реализации механизма идентификации функций — хранимое в таком указателе значение соответствует адресу начала кода функции в памяти. Такое разложение может быть затребовано явно операцией `static_cast` либо операцией взятия адреса, что обычно не требуется:

```
int f(int x)
{
    return x;
}

// Неявное разложение типа функции до указателя на функцию.
int (*pf)(int) = f;

// То же самое явно двумя способами.
int (*pf2)(int) = &f;
int (*pf3)(int) = static_cast<int (*)(int)>(f);

int a = pf2(42);
```

Давно рассмотренная нами операция вызова функции может в качестве первого операнда принимать значение не только типа категории «функция», но и «указатель на функцию», как показано в последней строке примера.

Так же, как и параметры функции типа категории «массив», параметры типа категории «функция» разлагаются в соответствующие указатели:

```
// Функция f не возвращает значений и имеет
// один параметр типа <<функция без параметров
// и возвращаемого значения>>.
void f(void ());

// Действительный тип приведённого выше параметра,
// отражающий его суть, - <<указатель на функцию
// без параметров и возвращаемого значения>>, что
// отражено в следующем совместимом описании:
void f(void (*)());
```

К указателям на функции арифметика указателей не применима, поскольку адреса инструкций машинного кода кроме тех, которые соответствуют началам функций и возникающие в программе в результате использования показанных выше конструкций, не являются «функциями», в том смысле, что не обеспечивают соблюдение протокола вызова функций, закреплённого в соглашениях о вызовах. Разыменовывать указатели на функции можно, но бессмысленно, поскольку они тут же будут разложены назад в указатели.

Приведём пример использования указателей на функции:

```
#include <iostream>

namespace
{
    void print_table(double from,double to,double step,
                     double (*f)(double))
    {
        for(double x=from;x<=to;x+=step)
            std::cout << x << " : " << f(x) << '\n';
        std::cout << '\n';
    }

    double f1(double x)
    {
        return x*x;
    }

    double f2(double x)
    {
        return 1.8+x*(2.5-x*(3.4+0.2*x));
    }
}

int main()
{
    std::cout << "f1(x):\n";
    print_table(0.,5.,.5,f1);
```

```
28      std::cout << "f2(x):\n";
29      print_table(0.,10.,0.1,f2);
30      return 0;
31  }
```

Данная программа содержит функцию `print_table`, печатающую таблицу значений любой функции, принимающей и возвращающей `double`.

Указатели на функции являются механизмом реализации множества интересных конструкций, с которыми мы познакомимся позднее.

7.6. Квалификаторы типов

Нам известно, что в понятие спецификаторов описания входят спецификаторы типа, задающие конкретный тип, о котором идёт речь в описании. Познакомимся с понятием ***квалификаторов типов (type qualifiers)***, которые позволяют модифицировать свойства операций чтения и записи, применительно к идентифицируемым выражениями объектам. Квалификаторы типа относятся к спецификаторам описаний. Если один и тот же квалификатор оказался среди спецификаторов описания несколько раз (явно или в результате использования псевдонимов типов), то повторные указания допускаются и игнорируются. ***Отметим, что квалификаторы типов применимы только к выражениям, идентифицирующим объекты, когда в результате преобразований это свойство теряется, вместе с ним пропадают и все квалификаторы.*** Квалификаторы не применимы к самим типам категории «функция», хотя и допустимы для их параметров и возвращаемого значения.

Наиболее часто используемым квалификатором типа является `const`. Леводопустимые выражения называются ***модифицируемыми (modifiable)***, если они ***не*** квалифицированы как `const` и не являются массивами. Операции присваивания, инкремента и декремента требуют в качестве своих операндов, идентифицирующих подлежащие записи объекты, не просто леводопустимые, а ***модифицируемые*** леводопустимые значения. Таким образом, значения `const`-квалифицированных объектов не могут меняться после их создания, им только можно задать начальное значение с помощью явной инициализации, которая в их определениях обязательна.

Кандидатами на `const`-квалифицирование могут быть, например, константы, чтобы компилятор смог обнаружить некорректные по смыслу попытки их изменить. Использование неизменяемых объектов в областях видимости пространств имён обычно не вносит проблем, связанных с модульной структурой программы и часто применяется для задания глобальных констант.

Подчеркнём, что когда леводопустимое выражение заменяется значением объекта, квалификаторы теряются:

```
1  // x - не модифицируемое леводопустимое выражение.
2  const int x = 5;
3
4  // В следующем выражении леводопустимое выражение x,
5  // идентифицирующее некоторый объект, заменяется вначале
6  // значением 5 типа просто int - без const. Результат
7  // сложения - не леводопустимое выражение со значением 6 типа int.
8  x+1;
```

Квалификатор const может быть применён не только к самому типу среди спецификаторов описания, но и к каждому произведению типа «указатель», путём указания его справа от соответствующего символа *, что соответствует его применению к тому значению, которому соответствует описатель на момент прочтения такого описателя изнутри:

```
1  int a = 10,b = 20;
2
3  // Указатель на int.
4  int *pi = &a;
5  *pi = 30; // Можно.
6  pi = &b;  // Можно.
7
8  // Указатель на немодифицируемый int.
9  const int *pci = &a;
10 *pci = 30; // ОШИБКА: *pci имеет тип const int - не модифицируемый.
```

```
11 pci = &b;  // Можно.
12 // Тот же самый тип, порядок спецификаторов описания не имеет значения.
13 int const *pci2;
14
15 // Немодифицируемый указатель на int.
16 int * const cpi = &a;
17 *cpi = 30;  // Можно.
18 cpi = &b;   // ОШИБКА: cpi имеет не модифицируемый тип.
19
20 // Немодифицируемый указатель на немодифицируемый int.
21 const int * const cpci = &a;
22 *cpci = 30; // ОШИБКА.
23 cpci = &b;  // ОШИБКА.
24 // Тот же самый тип, порядок спецификаторов не имеет значения.
25 int const * const cpci2;
```

Этот пример можно продолжить и на большее число произведений типов, например тип `int * const *` — это модифицируемый указатель на немодифицируемый указатель на модифицируемый `int`, и т.д. Квалифицирование указателя на каждом уровне возможно потому, что каждая операция в цепочке разыменований, необходимых для доступа к самому значению типа из описания, кроме последней, осуществляющей доступ к нему самому, является операцией чтения адреса, по которому производится следующее обращение. Поскольку эти значения адресов в памяти сами являются объектами типа категории «указатель», они леводопустимы и, следовательно, квалифицируемы.

> Другой квалификатор, который можно применить как к самому типу, так и к произведениям типа указатель — это `volatile`. Он означает, что это значение может меняться в неизвестные реализации моменты времени и/или иметь дополнительные нестандартные побочные эффекты. Таковыми могут быть, например, объекты, которые в памяти разделяют несколько выполняющихся одновременно программ или частей одной программы, либо отображённые в память аппаратные регистры — чтение/запись такой памяти на самом деле осуществляет взаимодействие с некоторым устройством. Транслятор обязан осуществлять все операции доступа к таким объектам строго в соответствие с правилами поведения абстрактной машины, описываемой стандартом языка, без каких-либо оптимизаций, даже если с его точки зрения они допустимы. Любой доступ к `volatile`-квалифицированному объекту через леводопустимое выражение, таковым не являющееся, ведёт к неопределённому поведению. В этой книге мы не встретимся с примерами таких объектов.

Описания со связанностью, квалифицированные `const`, но не `volatile`, без спецификатора `extern` и предыдущего связанного описания с внешней связанностью, имеют по умолчанию внутреннюю связанность:

```
1 // Обычные неизменяемые объекты со связанностью принимают
2 // внутреннюю её форму, если не указано обратного.
3 const double pi = 3.1415926;
```

Это позволяет включать такие определения в заголовочные файлы без возникновения ошибок множественных определений с внешней связанностью. Хотя умный компоновщик может объединять копии таких объектов, определяемые при каждом включении подобного заголовочного файла в единицы трансляции программы, надеяться на это и злоупотреблять такими определениями особенно больших объектов не следует.

Допустимы неявные преобразования между чисто праводопустимыми значениями категории типа «указатель» на по-разному квалифицированные объекты, называемые ***преобразованиями квалификаторов (qualification conversions)***. В цепочке имени типа

```
T cv1 * cv2 * ... * cvN *
```

набор множеств квалификаторов `const` и/или `volatile` $cv_1 ... cv_N$ называют ***сигнатурой cv-квалификаторов типа (cv-qualification signature)*** типа. (На верхнем уровне квалификаторов быть не может, поскольку речь идёт о преобразовании чистых праводопустимых выражений.) Возможны преобразования такого типа к типу, отличающегося только сигнатурой квалификаторов следующим образом: все существующие квалификаторы должны быть сохранены, при добавлении новых на всех уровнях правее изменяемого необходимо добавить `const`:

```
int p1,*q1 = &p1;

// Добавление квалификатора на базовом типе
// одноуровнёвого указателя всегда доступно:
// изменяемый объект без специальных свойств
// можно трактовать как неизменяемый или специальный.
const int* q2 = q;
volatile int* q3 = q;

// ОШИБКА: отбрасывание квалификатора запрещено.
int * q4 = q2;

int * * q5 = &q1;

// Добавление const на самом нижнем (левом) уровне
// потребовало добавления и на всех уровнях правее,
// кроме самого верхнего.
const int * const * q6 = q5;

// ОШИБКА: нет добавления const справа от изменяемого уровня:
const int * * q7 = q5;
```

> Если второй и третий операнды условной операции `?:` являются указателями на по разному квалифицированные типы, то результат имеет тип, к которому неявно могут быть приведены оба операнда, содержащий минимум квалификаторов:
>
> ```
> int * volatile * p = nullptr;
> int const * const * q = nullptr;
>
> // Тип выражения --- int const * const volatile *.
> true?p:q;
> ```
>
> Это расширение понятия общего типа для указателей.

Невозможность отбрасывания квалификаторов неявно соответствует важному правилу: ***попытка модификации объекта, определённого с квалификатором `const`, через леводопустимое выражение, его не имеющее, ведёт к неопределённому поведению.***

Когда квалификаторы применяются к типу категории «массив», их смысл распространяется на все элементы этого массива:

```
// Массив из 10 немодифицируемых int.
const int x[10];
int y = x[2]; // Читать можно.
x[2] = y;     // ОШИБКА: записывать нельзя.
```

Обратите внимание, что, когда квалификатор применяется к псевдониму типа, псевдоним типа рассматривается как неделимый самостоятельный тип:

```
// int_p - псевдоним типа "указатель на int".
using int_p = int*;

// p имеет тип "неизменяемое значение типа int_p".
const int_p p;

// Если в описании выше раскрыть псевдоним типа,
// то const применится ко всему значению в целом,
// а не к объекту, на который указывает указатель.
// Описание, эквивалентное предыдущему:
// "неизменяемый (указатель на int)".
int * const q;

// Если же попытаться подставить описание псевдонима
// типа как текст в описание, получится другой тип -
// "указатель на неизменяемый int"
const int *r;
```

Для явного осуществления произвольного преобразования cv-сигнатуры типа предназначена операция приведения типов `const_cast` с аналогичным `static_cast` синтаксисом. Она не ограничена правилами неявных преобразований квалификаторов и может, в том числе, их отбрасывать. Эта операция сама по себе не опасна, но если впоследствии произойдёт доступ к объекту через леводопустимое выражение с квалификаторами, не отражающими реальный статус объекта, будет вызвано неопределённое поведение. Наличие отдельной операции для такого потенциально опасного действия позволяет легко находить в программе все места, требующие скрупулёзной проверки. Применять её следует только в крайнем случае, если только вы не пользуетесь ею для добавления квалификаторов по правилам неявных прпеобразований, что изредка требуется.

При использовании синтаксиса конструкции создания производного типа категории массив в параметре функции, фактически соответствующей указателю, C++ не позволяет указать квалификаторы этого указателя. В языке C версии 99 или выше имеется конструкция для выражения этого крайне редко нужного свойства — квалификатор пишется внутри квадратных скобок:

```
1 // Явная форма: неизменяемый указатель на int.
2 void f(int * const x);
3 // Допускается в C99, но не в C++, аналог предыдущего описания:
4 void f(int x[const]);
```

7.6.1. Квалификаторы параметров функций. const-корректность.

Пожалуй, даже большую, чем для задания констант, пользу квалификатор `const` приносит при использовании его в параметрах функции, являющихся указателями. Как мы уже говорили, передачу массивов путём передачи указателя на их начало в качестве аргумента функции можно считать параметром типа in/out, поскольку он может использоваться как для чтения, так и для записи элементов. Чрезвычайно полезно отражать семантику входных/выходных параметров функции, и для этого может использоваться квалификатор `const`:

```
int sum(const int x[],std::size_t n);
```

Предположим, что функция `sum` возвращает сумму `n` элементов массива, начиная с `x`. В данном случае использование квалификатора `const` не только позволяет транслятору отследить случайные попытки изменения элементов массива, на которые указывает `x`, в функции `sum`, но, что ещё более важно, позволяет читающему описание этой функции сразу понять, что элементы, на которые указывает `x` — входные параметры и функцией модифицироваться не будут. ***Всегда используйте квалификатор `const` с параметрами функций, где это возможно, чтобы показать неизменяемость объектов, адреса которых через них передаются (явно через указатель или в форме описания массива, ему соответствующему).*** Обратите внимание на это замечание, оно является очень важным и даже имеет своё неофициальное название — ***const-корректность (const-correctness)***. Её соблюдение потребуется в том числе и при использовании стандартной библиотеки языка C++, так что к ней следует приучиться заранее.

Будьте внимательны — квалификаторы на верхнем уровне типа отвечают только за характеристики самого объекта-параметра и не входят формально в состав описания типа функции:

```
1  void f(int);
2
3  // Ещё одно описание, связанное с предыдущим той же функции f.
4  // Квалификатор на верхнем уровне параметра не учитывается, хотя
5  // и влиял бы на свойство этого объекта, если бы это было определением функции.
6  void f(const int);
7
8  // Тоже два связанных описания - квалификатор на самом объекте-параметре, а не
9  // глубже в его типе.
10 void g(int *);
11 void g(int * const);
```

const-корректность — понятие, отвечающее за свойства функции с точки зрения допустимых аргументов при её вызове. Квалификатор const на верхнем уровне типа параметра влияет только на свойства этого объекта с точки зрения самой функции, и потому к const-корректности не относится.

```
1  // Из описания функции видно, что она
2  // может изменять значения по данному адресу.
3  void read_and_write(int* p)
4  {
5      // Инкремент включает и чтение, и запись.
6      ++*p;
7  }
8
9  // Передаётся адрес неизменяемого объекта,
10 // модификация не возможна.
11 void read_only(const int* p)
12 {
13     // Чтение возможно.
14     std::cout << p << '\n';
15
16     // ОШИБКА: запись в немодифицируемый объект.
17     *p = 1;
18
19     // Сам параметр - объект-указатель изменять
20     // можно.
21     ++p;
22 }
23
24 // const на верхнем уровне,
25 // не изменяем объект-параметр.
26 void read_and_write2(int * const p)
27 {
28     // ОШИБКА: p неизменяем.
29     ++p;
30
31     // Значение по адресу p изменяемо!
32     ++*p;
33 }
34
35 void f()
36 {
37     // Адрес изменяемого объекта подойдёт
38     // обеим функциям.
39     int x = 1;
40     // Преобразования не требуются.
41     read_and_write(&x);
42     // Неявное преобразование квалификаторов int* -> const int*.
43     read_only(&x);
44
45     // Неизменяемый объект функции, которая
46     // декларирует, что может менять объект,
47     // не подойдёт.
48     const int y = 2;
49     // ОШИБКА: нет неявного преобразования const int* -> int*.
50     read_and_write(&y);
51     // Допустимо, преобразования не требуются.
52     read_only(&y);
53
54     // Допустимы оба вызова.
55     read_and_write2(&x);
56     read_and_write2(&y);
57 }
```

7.7. Расширенные константные выражения

Если объект имеет квалификатор `const` без `volatile`, целочисленный тип и инициализатор, являющийся константным выражением, подвыражение, его идентифицирующее, само может использоваться в основных констатных выражениях, вопреки запрету использования в них преобразования леводопустимого выражения. Эти требования позволяют компилятору выяснить значение объекта на этапе трансляции и получить гарантию, что во время выпол-

нения программы оно изменяться не будет. Это позволяет использовать такие объекты для задания повторяемых в тексте программы значений, которые должны быть константными выражениями:

```
// const, нет volatile, целый тип, инициализатор - константное выражение.
const std::size_t n = 16;

// n - преобразованное константное выражение,
// несмотря на "чтение" значения объекта n.
int a[n];

void f()
{
    for(std::size_t i=0;i<n;++i)
        std::cout << a[i] << '\n';
}
```

Таким образом, мы решили проблемы представления целочисленных констант без использования макросов, которая была поставлена выше.

Данное правило относится ко всем стандартам языка, но современный C++ позволяет пойти ещё дальше. Типы, операции с которыми транслятор может выполнить на этапе трансляции программы называются ***литеральными (literal)***. К ним относятся все известные нам типы данных: `void`, все скалярные типы и массивы с элементами литеральных типов. В определении объекта литерального типа с инициализатором-константным выражением, не являющимся параметром функции, можно использовать спецификатор описания `constexpr`, который включает в себя эффекты квалификтора `const`, но вместе с этим выражения, идентифицирующие такие объекты без квалификатора `volatile`, могут входить в основные константные выражения аналогично предыдущему случаю. Преимуществом по сравнению с `const` является отсутствие ограничения на целочисленность типа константы. По этой причине автор рекомендует применять этот спецификатор для всех «констант» в тексте программы, поскольку он является наиболее общей и современной формой из задания, независимо от типа:

```
constexpr char red = 'r',
               green = 'g',
               blue = 'b';

void f(char color)
{
    // Имена red/green/blue могут использоваться
    // в преобразованных константных выражениях
    // целочисленного типа char, которые требуют
    // метки case в данном операторе switch.
    switch(color){
        case red:
            // ...
            break;
        case green:
            // ...
            break;
        case blue:
            // ...
    }
}
```

7.8. Нулевой указатель

Указателю можно присвоить значение того или иного объекта, а значение объектов категории типа «указатель» с автоматическим временем хранения без инициализаторов не определено. Что тогда означает значение такого объекта со статическим временем хранения, которое инициализируется нулевыми байтами представления?

В языке имеется специальное значение, отражающее смысл «указатель, не являющийся адресом никакого объекта», называемое ***константой нулевого указателя (null pointer constant)***. Это значение может быть записано единственным литералом указателя в языке C++ — ключевым словом `nullptr`. Этот литерал имеет чисто праводопустимое значение

фундаментального литерального типа `std::nullptr_t`, описанного в `cstddef`. Этот тип имеет единственное возможное значение и представимое этим литералом, сам этот тип является фундаментальным скалярным типом, не являющимся указателем.

В то же время в языке имеется специальное ***преобразование нулевого указателя (null pointer conversion)***, входящее в состав ***преобразований указателей (pointer conversion)***, которое позволяет неявно преобразовать значение типа `std::nullptr_t` к любому типу категории указатель (сразу с требуемыми квалификаторами без отдельного преобразования квалификаторов). Полученное значение отличается от любого возможного адреса объекта и называется ***значением нулевого указателя (null pointer value)*** соответствующего типа. Именно оно может быть сохранено в указателе, чтобы явно указать, что он не указывает ни ни какой объект. Кроме того, именно это значение получают указатели, инициализированные нулём.

Такой указатель, как нетрудно догадаться, не может использоваться для разыменования и арифметики указателей — это приведёт к неопределённому поведению. Но в отличие от указателя с неопределённым значением, чтобы проверить, что в указателе хранится нулевое значение, можно явно сравнить его с константой нулевого указателя, поскольку общий тип значения типа категории указатель и `std::nullptr_t` — тип указателя. Также можно воспользоваться тем, что по булевым преобразованиям значение нулевого указателя или сама константа нулевого указателя преобразуется в `false`, а все остальные — в `true`.

```
1  // Начальное значение - нулевое.
2  int *p;
3
4  void f(int *x,int *y)
5  {
6      // Разыменовать указатель только,
7      // если он не нулевой.
8      if(x!=nullptr)
9          *x = 1;
10
11     // Более короткий вариант с использованием булевых преобразований.
12     if(y)
13         *y = 2;
14 }
```

> Из языка C C++ унаследовал ещё одно определение константы нулевого указателя — любое целочисленное константное выражение со значением 0. В стандартной библиотеке языка C его часто обозначают макросом `NULL`, чтобы отличить визуально от числа 0. Язык C++ тоже позволяет приводить такое выражение к типам указателей с нулевым значением, но автор рекомендует пользоваться новым вариантом — он сохраняет различие с числом 0, избегает использования макросов, и за счёт использования отдельного типа решает некоторые довольно непростые проблемы в разработке библиотек.

С помощью такой проверки можно реализовать выходные или входные параметры, которые опциональны:

```
1  #include <iostream>
2
3  namespace
4  {
5      bool has_negatives(const double array[],std::size_t n,
6                         double *first_negative)
7      {
8          for(std::size_t i=0;i<n;++i)
9              if(array[i]<0){
10                 if(first_negative)
11                     *first_negative = array[i];
12                 return true;
13             }
14         return false;
15     }
16 }
17
18 int main()
```

```
{
    double a1[] = {12.,34.1,0.};
    if(has_negatives(a1,3,nullptr))
        std::cout << "a1 has negative elements.\n";
    else
        std::cout << "a1 has no negative elements.\n";
    double a2[] = {15.6,-0.3,101.},fn;
    if(has_negatives(a2,3,&fn))
        std::cout << "First negative element in a2: " << fn << '\n';
    else
        std::cout << "a2 has no negative elements.\n";
    return 0;
}
```

Данная программа содержит функцию `has_negatives`, возвращающую логическое значение, означающее, есть ли отрицательные элементы в переданном ей массиве чисел с плавающей точкой заданной длины. Последним параметром функции является указатель, по адресу которого будет записано первое найденное отрицательное число (если оно будет найдено и функция вернёт `true`). Если это значение не требуется, функция позволяет указать `nullptr` в качестве адреса объекта для записи первого найденного отрицательного значения, и оно никуда записано не будет. Определить, допустим ли нулевой указатель в качестве аргумента функции, обычно можно только из документации, позднее мы покажем варианты самодокументации подобного кода.

7.9. Строковые литералы

Если ограничиться базовым набором символов исходного текста, то можно считать строки последовательностями символов. В общем случае это очень грубое упрощение, устранением которого мы займёмся позднее, а пока раскроем тип конструкции, которая уже часто применялась нами в программах — строкового литерала.

Значение строкового литерала имеет тип `const char [N]`, где N на единицу больше числа символов, заданных в тексте программы в этом литерале. Это выражение является леводопустимым и идентифицирует объект со статическим временем хранения без имени, его подобъекты содержат коды символов, из литерала в тексте программы, а дополнительный последний объект — символ с кодом 0, так называемый нулевой символ. Поскольку он завершает строку, в этой роли его также называют ***нуль-терминатором (null terminator)***. Напомним, что код символа 0 в любой кодировке не равен 0, и потому не является нулевым символом — это разные значения. Такое представление строк досталось C++ в наследство от языка C. Его преимуществом является то, что строки можно передавать в виде отдельных значений — указателей на первые символы таких строк, также называемых нуль-терминированными. Недостатком этого представления является то, что, чтобы выяснить длину строки, нужно просканировать её всю в поиске нуль-терминатора, и, поскольку он является признаком конца строки, он не может содержаться внутри неё.

Для удобства инициализации массивов узких символов, допускается использование строкового литерала в синтаксисе их инициализаторов на месте списка символьных литералов, разделённых запятыми, скобки вокруг могут быть даны или опущены. Примеры:

```
#include <iostream>

int main()
{
    // s - указатель на нуль-терминированную строку,
    // хранящуюся в объекте, соответствующем строковому
    // литералу. Неявное преобразование массива в указатель.
    const char* s1 = "abc";

    // Массив из 4 неизменяемых символов (3 указанных + нуль-терминатор),
    // инициализированных указанными в литерале символами.
    // Отдельного объекта, соответствующего строковому литералу
    // не создаётся, он используется только в качестве синтаксиса
    // записи последовательности символов...
    const char s2[] = "abc";

```

```
17      // эквивалентных следующей явной записи:
18      const char s3[] = {'a','b','c','\0'};
19
20      // Допускается запись со скобками, то же самое.
21      const char s4[] = {"abc"},
22                 s5[]("abc");
23
24      // ОШИБКА: инициализаторов больше, чем элементов массива
25      //         (не учтён нуль-терминатор).
26      const char s6[3] = "abc";
27
28      // В операции << над стандартными потоками вывода и ошибок
29      // может быть указан адрес произвольной нуль-терминированной
30      // строки, а не только строковые литералы, которые принимаются
31      // именно за счёт разложения.
32      std::cout << s3 << '\n';
33  }
```

Большинство отладчиков отображат в качестве значения объектов типа «массив символьного типа» и «указатель на символьный тип» соответствующие строки. В случае с указателем предполагается, что строка является нуль-терминированной, если это не так, после реальных данных будет отображаться мусор.

7.10. Задания на массивы

В задачах, где требуется генерация равномерно распределённых целых псевдослучайных чисел, использовать следующую вспомогательную функцию:

```
1  #include <random>
2
3  // Возвращает псевдослучайное число в интервале [min,max].
4  int uniform_random(int min,int max)
5  {
6      static std::mt19937 prng(std::random_device{}());
7      return std::uniform_int_distribution<>(min,max)(prng);
8  }
```

Тригонометрические функции описаны в заголовочном файле `<cmath>` в пространстве имён `std` для всех вещественных типов со стандартными именами (`sin`,`cos`).

В данном задании также необходимо продемонстрировать умение использовать функции, включая их использование с массивами и указателями. Предложения о том, какую функциональность выделить в функции, даны в конце каждой задачи, более широкое применение функций только приветствуется. Проследите, чтобы ваши функции были const-корректными, и все требуемые им данные являлись их параметрами — использование объектов со статическим временем хранения (кроме констант) запрещено!

0. (”Семена”). Выведите на экран 10 последовательных состояний системы клеточных автоматов, живущих на торе, разбитом на 10×10 клеток. В каждый момент времени в каждой клетке поля автомат либо ”жив”, либо ”мёртв”. Начальное состояние системы сгенерируйте псевдослучайным образом с вероятностью жизни один из трёх (если результат выглядит неинтересно, попробуйте поменять это значение). Новое состояние системы в следующий момент времени определяется независимо для каждой клетки: если автомат в клетке ”жив” — он умирает. Если автомат в клетке ”мёртв” и из его 8 соседей ровно 2 были живы в предыдущем состоянии, он оживает. 8 соседей определяются как примыкающие клетки по вертикали, горизонтали и диагонали. Поскольку речь идёт о тороидальной поверхности, все клетки, включая находящиеся на ”границах” поля имеют по 8 соседей, т.к. верх переходит в низ, а левый край — в правый, и наоборот.

   Выделите в отдельную функцию расчёт нового состояния поля.

1. (”Морской бой”). Расставьте на поле 10×10 клеток корабли по стандартным правилам псевдослучайным образом: 1 из 4 клеток, 2 из 3 клеток, 3 из 2 клеток и 4 из одной клетки так, чтобы они не касались друг друга. Созданное поле выведите на экран.

   Выделите в отдельную функцию установку на поле корабля указанного размера.

2. Выведите график параметрически заданной функции размером 40×30 символов:

$$\begin{cases} x = sin(2t) \\ y = cos(3t) \end{cases}, \ t \in [0; 2\pi)$$

   Выделите в отдельную функцию вывод сформированного в памяти графика на экран.

3. Пользователь вводит числа с плавающей точкой. После ввода каждого из них программа выводит среднее арифметическое последних пяти введённых чисел и то из них, которое ближе всего к среднему. Пока введённых чисел меньше пяти, работать с имеющимися числами. Прекращать работу программы при неудаче ввода.

   Выделите в отдельную функцию расчёт обоих характеристик, выводимых после каждого ввода.

4. Реализуйте калькулятор для ***обратной польской нотации***, работающий в диалоговом режиме. У калькулятора есть ***стек*** из промежуточных результатов в виде чисел с плавающей точкой, который изначально пуст. Калькулятор отображает текущее содержимое стека по порядку помещения в него элементов и предлагает пользователю на выбор следующие операции:

   (a) "Ввод": ввести число с плавающей точкой и поместить его в стек.
   (b) "Дубликат": поместить в стек значение из его вершины.
   (c) "Обмен": обменять местами два элемента стека, помещённых туда последними.
   (d) "+,-,*,/": извлечь из стека два значения, выполнить над ними арифметическую операцию, и поместить результат в стек. Первое извлечённое значение является правым операндом операции.
   (e) "Выход": завершить работу программы.

   Программа должна контролировать корректность операций, которые пользователь пытается совершить. Помимо этого, считать ошибкой помещение в стек 9-го элемента. Выделите в отдельную функцию вывод содержимого стека на экран.

5. Компьютер играет с пользователем, пытаясь угадать, что он задумал — 0 или 1. Машина делает догадку, после чего запрашивает у пользователя задуманное им число. Если машина угадывает, то она получает очко, если нет — очко получает пользователь. Перед новым запросом текущий счёт (начинаемый с нуля) выводится на экран. Машина не жульничает — после того, как она сделала догадку, менять её она не имеет права, то, что пользователь задумал, запрашивается только для ведения счёта и ***будущих*** догадок. Догадки машина делает так: помнит N последних задуманных пользователем чисел ($1 \leqslant N \leqslant 5$) и в зависимости от того, что чаще после n таких задумок пользователь загадывал следующим, делает своё предположение. Играть, пока пользователь корректно вводит свои загадки. Значение N запрашивать в начале работы программы.

   Выделите в отдельную функцию вычисление номера ситуации по известным прошлым загаданным пользователем значениям.

6. Программа псевдослучайным образом выбирает из колоды в 54 карты (2–10, валет, дама, король, туз в мастях пики, трефы, бубны, червы + 2 джокера) пять карт и отображает свой выбор. После этого определяется, сколько максимально одинаковых карт (без учёта масти) имеется среди выбранных. Джокер можно считать за любую карту, включая дублирование карт в колоде, в пользу игрока.

   Выделите в отдельную функцию вычисление максимального числа одинаковых карт среди выбранных.

7. Неформально, ***направленным графом*** называется множество ***вершин***, некоторые из которых соединены ***дугами***, имеющими направления. Граф называют ***некратным***, если из каждой вершины в другую не идёт более одной дуги. ***Петлёй*** называется дуга из вершины в саму себя. Если каждой дуге сопоставить некоторое число — "расстояние" между вершинами (вес), то некратный направленный граф из n вершин может быть представлен матрицей $n \times n$, где на пересечении i-ой строки и j-го столбца хранится вес дуги из вершины i в вершину j, или специальное значение ("бесконечно далеко"), если такой дуги нет.

   Сгенерируйте 30 дуг некратного графа без петель из 10 вершин псевдослучайным образом, задав им целые веса из диапазона [1;20]. Элементам матрицы, соответствующим отсутствующим дугам, присвойте удобное вам специальное значение. Выведите эту матрицу на экран.

Найдите минимальные суммарные веса ***путей*** — последовательностей дуг, где каждая следующая исходит из вершины, куда входит предыдущая — между всеми парами вершин графа ***алгоритмом Флойда***:

- Взять матрицу размера $n \times n$, смысл элементов которой — вес кратчайшего из пока известных путей между соответствующими строке и столбцу вершинами.
- Инициализировать матрицу: в элементы главной диагонали записать 0 ("из вершины в саму себя расстояние - 0"), для каждой дуги из i-ой вершины в j-ю записать в строку i столбец j значение веса этой дуги, в остальные элементы — "бесконечность".
- Для каждой вершины k перебрать все пары вершин i и j и проверить: если сумма известных на текущий момент кратчайших расстояний из i в k и из k в j меньше, чем кратчайшее известное расстояние из i в j, то заменить его вычисленной суммой.
- Теперь матрица содержит кратчайшие расстояния между вершинами.

Выведите матрицу кратчайших расстояний.

Выделите в отдельную функцию вывод матрицы целых чисел на экран.

8. Запросите у пользователя целые значения k, n и m, $2 \leqslant k \leqslant 10$, $1 \leqslant n \leqslant 40$, $1 \leqslant m \leqslant n$. Выведите все целые числа из n цифр в системе счисления с основанием k, в которых каждая цифра встречается не более m раз. Для чисел, в которых значащих цифр меньше n, дополняющие до n разрядов незначащие нули учитывать.

   Выделите в отдельную функцию формирование следующего по порядку числа, хранимого в виде массива цифр.

9. Введите последовательность из не более, чем 128 не пробельных символов. Проверьте, все ли круглые, квадратные и фигурные скобки в ней являются парными и корректно вложены друг в друга. Все остальные символы в строке игнорировать. Вывести место первой ошибки или символы, заключённые в самой вложенной паре скобок (первой из них, если таких несколько).

   Выделите в отдельную функцию ввод строки символов указанного максимального размера.

Глава 8

Динамические структуры данных

В этой главе мы познакомимся с ещё одним временем хранения объектов — ***динамическим (dynamic)***. Объекты с таким временем хранения создаются не в определениях. Вместо этого программист сам определяет, когда такие объекты создаются и уничтожаются. Эти возможности позволяют строить структуры данных любой сложности, но для этого потребуется понимание того, ***какие*** структуры данных нужны в той или иной ситуации, и как с ними работать. Теме алгоритмов и структур данных посвящены многие отдельные книги, и вопрос о выборе тех или иных методов решения задачи обычно ставят не просто в форме «как сделать», а «как сделать наиболее эффективно». Именно поэтому, прежде чем приступать к рассмотрению алгоритмов и структур данных, мы рассмотрим основы теории сложности вычислений.

8.1. Сложность вычислений

Два основных ресурса, которые используют все программы — процессорное время и память. Хотя на практике наиболее интересным является фактическое время работы, например, в секундах, при рассмотрении сложности вычислений рассматривают абстрактные шаги алгоритма, а при оптимизации реализации — число тактов процессора. При оценке объёма памяти, требуемого для работы алгоритма, в качестве единицы измерения обычно используют элемент входных данных, при этом сама память, занимаемая этими данными, не учитывается — говорят только о дополнительных расходах.

В теории сложности вычислений конкретная единица измерений не имеет значения, поскольку рассматривается ***асимптотическое*** поведение алгоритма при усложнении задачи — обычно увеличении объёма входных данных. Это позволяет сравнивать поведение алгоритмов относительно друг друга даже тогда, когда точно вычислить их ресурсоёмкость не удаётся. Различные ресурсы — процессорное время, память и другие — рассматриваются в рамках этого подхода аналогично, при этом, если не сказано иначе, под «сложностью» подразумевают время выполнения.

Существуют многочисленные разногласия по поводу точных определений и синтаксиса записи асимптотических отношений, как между разделами математики, так и отдельными авторами. Автор этой работы будет придерживаться идеологического подхода, изложенного Дональдом Кнутом в [3] с некоторыми синтаксическими изменениями.

Пусть количество некоторого ресурса, требуемого для обработки алгоритмом входных данных в количестве n элементов, есть $f(n)$ $(n, f(n) \in \{0\} \cup \mathbb{N})$. При сравнениях мы будем пользоваться и другими функциями, имеющими тот же смысл и свойства. Говорят, что «$f(n)$ есть O большое от $g(n)$», тогда и только тогда, когда $g(n)$ является ***асимптотической границей сверху*** для $f(n)$:

$$\begin{aligned} f(n) \in O(g(n)) &\Leftrightarrow \exists c > 0, n_0 : \forall n > n_0 \; f(n) \leqslant cg(n) \\ &\Leftrightarrow \overline{\lim_{n\to\infty}} \frac{f(n)}{g(n)} = c < \infty \end{aligned} \tag{8.1}$$

В [3] указывается на то, что в данной ситуации часто применяется знак равенства = в «одностороннем» смысле по традиции. Автор будет придерживаться более строгой записи через знак принадлежности элемента множеству $\in$.

Т.е., говоря неформально, f растёт асимптотически не быстрее g с точностью до константы. Символ $\in$ используется, поскольку мы подразумеваем под $O(g(n))$ *множество* всех функций, ограниченных ей сверху.

Аналогично, говорят «$f(n)$ есть Ω большое от $g(n)$», тогда и только тогда, когда $g(n)$ является ***асимптотической границей снизу*** для $f(n)$:

$$\begin{aligned} f(n) \in \Omega(g(n)) &\Leftrightarrow \exists c > 0, n_0 : \forall n > n_0 \ f(n) \geqslant cg(n) \\ &\Leftrightarrow \overline{\lim_{n\to\infty}} \frac{g(n)}{f(n)} = c < \infty \end{aligned} \tag{8.2}$$

Если для функций $f(n)$ и $g(n)$ справедливы оба указанных выше отношения, то говорят, что «$f(n)$ является Θ большое от g(n)»:

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1 > 0, c_2 > 0, n_0 : \forall n > n_0 \quad c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \tag{8.3}$$

Тот факт, что асимптотические отношения описывают поведение функций с точностью до константного множителя, и является причиной того, что выбор «масштаба», т.е. единицы измерения ресурсов не существенен. Это также позволяет абстрагироваться от деталей конкретной реализации, например, системы команд процессора. При использовании асимптотических нотаций выбирают канонического представителя с коэффициентом 1 для записи множества: хотя $f(n) = 3n \in O(5n)$, на практике используют запись $f(n) = 3n \in O(n)$.

По своему асимптотическому поведению алгоритмы разбивают на классы сложности, основной интерес при этом представляет верхняя граница. Перечислим наиболее часто встречающиеся классы в порядке усложнения и примеры алгоритмов, относящиеся к ним:

- $O(1)$ — ***константная сложность***. Ресурсоёмкость алгоритма вообще не зависит от n. Число операций при занесении элемента в аппаратный стек не зависит от числа уже находящихся в нём элементов, поэтому имеет константную сложность.
- $O(\log n)$ — ***логарифмическая сложность***. Нахождение элемента в отсортированном массиве может быть выполнено за логарифмическое время. Из формулы перехода к новому основанию логарифма следует, что все логарифмы по разным основанием от одной величины отличаются на константный множитель, который с точки зрения асимптотической сложности не существенен, поэтому в этой записи основание не указывают.
- $O(n)$ — ***линейная сложность*** или ***сложность полного перебора***. Нахождение минимального элемента в массиве требует обращения к каждому из них и имеет линейную сложность.
- $O(n \log n)$. Эту сложность имеют большинство хороших алгоритмов сортировки.
- $O(n^c), c > 1$ — ***полиномиальная сложность***. Приведение матрицы к ступенчатому виду методом Гаусса имеет сложность $O(n^3)$.
- $O(c^n), c > 1$ — ***экспоненциальная сложность***. Проверка эквивалентности двух булевых функций от n переменных путём сравнения их значений для всех возможных комбинаций аргументов имеет оценку сложности $O(2^n)$.

При указании асимптотического поведения указывают наиболее строгую оценку: для $f(n) = 3n^2$ говорят, что $f(n) \in O(n^2)$, хотя, формально, также справедливо, что $f(n) \in O(n^3)$ и $f(n) \in O(2^n)$, т.к., например, $O(n^2) \subset O(n^3)$ — множество функций, асимптотически растущих не быстрее n^3, включает множество функций, растущих не быстрее n^2 при $n \to \infty$. Аналогично поступают и с нижней границей. Для Θ такой неоднозначности не возникает, для рассматриваемой функции высказывание $f(n) \in \Theta(n^2)$ является наиболее строгим и информативным.

Задачи, для которых не известно существование алгоритмов хотя бы полиномиальной сложности, считаются «сложными». Грубо говоря, их время выполнения растёт настолько быстро, что нетрудно составить задачу, время решения которой будет неприемлемым, независимо от количественного роста вычислительной мощности техники. С вопросами существования алгоритмов полиномиальной сложности для известных сложных задач связаны основные пока не разрешённые вопросы теории вычислительной сложности.

Помимо оценок сложности сверху и снизу, можно говорить о сложности в худшем, в среднем и в лучшем случае. Что означают эти термины, зависит от рассматриваемого алгоритма, а какая оценка важнее — от решаемой задачи.

Отметим, что асимптотическая сложность не является абсолютным критерием выбора алгоритма: с её точки зрения алгоритм сложности $f(n) = n^2$ «хуже», чем $g(n) = 1000n$ (полиномиальная сложность против линейной), поскольку константные множители в асимптотических оценках не учитываются. Однако на самом деле алгоритм f станет затрачивать больше шагов, чем g, только для $n > 1000$. Возможна ситуация, когда известно, что программе никогда не потребуется обрабатывать задачи сложности большей 1000, в этом случае, очевидно, предпочтителен алгоритм f, несмотря на его более высокую асимптотическую оценку. Наконец, если вы обрабатываете не более 10 элементов, выбор алгоритма, скорее всего, никак не повлияет на ощущаемую пользователем скорость работы программы на машине, выполняющей миллиарды операций в секунду — в любом не патологическом случае вычисления будут производиться мгновенно, и следует просто выбрать наиболее простой в реализации вариант.

8.1.1. Оценка алгоритмической сложности

Рассмотрим оценку алгоритмической сложности для операций работы с массивом, используемым для хранения значений числом меньшим или равным, чем число элементов в нём. Будем записывать значения с начала массива, оставляя нужное число элементов в конце неиспользуемым. В качестве n будем рассматривать число используемых элементов массива. Ёмкостью (*capacity*) назовём общее число элементов. Основной задачей в организации такой работы с массивом является реализация операций вставки и удаления элементов в произвольные места. Вставки потребует копирования значений между элементами массива со сдвигом, чтобы освободить место в нужной позиции под новые элементы, а удаление потребует сдвига в обратном направлении, чтобы в последовательности начальных элементов массива, хранящих в настоящий момент значения, не было неиспользуемых «дырок».

```
#include <cassert>
#include <iostream>

namespace
{
    // Вывод n элементов, начиная с x, с указанием их числа.
    void print_values(const int x[],std::size_t n)
    {
        std::cout << '[' << n << ']';
        while(n--)
            std::cout << ' ' << *x++;
        std::cout << '\n';
    }

    // Функция сдвигает элементы в массиве a из capacity элементов,
    // из которых используются первые n, чтобы освободить место для
    // count элементов, начиная с where. Значения освобождённых элементов не уточняются.
    // n - указатель на число используемых элементов, которое
    // будет увеличено на необходимое значение.
    void insert_elements(int a[],std::size_t* n,std::size_t capacity,
                         std::size_t where,std::size_t count)
    {
        // Проверим входные параметры:
        // 1. Нельзя вставлять в не смежное с занятыми элементами место.
        assert(where<=*n&&
        // 2. Нельзя вставлять, если не хватит места.
               *n+count<=capacity);
        // Цикл с постусловием не подходит для тривиального случая
        // count==0, пропустим просто весь код.
        if(!count)
            return;
        // Сдвиг начиная с последнего элемента,
        // чтобы не затереть ещё не сдвинутые.
        // Если вставка в конец массива, сдвигать ничего не нужно.
        if(where<*n){
            std::size_t i = *n-1;
```

```
37              do
38                  // Сдвиг на count,
39                  a[i+count] = a[i];
40              // Пока не сдвинут последний элемент с индексом
41              // места вставки.
42              while(i--!=where);
43          }
44          // Увеличим число занятых элементов массива.
45          *n += count;
46      }
47
48      // Функция сдвигает в массиве a элементы, удаляя count
49      // значений, начиная с where. Текущее значение занятых
50      // элементов в n уменьшается на count.
51      void delete_elements(int a[],std::size_t* n,std::size_t where,std::size_t count)
52      {
53          // Нельзя удалять интервал, в котором есть не занятые элементы.
54          assert(where+count<=*n);
55          // Если нечего удалять, то зря крутить цикл нет смысла.
56          if(!count)
57              return;
58          // Сдвинем элементы, начиная с первого оставшегося
59          // слева от точки вставки where после удаления count
60          // значений, до последнего используемого...
61          for(std::size_t i=where+count;i<*n;++i)
62              // ... на count влево.
63              a[i-count] = a[i];
64          // Уменьшим число занятых элементов массива.
65          *n -= count;
66      }
67  }
68
69  int main()
70  {
71      // Ёмкость равна 10.
72      constexpr std::size_t capacity = 10;
73      // В массиве будет 5 осмысленных значений.
74      std::size_t n = 5;
75      // В массиве есть место под хранение до 10 значений,
76      // хотя в начале мы считаем, что храним там только n.
77      int a[capacity] = {1,2,3,4,5};
78      // Выведем текущие значения:
79      std::cout << "Original array: ";
80      print_values(a,n);
81      // Добавим значения 10 и 11 на место 1-го элемента:
82      insert_elements(a,&n,capacity,1,2);
83      a[1] = 10, a[2] = 11;
84      std::cout << "After insertion of 10,11 at 1: ";
85      print_values(a,n);
86      // Удалим 3 элемента, начиная с 2:
87      delete_elements(a,&n,2,3);
88      std::cout << "After deletion of 3 elements at 2: ";
89      print_values(a,n);
90      return 0;
91  }
```

Опишем сложность основных операций над такой структурой данных в асимптотической нотации:

- Доступ к элементу по его индексу — $\Theta(1)$ в любом случае. Для этого используется арифметика указателей: к адресу начала массива прибавляется индекс элемента, помноженный на его размер в байтах, после чего осуществляется доступ к памяти по этому значению. Это фиксированное число операций, не зависящее от числа хранимых значений — константная сложность.
- Поиск элемента по его значению — $O(n)$ в среднем. Просматривая элементы по порядку, придётся просмотреть в среднем $n/2$ элементов, прежде чем будет найден элемент, равный

данному. Если он отсутствует, чтобы выяснить это, понадобиться просмотреть все n, что даст сложность в худшем случае $\Theta(n)$. Такой последовательный поиск называют ***линейным (linear)***. Сложность в лучшем случае, если повезло, и искомый элемент — нулевой — $\Theta(1)$, но это не представляет интереса на практике.

- Вставка элемента — $\Theta(1)$ в конец, $O(n)$ в другие места. Вставка элемента в конец требует только доступ к первому свободному элементу и инкремент счётчика используемых. Для вставки в начало или середину придётся освобождать место, сдвигая в среднем $n/2$ элементов, что даёт линейную сложность.
- Удаление элемента — $\Theta(1)$ из конца, $O(n)$ из других мест. Аналогично вставке, только сдвиг происходит в другом направлении.

Полиномиальная оценка часто возникает при оценке алгоритмов, содержащих вложенные циклы с числом итераций, полиномиально зависящим от n. При этом оценить асимптотику часто можно без точного подсчёта числа шагов, учитывая, что если функция $f(n)$ имеет вид многочлена от n степени c с положительным коэффициентом при старшем члене, то её асимптотическая сложность есть n^c (это свойство нетрудно проверить, исходя из определений). Например, алгоритм Гаусса приведения матрицы к ступенчатому виду для квадратной матрицы порядка n имеет сложность $O(n^3)$, поскольку состоит из трёх вложенных циклов: по ведущим строкам, которые вычитаются из нижележащих, по обнуляемым строкам, и по элементам внутри обнуляемых строк. Хотя число итераций во всех циклах не равно в точности n, а для двух вложенных циклов меняется по ходу работу алгоритма, все они зависят от n линейно, в чём можно убедиться, вычислив точное число шагов алгоритма: после раскрытия всех сумм образуется многочлен третьей степени. Это оценка в среднем и худшем случае, которым для данного алгоритма является матрица полного ранга.

8.1.2. Сортировка массива

Одной из наиболее часто требуемых операций над массивом является сортировка его элементов в некотором порядке. Эта задача изучается очень давно и имеет множество решений, отличающихся разнообразными свойствами:

- Работа как с массивами данных, помещающимися в память целиком с возможностью произвольного доступа, так и с большими объёмами данных, доступ к которым ограничен. Мы будем рассматривать только первый вариант.
- Многие алгоритмы сортировки осуществляют анализ данных попарными сравнениями элементов, но возможны и другие подходы. Мы рассмотрим именно алгоритм с попарными сравнениями. Ограничением всех таких алгоритмов является асимптотическая граница снизу в среднем $\Omega(n \log n)$.
- Алгоритмы сортировки могут сортировать последовательность непосредственно там, где она хранится. Мы рассмотрим именно такие алгоритмы, при этом сложностью по памяти считают дополнительные расходы, не связанные с хранением исходной последовательности. В других подходах исходная последовательность не затрагивается, а сортированная создаётся в другом месте.
- Алгоритм сортировки может обладать свойством ***устойчивости (stability)***: сохранения относительного порядка эквивалентных элементов. Например, при сортировке чисел по возрастанию их абсолютных значений, элементы 3 и -3 являются ***эквивалентными (equivalent)*** — их относительный порядок в отсортированном массиве не важен, любой вариант удовлетворяет критерию сортировки. Все эквивалентные элементы в отсортированной последовательности будут расположены последовательно. Устойчивый алгоритм сортировки гарантирует, что относительный порядок в каждой группе эквивалентных элементов сохранится. Например, если в последовательности были только два приведённых выше значения с абсолютным значением 3 и положительное предшествовало отрицательному, то именно в таком порядке они и будут находится в отсортированном результате. Неустойчивый алгоритм сортировки может содержать в результате как последовательность 3, -3, так и -3, 3.
- ***Адаптивный (adaptive)*** алгоритм сортировки или алгоритм сортировки ***с естественным поведением*** работает тем эффективнее, чем ближе к сортированному входной набор данных. Для таких алгоритмов «лучшим» случаем является изначально отсортированный массив и для них они имеют гораздо более лучшую оценку сложности, чем в среднем.

Рассмотрим один из простых алгоритмов сортировки, называемый ***сортировкой вставками (insert sort)*** на примере сортировки целочисленных значений по возрастанию:

```
1  // Сортировка вставками n элементов, начиная с x, по возрастанию.
2  void insertsort(int x[],std::size_t n)
3  {
4      for(std::size_t i=1;i<n;++i)
5          if(x[i]<x[i-1]){
6              int t = x[i];
7              std::size_t j = i-1;
8              while(j&&t<x[j-1])
9                  --j;
10             for(std::size_t k=i;k>j;--k)
11                 x[k] = x[k-1];
12             x[j] = t;
13         }
14 }
```

Процесс сортировки вставками проиллюстрирован на рис. 8.1. Алгоритм на каждом шаге своей работы предполагает, что первые `i` элементов массива (выделены серым) уже отсортированы, для одного нулевого элемента это свойство тривиально выполняется. Затем проверяется, правильно ли расположен следующий элемент (отмечен стрелкой): если он не меньше последнего в отсортированной последовательности, то он уже стоит на своём месте и ничего делать не надо. Иначе, его необходимо «извлечь» из массива и вставить в нужное место в середину отсортированной последовательности — алгоритм носит своё название именно по этой операции. Извлечённое значение сохраняется в объекте `t`, после чего отсортированная последовательность просматривается по порядку с конца, пока не будет найдено место вставки или не будет достигнуто начало сортируемой последовательности. После этого сортированные элементы справа от места вставки сдвигаются на один элемент «вправо» (затирая обрабатываемый, для этого он и был скопирован в `t`), и производится вставка сохранённого элемента. Теперь последовательность сохранённых элементов удлинилась на один, и цикл продолжается до полной сортировки всех элементов.

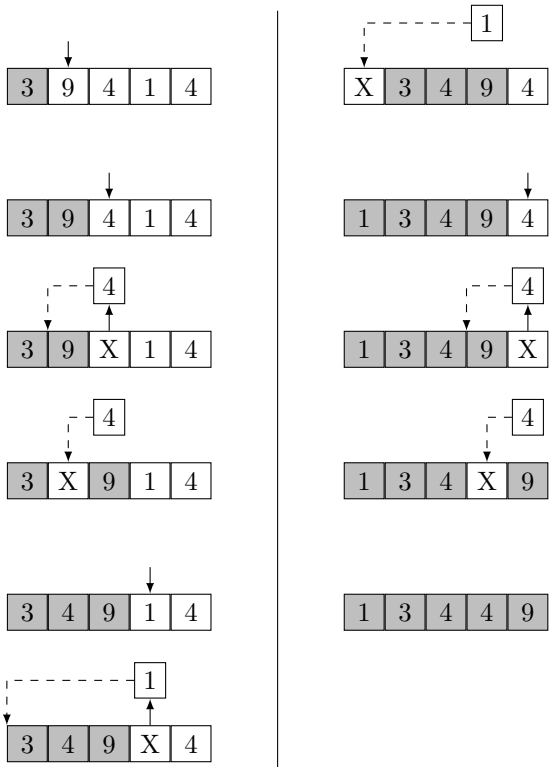


Рис. 8.1: Сортировка вставками

Оценим вычислительную сложность этого алгоритма. Для алгоритмов, использующих попарные сравнения элементов, можно отдельно оценивать количество операций сравнения

и обмена двух элементов местами, но для простоты оценим общее число шагов. При формулировке алгоритма сортировки вставками как последовательности обмена местами пар элементов этот обмен осуществляется в теле цикла поиска места вставки — обрабатываемый элемент сдвигается по одному элементу, пока не окажется на своём месте. Рассмотренная нами реализация содержит простую оптимизацию: попарные обмены заменены сдвигом элементов с сохранением обрабатываемого значения вне массива, которое потом сразу записывается в нужное место — это позволяет уменьшить (на константный множитель, а не асимптотически) число обращений к памяти. Таким образом, мы имеем цикл на $n-1$ итераций, содержащий один-два цикла с числом итераций, зависящим линейно от n, то есть алгоритм имеет сложность $O(n^2)$. Это оценка в среднем и худшем случаях, которым для данного алгоритма является последовательность, отсортированная в обратном порядке. Лучшим случаем является изначально отсортированный массив, который потребует только $n-1$ сравнений, для таких данных оценка числа шагов есть $\Theta(n)$, так что алгоритм можно считать адаптивным. Помимо объектов-счётчиков итераций и хранящих копию обрабатываемого значения, память для работы алгоритма не требуется, так что его сложность по памяти всегда $\Theta(1)$. За счёт того, что элементы перемещаются только в одном направлении этот алгоритм сортировки является устойчивым.

Хотя алгоритм сортировки вставками является весьма простым, его полиномиальная сложность не позволяет рекомендовать его для практического использования. На практике для сортировок, не имеющих каких-либо специальных требований, используются готовые гибридные реализации нескольких алгоритмов, включая более сложные с асимптотической сложностью в среднем $O(n \log n)$. Библиотека языка C++ содержит подобную реализацию алгоритма сортировки, но с ней мы познакомимся позднее.

8.1.3. Двоичный поиск

Оказывается, что осуществить поиск элемента в массиве можно за время асимптотически меньшее, чем линейное. Соответствующий алгоритм называется ***двоичным поиском (binary search)***. Этот алгоритм позволяет найти элемент (или установить его отсутствие) в отсортированной последовательности за $O(\log n)$ обращений к элементам и их сравнений с искомым.

Алгоритм работает следующим образом: рассмотрим начальную последовательность из n отсортированных элементов. Сравним элемент с индексом $n/2$ с искомым: если это искомый элемент, задача решена. Иначе в силу упорядоченности последовательности искомый элемент может находится только в одной из двух подпоследовательностей, на которые начальную разделяет проверяемый элемент: в левой, если искомый элемент меньше проверяемого, и в правой в другом случае. Примем соответствующую подпоследовательность длиной $\frac{n-1}{2}$ (плюс, возможно, ещё один элемент, если начальная последовательность содержала чётное количество элементов, и это более длинная половина) за новую начальную и повторим для неё шаг алгоритма, проверив элемент в её середине. Таким образом каждый шаг либо находит искомый элемент, либо сокращает длину интервала, в котором может находиться искомый элемент вдвое. В худшем случае на сокращение искомой последовательности до одного элемента потребуется величина порядка $\log_2 n$, т.е. сложность в худшем (и среднем) случае есть $O(\log n)$. Алгоритм проиллюстрирован на рис. 8.2. Показанная реализация представляет текущую последовательность в виде полуинтервала индексов, отсчитываемых от 0, $[begin; end)$, как это принято и в языке C++, и выбирает средний элемент как $\lfloor (begin + end)/2 \rfloor$. Возможны другие представления и округления, суть алгоритма при этом не меняется.

При наличии нескольких эквивалентных элементов двоичный поиск находит один из них. Поскольку все эквивалентные элементы расположены в отсортированной последовательности подряд, найти остальные нетрудно. В этом поведении двоичный поиск отличается от линейного, который всегда находит первый эквивалентный элемент по порядку сканирования последовательности. Кроме этого, формально, двоичный поиск требует лишь, чтобы в последовательности все элементы слева от искомого были меньше него, а все бóльшие — справа, не требуя упорядоченности этих подпоследовательностей. Однако, обобщая это требование на все возможные искомые элементы, это требование и превращается в необходимость упорядоченности всей последовательности.

Во многих случаях структура данных, которую необходимо поддерживать отсортированной для возможности быстрого поиска, не является фиксированной. В таком случае добавление в неё элементов должно производиться не в любое место, куда удобно (для массива — в конец), а в то место, где этот элемент должен находиться в соответствие с порядком сортировки. Чтобы преодолеть эту проблему, реализуем самостоятельно алгоритм ***нижней границы***

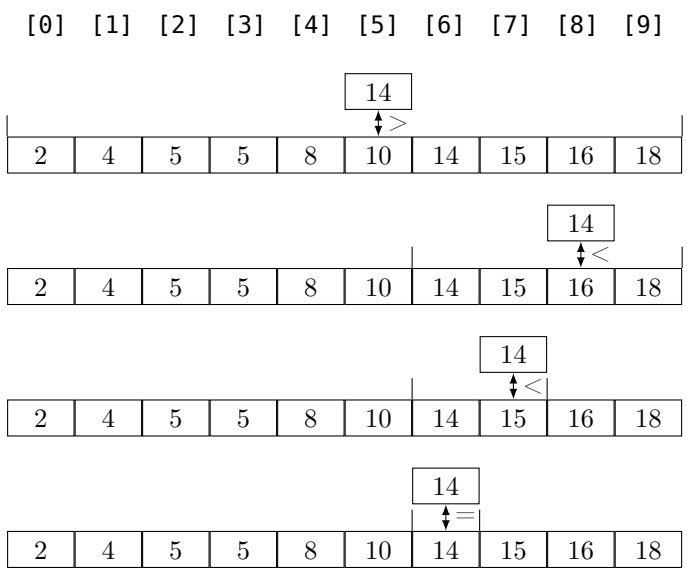


Рис. 8.2: Двоичный поиск

(lower bound), являющийся небольшой модификацией двоичного поиска. Задачей алгоритма является нахождение первого по порядку в последовательности элемента, большего либо равному данному (если такого нет, возвращается указатель на послеконечный элемент). Таким образом, результат работы этого алгоритма подходит в качестве точки вставки искомого элемента для сохранения массива в отсортированном порядке, независимо от наличия его в нём. Также, в отличие от двоичного поиска, возвращающего первый равный элемент, найденный им, алгоритм поиска нижней границы гарантировано возвращает первый по порядку элемент из идущих подряд в массиве элементов, равных данному. За это он платит сменой оценки сложности с $O(\log n)$ на $\Theta(\log n)$ — ему не может «повезти», он всегда делит начальный интервал, пока он не станет пустым. Впрочем, на практике это усложнение не играет большой роли.

```
1  // Поиск нижней границы, соответствующей данного элементу key
2  // в последовательности из n элементов по адресу p.
3  // Критерий сравнения задаётся указателем на функцию сравнения
4  // compare, которая возвращает true тогда и только тогда,
5  // как первый её параметр должен предшествовать второму
6  // в порядке, по которому упорядочен массив.
7  // Возвращается указатель на первый по порядку элемент
8  // последовательности, не <<меньший>> искомого или указатель
9  // на послеконечный элемент, если такого нет.
10 const int *lower_bound(int key,const int p[],std::size_t n,
11                         bool (*compare)(int,int))
12 {
13     std::size_t begin = 0,end = n;
14     // Пока интервал не выродился.
15     while(begin!=end){
16         // Вычислить индекс середины.
17         // (begin+end)/2 не годится из-за возможности
18         // переполнения, при котором результат будет
19         // определён, но не верен.
20         std::size_t index = begin+(end-begin)/2;
21         // Передать тестируемый элемент и ключ
22         // функции сравнения.
23         if(compare(p[index],key))
24             // Сдвинуть левый край интервала.
25             begin = index+1;
26         else
27             // Сдвинуть правый край интервала.
28             end = index;
29     }
30     // Интервал выродился в пустой, вернуть указатель
```

```
31      // на его положение.
32      return p+begin;
33  }
```

Небольшая модификация позволит превратить этот алгоритм в нахождение ***верней границы (upper bound)*** — первого по порядку элемента, большего данному. Полуинтервал между верхней и нижней границами содержит всю последовательность элементов, эквивалентных данному (он пуст, когда их нет).

8.2. Работа с динамической памятью

Два известных нам времени хранения объектов выделяют память фиксированного объёма, задаваемого типом в определении объекта. Кроме этого начало и конец существования программы они привязывают к времени работы программы или её частей. Чтобы устранить эти ограничения, рассмотрим работу с ***динамическим (dynamic)*** временем хранения.

Объекты с динамическим временем хранения создаются не в результате определений, а в качестве побочных эффектов операции `new`. Её синтаксис — ключевое слово `new`, имя типа создаваемого объекта и, опционально, инициализатор не в форме инициализации копированием. В качестве побочного эффекта вычисления этой операции производится выделение памяти под объект с динамическим временем хранения указанного типа и его инициализация данным инициализатором или по умолчанию, если его нет. Значение этой операции - адрес созданного объекта.

Объект с динамическим временем хранения существует, пока явно не будет удалён программистом, для чего используется операция `delete`: её синтаксис — ключевое слово `delete` и выражение, значение которого должно быть значением, являвшимся значением вычисления операции `new` ранее в программе, и для которого ещё не была вызвана эта операция. Значение этой операции имеет тип `void`, она обычно применяется напрямую в операторе-выражении.

Приведём простой пример:

```
1   #include <iostream>
2
3   int main()
4   {
5       // Выделение памяти под объект типа int, инициализация его
6       // по умолчанию (неопределённым значением), сохранение его
7       // адреса в объект p.
8       int* p = new int;
9
10      // Объект p создан обычным определением, его время хранения
11      // автоматическое. Объект, адрес которого он хранит, имеет
12      // динамическое время хранения и у него нет имени.
13      // В него можно записывать через разыменование указателя значения...
14
15      *p = 42;
16
17      // ... и считывать их.
18
19      std::cout << *p << '\n';
20
21      // Удаление объекта вычислением операции delete.
22      delete p;
23
24      // Теперь p - висячий указатель.
25  }
```

Поскольку объектам с динамическим временем хранения не назначаются имена, об области их видимости и связанности речь идти не может. Память для таких объектов выделяется в специальной структуре данных (***куче (heap)***), которую стандартная библиотека поддерживает для минимизации обращений к операционной системе. Часто для сокращения об её использовании так и говорят: выделение динамической памяти. Вычислительную сложность такой операции сложно оценить, поэтому работу с динамической памятью в циклах по возможности избегают, предпочитая выделять её заранее вне их тел и повторно использовать.

Этот не очень интересный пример (обычный скаляр можно было просто определить) тем не менее показывает, что время хранения такого объекта полностью контролируется

программистом. Управление памятью является одной из наиболее подверженных ошибкам аспектов реализации алгоритмов. Двумя наиболее типичными ошибками является отсутствие соотношения один-к-одному между выделениями памяти и её освобождениями. Если память не освобождается, говорят об ***утечках (leak)*** памяти. Хотя операционная система обязательно вернёт при завершении программы всю запрошенную ей память, для любой нетривиальной программы наличие утечек в процессе её работы приведёт рано или поздно к исчерпанию всей памяти среды выполнения. Утечки памяти часто проявляются в виде перезаписи значения указателя, хранящего адрес выделенного блока памяти, до того, как он был освобождён, при этом программа теряет информацию о его расположении, и он более не может быть освобождён в принципе. ***Важно понимать разницу между объектом-указателем, хранящим адрес объекта в динамической памяти, и самим этим значением, которое должно быть указано в операции*** **`delete`**:

```
void f()
{
    int* p = new int;

    ++p;

    // Ни один объект в программе сейчас не хранит
    // адреса объекта, который должен быть передан delete,
    // но память не утеряна, т.к. это значение можно восстановить:

    delete (p-1); // скобки из-за приоритета операций
}
```

Повторный вызов `delete` с тем же значением, полученным от операции `new` (если, конечно, она с тех пор не была вычислена снова и по совпадению не вернула то же значение), ведёт к неопределённому поведению, как и любое другое использование «висячего» указателя. Тот же результат будет, если попытаться применить её к любому другому значению, включая адреса объектов, время хранения которых не динамическое.

```
void f()
{
    // Указатели на три объекта типа int,
    // инициализированные адресами объектов
    // с динамическим временем хранения.
    int *pi1 = new int,
        *pi2 = new int,
        *pi3 = new int;

    // ОШИБКА: адрес первого выделенного
    //         объекта перезаписан, он теперь потерян и
    //         не может быть освобождён - утечка памяти.
    pi1 = nullptr;

    // Освобождение памяти, занимаемой вторым
    // выделенным объектом, pi2 становится висячим.
    delete pi2;

    // ОШИБКА: использование висячего указателя,
    //         undefined behavior.
    *pi2 = 0;

    // ОШИБКА: то же самое.
    delete pi2;

    int x;
    // ОШИБКА: удаление объекта не с динамическим временем хранения.
    delete &x;

    // ОШИБКА: завершается время хранения pi3, значение,
    //         которое он хранит утеряно и более не сможет
    //         быть операндом операции delete - утечка памяти.
}
```

Утечки памяти страшны тем, что могут внешне практически не проявлять себя. Для их обнаружения могут применяться специальные инструментальные средства.

Рассмотренная форма операции `new`, создающей единственный объект, называется ***скалярной (scalar)***. Больший интерес представляет ***векторная форма (vector)*** этой операции, создающая последовательность однотипных объектов в памяти в указанном количестве. Эта форма синтаксически идентична уже известной нам скалярной форме для случая, когда указанный тип внешне имеет категорию «массив», но имеет следующие отличия:

- То, что выглядит в ней как конструкция создания производного типа категории «массив» на верхнем уровне формально является отдельным элементом языка с аналогичным синтаксисом, который ***не*** требует константности выражения, задающего число элементов. Эта основная особенность позволяет создавать объекты в количестве, не известном во время трансляции программы.
- Векторная форма возвращает указатель на начальный элемент последовательности созданных объектов. Это соответствует формально указателю на указанный в ней тип, если учесть предыдущую оговорку, что верхняя конструкция создания производного типа в него не входит.
- Значение этой операции должно в конечном итоге быть передано векторной форме операции `delete`, синтаксически содержащей после ключевого слова пустую пару квадратных скобок. Скалярная форма `delete` должна применяться только к результатам скалярной формы `new`, а векторная — к векторной, при несоответствии наступает неопределённое поведение.

Поскольку результат векторной формы выделения динамической памяти устроен аналогично массиву, этот термин применяют и для него — «массив в динамической памяти», хотя формально ни одного имени массива в программе нет.

Покажем, наконец, использование динамической памяти для работы с определяемым во время работы программы числом объектов.

```
#include <iostream>

int main()
{
    // Ввести число элементов.
    std::cout << "Enter number of elements: ";
    std::size_t n;
    if(!(std::cin >> n)){
        std::cerr << "Input failed.\n";
        return 1;
    }
    // Выделить память под n элементов размером с int.
    // p - адрес начала этого блока памяти, также по
    // смыслу "нулевого элемента выделенного массива".
    // Формально в выражении присутствует только тип элемента int,
    // а [n] - часть синтаксиса векторной операции new, задающего
    // число создаваемых объектов не константным выражением.
    int *p = new int[n];
    // Ввести элементы.
    for(std::size_t i=0;i<n;++i){
        std::cout << "Enter element " << i << ": ";
        if(!(std::cin >> p[i])){
            // Не забудем освободить память на всех
            // ветвях выполнения программы после её
            // успешного выделения.
            delete[] p;
            std::cerr << "Input failed.\n";
            return 1;
        }
    }
    // Вывести элементы в обратном порядке.
    for(std::size_t i=n-1;;--i){
        std::cout << "Element " << i << ": " << p[i] << '\n';
        if(!i)
            break;
    }
    // Освободить память и завершиться.
```

```
38      delete[] p;
39      return 0;
40  }
```

Обратите внимание, что эта программа не проверяет особый случай `n==0`. Запрос памяти в объёме 0 элементов, в отличие от числа элементов настоящего типа категории «массив», считается корректным, при этом возвращается либо нулевой указатель, либо какое-то ненулевое значение, единственной корректной операцией над которым является передача операции `delete[]`. В том числе по этой причине всегда гарантируется, что операции `delete` в любой форме можно передавать нулевой указатель — в этом случае она ничего не делает.

Также обратим внимание, что во всех ошибочных ветвях после выделения памяти она освобождается, несмотря на то, что фактически программа сразу после этого завершается. ***Автор настоятельно рекомендует трактовать утечки памяти как полноценные ошибки — попытки найти оправдания этому приведут к выработке привычки, которая обернётся серьёзными проблемами в реальных проектах.***

В этом примере не было показано использование инициализаторов в операции `new`, продемонстрируем его отдельно:

```
1   void f()
2   {
3       // Выделение массива в динамической памяти и инициализация
4       // его напрямую списком.
5       int *p = new int[3]{1,2,3};
6       delete[] p;
7
8       // ОШИБКА: специальный синтаксис векторной формы не умеет
9       //         определять число элементов по инициализатору.
10      p = new int[]{1,2,3};
11
12      // Создать много объектов с начальным значением 0.
13      p = new int[1000]{};
14      delete[] p;
15  }
```

Обратите внимание, как в этом примере один объект-указатель используется для последовательного хранения адресов разных блоков динамической памяти (освобождение памяти для второго синтаксически некорректного примера не показано). Это один из основных приёмов работы с динамической памятью.

Что произойдёт с программой, если она затребует динамической памяти больше, чем есть в наличии у ОС? Используемые формы операции `new` пока приведут к аварийному завершению программы. На данный момент мы оставим открытым вопрос, как этого избежать, и вернёмся к нему после рассмотрения всех возможных языковых средств языка.

> Среда выполнения может своим устройством накладывать дополнительные ограничения на объём выделяемой памяти сверх аппаратных пределов. Например, рассмотрим ОС Windows на архитектуре x86. 32-битное адресное пространство имеет объём в 4 Гбайта, так что бóльшим объёмом памяти программа никак напрямую не может располагать, даже если физически установлено больше ОЗУ или настроено виртуальной памяти. Кроме этого, операционная система в стандартном режиме работы резервирует за собой верхнюю половину этого объёма, так что под данные программы остаются нижние 2 Гбайта. В нём обязательно находятся части образа программы и динамических библиотек, расположенные к тому же не вплотную, а с «дырками». В итоге на практике на такой системе не удаётся выделить более одного с небольшим гигабайта памяти в виде непрерывного блока.
>
> Во многих конфигурациях ОС Linux включён специальный режим, в котором система удовлетворяет запрос выделения памяти, даже если соответствующими ресурсами не располагает. В таких случаях операция выделения памяти успешна, но при фактической попытке обращения к памяти по полученному адресу происходит аварийное завершение работы программы. Это не очень удобное во многих случаях поведение ещё более усложняет отложенный вопрос об обработке ошибок выделения памяти.

При работе в отладчике Qt Creator указатели на последовательности элементов, выделенные в динамической памяти, всегда будут содержать только один дочерний элемент, как и любые другие указатели. Необходимость изучить последовательность объектов по данному адресу

могла возникать и ранее, но при использовании динамической памяти она возникает постоянно. Рассмотрим пример:

```
1  void f(std::size_t n)
2  {
3      int* p = new int[n]{};
4      // ...
5      p[2] = 3; // <-- остановим программу здесь
6      // ...
7      delete[] p;
8  }
```

С помощью опции контекстного меню «Add New Expression Evaluator» («Добавить новое вычисляемое выражение») добавьте выражение `*p@n`. Суффикс `@` с выражением указывает, что необходимо отобразить последовательность из указанного числа объектов, начиная с данного. Выражение длины само может быть произвольным не константным выражением. Если в рассматриваемом примере `n==3`, вы увидите, что тип этого выражения в данный момент `int [3]` и оно содержит три дочерних элемента. Этот приём работает независимо от времени хранения объектов. В зависимости от используемого отладчика и/или среду разработки, синтаксис таких выражений может различаться.

Иногда вместо этого проще просто выделить единственный дочерний узел указателя и выбрать в контекстном меню «Open Memory Editor — Open Memory View/Editor at Object Address» («Открыть редактор памяти — Открыть вид/редактор памяти по адресу объекта»), чтобы изучить интересующий участок памяти в шестнадцатеричном виде.

8.2.1. Владение

Для обсуждения поведения программы по обеспечению парности вычисления операций выделения и освобождения динамической памяти вводят понятие ***владения (ownership)***. Владением называют ответственность той или иной части программы по освобождению полученного ей ресурса. Ресурсы могут идентифицироваться различными значениями, и основной трудностью в их использовании является сохранение значения и выполнение действий по освобождению идентифицируемого им ресурса во всех вариантах передачи управления в программе. Ресурсы должны освобождаться однократно, только после того, как перестают быть нужны, и, желательно, сразу в этот момент, а не позднее.

Значения, идентифицирующие ресурсы, могут хранится в произвольном количестве объектов, но независимо от этого владение классифицируют по множеству логически независимых частей программы, которые им пользуются. Часто ресурс требуется одной конкретной области в программе, или такое разделение на части не имеет смысла, в таком случае говорят о ***единственном (unique)*** владении. В более сложных случаях несколько частей программы используют один и тот же ресурс, и предсказать в каком порядке они заканчивают пользоваться им (и, следовательно выбрать одного единственного владельца) нельзя. В таком случае говорят о ***разделяемом (shared)*** владении. Этот более сложный случай пока в наших программах встречаться не будет.

Одним из основных ресурсов в программах является динамическая память, поскольку она требует явных действий по своему освобождению. Этот вид ресурса идентифицируется указателями на выделенные в динамической памяти объекты.

При рассмотрении объектов с динамическим временем хранения в качестве ресурсов, ***владеющим (owning)*** называют объект-указатель, значение которого будет использоваться для передачи операции `delete` до его изменения или потери. В случае единственного владения только один указатель на объект в динамической памяти может быть только владеющим в любой момент времени, т.к. вызывать `delete` для одного значения дважды нельзя. Изменение, какой из указателей является владеющим для объекта в динамической памяти, называют ***передачей владения (ownership transfer)***. Рассмотрим это понятие на примере.

```
1  #include <iostream>
2
3  void f(int *p)
4  {
5      *p = 42;
6  }
7
8  void g(int *p)
```

```
 9  {
10      std::cout << *p << '\n';
11      delete p;
12  }
13
14  int main()
15  {
16      int *p = new int;
17      f(p);
18      g(p);
19  }
```

После выделения памяти операцией `new` единственный объект, хранящий связанное с этим адресом значение — `p`. Как единственный такой указатель, он должен считаться владеющим, поскольку других нет, а отсутствие у объекта в динамической памяти владеющего указателя по определению совпадает с утечкой памяти — перед потерей значения владеющего указателя из-за присваивания ему нового или завершения его времени хранения, имеющееся значение должно быть передано `delete`.

Далее происходит вызов функции `f`, и в программе появляется второй указатель на объект в динамической памяти — её параметр. Функция `f` не содержит каких-либо попыток передать полученное значение операции `delete` или сохранить его где-либо ещё, и в её конце параметр как автоматический объект уничтожается. Говорят, что эта функция не принимает владения объектом, адрес которой передан ей в качестве параметра, поэтому этот параметр — не владеющий указатель. Такой функции, вообще говоря, даже всё равно, какое время хранения имеет этот объект. Это поведение функций относительно параметров-указателей обычно не документируется, потому что считается стандартным.

Функция `g` напротив содержит вызов операции удаления объекта из динамической памяти по полученному ей адресу. Мало того, что необходимо задокументировать, что это единственное время хранения, адреса объектов с которым ей можно передавать, следует чётко отметить, что эта функция ***принимает владение объектом, адрес которого указан её параметром***. Поскольку только один указатель по смыслу может быть владеющим в тот или иной момент, при вызове функции происходит передача владения объектом, выделенным в функции `main`, и теперь функция `g` ответственна за его удаление. Её параметр — владеющий указатель, а после её вызова указатель в основной программе перестаёт таковым быть. В данном случае он ещё и становится висячим, хотя в общем случае это может и не происходить — функция могла записать адрес объекта, которым теперь владеет, например, в другой объект со статическим временем хранения, и к моменту возврата из неё данный объект всё ещё существовал бы. В любом случае, поскольку указатель в функции `main` более не является владеющим, операция `delete` на его значение не вызывается.

Свойство владения пока не имеет синтаксического отражения в языке — функции `f` и `g` имеют идентичные описания, но их семантика в части владения принципиальна отличается. Пока нам придётся закреплять эти соглашения исключительно в документации, но позднее мы познакомимся со средствами, которые позволят закрепить за обычными указателями языка статус не владеющих, чтобы устранить ручное отслеживание владения.

> Спецификой указателей, как потенциальных идентификаторов ресурсов, является то, что не обязательно хранить непосредственно значение, полученное в результате вычисления операции `new`, если его можно восстановить:
>
> ```
> 1  void f()
> 2  {
> 3      int *p = new int[10]+5;
> 4      // Значения, непосредственно идентифицирующего
> 5      // выделенную динамическую память, в программе сейчас нигде не хранится.
> 6      p[3] = 2;
> 7      // Но память не потеряна, т.к. это значение можно восстановить:
> 8      delete[] (p-5);
> 9  }
> ```
>
> Подобное на практике встречается не часто, но является примером того, что понятие владеющего указателя является лишь способом упорядочивания рассуждений об удовлетворении обязанностей, наложенных владением.

Владеть ресурсами могут разные элементы программы — указатели, функции и другие

конструкции. Мы будем стараться обсуждать эти вопросы в дальнейших примерах, поскольку они являются способом отслеживания ошибок работы с динамической памятью.

8.2.2. Динамические массивы

Под ***динамическими массивами*** будем понимать стратегию работы с последовательностью объектов с динамическим временем хранения, где размер выделенной памяти, и, следовательно, число элементов в ней может изменяться в процессе работы программы. Помимо числа хранимых элементов динамический массив обладает ***ёмкостью (capacity)*** — числом элементов, под которые фактически выделена память, оно всегда больше либо равно числу хранимых элементов. Разницу между фактически хранимыми элементами и ёмкостью составляют элементы в конце динамического массива, которые не используются и составляют запас для его роста. Когда запас кончается, необходимо выделить больший блок памяти, скопировать туда старые элементы, после чего удалить старый блок.

Рассмотрим реализацию двух основных операций по изменению количества объектов в такой последовательности: вставка и удаление элементов. Эти операции осуществляют копирование значений справа от точки вставки или удаления элементов так, что хранимые элементы всегда расположены в памяти подряд, как в обычном массиве. При росте числа элементов может потребоваться заменить старый блок динамической памяти новым большего объёма, поэтому функция `insert_space`, осуществляющая раздвижку элементов для освобождения места новым, принимает указатель на объект, хранящий этот адрес, т.е. указатель на указатель. Ёмкость тоже может меняться, поэтому тоже принимается по указателю. При удалении элементов функцией `delete_elements` создавать новый блок памяти не требуется, поэтому в её описании достаточно передачи указателя на блок памяти по значению. Также в следующем примере показана функция вывода содержимого динамического массива.

```
1  #include <cassert>
2  #include <iostream>
3
4  namespace
5  {
6      std::size_t max(std::size_t a,std::size_t b)
7      {
8          return a>b?a:b;
9      }
10
11     void insert_space(int** a,std::size_t* n,std::size_t* capacity,
12                       std::size_t where,std::size_t count)
13     {
14         // Проверим входные параметры.
15         // *n+count>*capacity больше не является ошибочной ситуацией,
16         // т.к. мы теперь можем расширить массив.
17         assert(*n<=*capacity&&where<=*n);
18         if(!count)
19             return;
20         std::size_t new_n = *n+count;
21         if(new_n>*capacity){
22             // Число занятых элементов плюс длина вставляемой
23             // последовательности больше, чем объём выделенной памяти,
24             // расширим его.
25             // Новый объём памяти в элементах в 1.5 раза больше старого,
26             // но не менее 4 (меньше нет смысла выделять, слишком быстро
27             // понадобиться перевыделять заново, граница примерная), или
28             // новое число элементов, если оно больше.
29             *capacity = max(max(*capacity*3/2,4),new_n);
30             // Пробуем выделить новый блок памяти.
31             int *new_a = new int[*capacity];
32             // Скопируем значения из старого блока в новый,
33             // оставляя неиспользуемый промежуток в нужном месте.
34             for(std::size_t i=0;i<where;++i)
35                 new_a[i] = (*a)[i];
36             for(std::size_t i=where;i<*n;++i)
37                 new_a[i+count] = (*a)[i];
38             // Удалим старый блок и заменим его новым.
```

```
39              delete[] *a;
40              *a = new_a;
41          }else{
42              // Место есть, сдвигаем как и раньше.
43              if(where<*n){
44                  std::size_t i = *n-1;
45                  do
46                      a[i+count] = a[i];
47                  while(i--!=where);
48              }
49          }
50          *n += count;
51      }
52
53      // delete_elements остаётся без изменений, т.к. не приводит к расширению.
54      void delete_elements(int a[],std::size_t* n,std::size_t where,std::size_t count)
55      {
56          assert(where+count<=*n);
57          if(!count)
58              return;
59          for(std::size_t i=where+count;i<*n;++i)
60              a[i-count] = a[i];
61          *n -= count;
62      }
63
64      void print_array(const int a[],std::size_t n,std::size_t capacity)
65      {
66          std::cout << n << '(' << capacity << "): ";
67          for(std::size_t i=0;i<n;++i)
68              std::cout << ' ' << a[i];
69          std::cout << '\n';
70      }
71  }
72
73  int main()
74  {
75      // Пустой динамический массив.
76      int* a = nullptr;
77      std::size_t capacity = 0,
78                  n = 0;
79      print_array(a,n,capacity);
80
81      // Вставка элементов {0,1,2,3,4,5} в начало массива.
82      insert_space(&a,&n,&capacity,0,6);
83      for(std::size_t i=0;i<6;++i)
84          a[i] = i;
85      print_array(a,n,capacity);
86
87      // Вставка элемента 100 в позицию 2.
88      insert_space(&a,&n,&capacity,2,1);
89      a[2] = 100;
90      print_array(a,n,capacity);
91
92      // Удаление интервала [4;5].
93      delete_elements(a,&n,4,2);
94      print_array(a,n,capacity);
95
96      delete[] a;
97
98      return 0;
99  }
```

При увеличении динамического массива приходится выбирать стратегию увеличения роста его ёмкости. Увеличивать её каждый раз ровно на столько, сколько нужно, не выгодно — вставка часто осуществляется всего по одному элементу, а значит перевыделение памяти, влекущее за собой потенциально копирование всех элементов в новый блок памяти, будет

выполняться каждый раз. Хорошие результаты выделения с запасом даёт геометрическая прогрессия: требуемый размер умножается на 1.5. Это значение учитывает особенности работы механизмов выделения памяти. Начальный размер массива 4 задан произвольно, обычно его выбирают исходя из особенностей задачи.

Оценим алгоритмическую сложность вставки элемента в динамический массив. Вставка не в конец имеет ту же сложность, что и для массива фиксированной размерности: на перемещение элементов как для освобождения места для вставки, так и при перемещении блока памяти, требуется число шагов, линейно зависящее от числа элементов: $O(n)$. О вставке элементов в конец динамически расширяемого массива говорят, что она имеет ***амортизированную константную сложность***. Это означает, что, хотя большинство таких операций (которые не потребуют перевыделения памяти) завершаются за время, не зависящее от n, некоторые редкие операции будут иметь линейную сложность (с перевыделением). Выбор геометрической прогрессии в качестве функции роста размера массива обеспечивает тот факт, что число перевыделений при вставке в конец n элементов зависит от него логарифмически, поэтому дополнительная сложность будет заметна только на отдельных операциях, и всегда найдётся такое $c > 0$, что для каждой последовательности операций длины L число шагов не превышает $c \times L$.

Алгоритм удаления элемента никогда не заменяет имеющийся блок памяти — удаление элементов только увеличивает запас ёмкости, который можно будет использовать для дальнейших вставок без перевыделения памяти. Если известно, что использование памяти сокращается намного и надолго, можно модифицировать функцию `delete_elements`, чтобы она освобождала неиспользуемую память, если её объём относительно выделенной превышает определённую часть. Такое поведение требуется не часто, оставим его реализацию читателю.

Приведём ещё один пример использования динамического выделения памяти. Напишем функцию, которая считывает с клавиатуры строки любой длины. Она будет использовать ту же стратегию роста массива, поскольку необходимый его объём наперёд не известен. Ввод символа из стандартного потока ввода пропускает пробельные символы, включая переводы строки, которые нам интересны. Отключить это поведение можно вводом из него специального имени std::noskipws, которое также описано в `iostream`, восстановить стандартное поведение можно вводом объекта `std::skipws` (SKIP WhiteSpace).

```
#include <iostream>

namespace
{
    // Возвращает динамически выделенную нуль-терминированную строку, содержащую
    // считанные из потока stream символы до первого перевода
    // строки (не включая его) или конца файла.
    // Если в процессе чтения происходит ошибка, включая невозможность
    // считать ни один символ, возвращается нулевой указатель.
    // Функция передаёт владение этим блоком памяти вызывающей функции,
    // которая должна освободить его операцией delete, когда оно более не нужно.
    char* get_string()
    {
        // Начнём с 4 байт.
        constexpr std::size_t initial_size = 4;
        char* buf = new char[initial_size];
        std::size_t used = 0,allocated = initial_size;
        // Отключаем пропуск пробельных символов.
        std::cin >> std::noskipws;
        // Пока не конец строки или неудача ввода, вводим символы.
        char c;
        while((std::cin>>c)&&c!='\n'){
            // Если осталось меньше двух свободных байтов
            // (один под считанный символ, второй заранее
            // резервируемый под нуль-терминатор), расширить блок.
            if(used+2>allocated){
                // Геометрический рост.
                allocated = allocated*3/2;
                char* new_buf = new char[allocated];
                for(std::size_t i=0;i<used;++i)
                    new_buf[i] = buf[i];
                delete[] buf;
                buf = new_buf;
```

```
34              }
35              // Запишем очередной символ в массив и увеличим
36              // счётчик считанных символов.
37              buf[used++] = c;
38          }
39          // Включаем пропуск пробельных символов.
40          std::cin >> std::skipws;
41          if(!std::cin){
42              // Ввод неудачен, удаляем считанное и
43              // зануляем адрес, который вернём.
44              delete[] buf;
45              buf = nullptr;
46          }else
47              // Ввод удачен, строку надо завершить нуль-терминатором.
48              buf[used] = '\0';
49          return buf;
50      }
51  }
52
53  int main()
54  {
55      std::cout << "Input a string: ";
56      char* s = get_string();
57      if(s){
58          std::cout << "Your input: " << s << '\n';
59          delete[] s;
60      }
61      return 0;
62  }
```

8.2.3. Обзор линейных структур данных

Рассмотрим простые структуры данных, которые можно представить последовательностью элементов массива, и их свойства. Указаны средние сложности операций.

- ***(Динамические) массивы***. В самом общем случае массив может использоваться для хранения последовательности значений в количестве, не превышающем число элементов в нём. Использование динамического выделения памяти позволяет произвольно увеличивать этот объём.

  Сложность операций: доступ к элементу по индексу — $\Theta(1)$, поиск элемента — $O(n)$ (линейный) или ($O(\log n)$ (двоичный, только для отсортированных последовательностей), вставка и удаление с конца — (амортизированное) $O(1)$ (амортизация при удалении возникает, когда используется стратегия освобождения лишнего не используемого выделенного пространства), вставка и удаление в другие места — $O(n)$.

- ***Стек***. Обычный массив, в котором используется только некоторое количество элементов в начале, позволяет добавлять и удалять элементы с конца этой последовательности, поэтому он удовлетворяет определению стека — аппаратный стек является таким массивом машинных слов. Помимо строго определения только с операциями push и pop, элементы массива поддерживают эффективный доступ по индексам, начиная от вершины стека. Использование динамического выделения памяти позволяет произвольно расширять максимальную ёмкость стека.

  Сложность операций: push и pop — (амортизированное) $O(1)$, peek и доступ к элементу по индексу относительно вершины — $\Theta(1)$.

- ***Буферное окно (gap buffer)***. Буферное окно является стратегией работы с элементами массива, дающей выигрыш в производительности операций вставки и удаления, когда известно, что последовательности таких операций часто осуществляют работу с одним и тем же местом массива подряд. В отличие от обычного массива, неиспользуемое место располагается не всегда в конце массива, а сдвигается при каждой операции вставки или удаления на место этой операции. В общем случае элементы массива содержат последовательно: первую часть данных, неиспользуемое пространство и вторую часть в конце, где первая и последняя части могут отсутствовать, когда окно расположено строго в начале или конце массива. При такой организации операции вставки и удаления в то же место, где происходила последняя такая операция, имеют (амортизированную для динамически расширяемого

варианта) константную сложность. Такие последовательности операций характерны, например, при работе с буфером символов текстового редактора: между перемещениями курсора обычно происходит множество нажатий клавиш с символами или `Baskspace` и `Delete`. За ускорение этих операций приходится платить усложнением алгоритмов работы с такой структурой данных: помимо числа используемых элементов и общего их количества необходимо хранить и учитывать положение «окна» из неиспользуемых элементов. Кроме того, когда окно расположено не в конце массива, т.е. большинстве случаев, используемые элементы не образуют непрерывной последовательности в памяти, что может не позволить использовать с таким представлением некоторые стандартные средства.

Сложность операций: как у массива, но (амортизированную) $O(1)$ сложность вставки и удаления имеют операции не в конец, а в место последней такой операции.

- ***Очередь (queue)***. В отличие от стека, являющегося структурой типа FILO — First In, Last Out, очередь обладает поведением FIFO — First In, First Out. Очередью называют последовательность элементов, к которой применимы две операции: помещение в очередь (***enqueue***) — добавление элемента с одного конца (***хвоста (tail)***) последовательности и извлечение из очереди (***dequeue***) — удаление элемента с другого конца (***головы (head)***). Набор дополнительных операций, применяемых сверх этого определения, можно получить аналогично стеку. Чтобы обеспечить при хранении элементов в очереди константную сложность двух основных операций, обходятся без сдвига элементов. Вместо этого, последовательность элементов стека представляют замкнутой в кольцо. Вместе с массивом хранят указатели (или индексы) головы и элемента после хвоста (чтобы оперировать полуинтервалами). При добавлении и извлечении элементов эти индексы сдвигаются, если они при этом выходят за границы допустимых для массива, они, согласно его концептуальной кольцевой структуре, перемещаются на противоположный его конец. Равными они являются только для случаев пустой и полностью заполненной очереди. В зависимости от их положения используемые элементы могут храниться в виде одной или двух последовательностей элементов с дыркой. Когда такая организация очереди использует массив фиксированного размера, её называют ***Кольцевым буфером (ring buffer)***.

  Сложность операций: enqueue и dequeue — (амортизированное при динамическом расширении) $O(1)$.

- ***Дек (двухстороняя или двусвязная очередь) (deque, dequeue, double-ended queue)***. Концептуально дек является объединением понятий стека и очереди — он допускает и добавление и извлечение элементов с обоих концов последовательности (***push back, pop back, push front, pop back***). При реализации его на базе массива с ним работают аналогично очереди.

  Сложность операций: –push back, pop back, push front, pop back: — (амортизированное при динамическом расширении) $O(1)$.

8.3. Многомерные структуры данных

Мы уже работали с массивами массивов, т.е. «многомерными массивами». Проблемы, связанные с работой с такими структурами, когда их размер не фиксирован, были описаны в разделе 7.4.1. Рассмотрим два типовых подхода к работе с многомерными структурами данных с использованием динамического выделения памяти.

8.3.1. Шаговый доступ

Данный способ идентичен случаю, когда происходит последовательная индексация массива массивов на уровне языка C++. Вспомним (см. рис.7.1), что массив массивов хранится в памяти как последовательность элементов, индексы которых последовательно возрастают по порядку, начиная с самых вложенных в конструкциях создания производных типов. Рассмотренный случай можно обобщить на случай n-мерного массива, являющегося n-мерным гиперпрямоугольником.

Пусть s_i — размерность i-го измерения, $i = 1 \dots n$. Память для хранения всех элементов в этом подходе выделяется одним блоком памяти, размером в $\prod_{i=1}^{n} s_i$ элементов. Будем считать первое измерение самым «вложенным», тогда элементы, отличающиеся только на единицу этим индексом расположены в памяти друг за другом, таким образом, расстояние между ними или ***шаг (stride)*** равен 1 (в элементах). Элементы, отличающиеся только вторым индексом на единицу, содержатся в разных одномерных «массивах» первого уровня по одинаковым

индексам в них, т.е. расстояние в памяти между ними есть s_1. То же расстояние для элементов, отличающихся на единицу третьим индексом, есть полный размер «слоя» из первого и второго измерений размером в $s_1 \times s_2$ элементов. В общем случае, изменение индекса i на единицу означает смещение в памяти на число элементов в слоях с первого по $(i-1)$-ый, т.е. $\prod_{j=1}^{n-1} s_j$.

Вектор $S = \left\{\prod_{j=1}^{i-1} s_j\right\}_i, i = 1 \ldots n$ называют ***вектором шагов***. Чтобы получить смещение в элементах от начала выделенного блока памяти до элемента с вектором индексов $X = x_i$, $i = 1 \ldots n$, необходимо вычислить скалярное произведение (S, X).

Как уже было сказано ранее, из определений понятий «массив» и операций арифметики указателей следует, что именно так и проводит вычисления компилятор при работе с массивами массивов, с той лишь разницей, что оперирует смещениями не в элементах первого измерения, а в байтах. Рассмотрим, например, определение `int a[s3][s2][s1]`, где `s1`, `s2` и `s3` — некоторые целочисленные константные выражения. Раскроем выражение `a[x3][x2][x1]` по определению:

```
a[x3][x2][x1] == *(*(*(a+x3)+x2)+x1)
```

Разыменованные значения во всех случаях, кроме последнего, имеют категорию типа «массив», которая тут же разлагается в указатель на свой нулевой элемент, поэтому этой паре операций уровня языка C++ на уровне оперирования адресами в машинном коде никаких действий не соответствует. Раскроем также по определению арифметики указателей операции сложения на уровне адресов, домножив смещения на размер типа соответствующего операнда-указателя в байтах. Получим следующий адрес, по которому осуществит обращение самая внешняя операция разыменования:

$$\begin{gathered}\text{адрес } a + \texttt{x3} \times \texttt{sizeof(int [s2][s1])} + \\ \texttt{x2} \times \texttt{sizeof(int [s1])} + \texttt{x1} \times \texttt{sizeof(int)} = \\ \text{адрес } a + \texttt{x3} \times \texttt{sizeof(int)} \times \texttt{s2} \times \texttt{s1} + \\ \texttt{x2} \times \texttt{sizeof(int)} \times \texttt{s1} + \texttt{x1} \times \texttt{sizeof(int)} = \\ \text{адрес } a + \texttt{sizeof(int)} \times (\texttt{x3} \times (\texttt{s2} \times \texttt{s1}) + \\ \texttt{x2} \times \texttt{s1} + \texttt{x1} \times 1)\end{gathered}$$

Таким образом, адрес элемента вычисляется как адрес начала всего многомерного объекта плюс смещение, равное размеру его элемента самого нижнего уровня (`sizeof(int)`), помноженное на выражение в скобках, являющееся раскрытием скалярного произведения вектора координат $\{\texttt{x1}, \texttt{x2}, \texttt{x3}\}$ на вектор шагов, который для данной трёхмерной структуры есть $\{1, \texttt{s1}, \texttt{s1} \times \texttt{s2}\}$.

Приведём пример использования этого способа организации хранения данных.

```
#include <iostream>

namespace
{
    void generate_data(double x[],const std::size_t dims[3])
    {
        for(std::size_t i=0;i<dims[0];++i)
            for(std::size_t j=0;j<dims[1];++j)
                for(std::size_t k=0;k<dims[2];++k)
                    x[k+dims[2]*(j+i*dims[1])] =
                        3.5+i*(2.8-j*(0.1+2.5*k));
    }

    void display_data(const double x[],const std::size_t dims[3])
    {
        std::cout << "Table of f(i,j,k) = 3.5+i*(2.8-j*(0.1+2.5*k))\n"
                     "for integer i in [0;" << dims[0] << "), j in [0;"
                  << dims[1] << ") and k in [0;" << dims[2] << "):\n";
        for(std::size_t i=0;i<dims[0];++i)
            for(std::size_t j=0;j<dims[1];++j)
                for(std::size_t k=0;k<dims[2];++k)
                    std::cout << "f(" << i << ',' << j << ',' << k << ") = "
                              << x[k+dims[2]*(j+i*dims[1])] << '\n';
    }
```

```
25  }
26
27  int main()
28  {
29      std::cout << "Enter 3 dimensions: ";
30      std::size_t dims[3];
31      if(!(std::cin >> dims[0] >> dims[1] >> dims[2])){
32          std::cerr << "Invalid input.\n";
33          return 1;
34      }
35      double* x = new double[dims[0]*dims[1]*dims[2]];
36      generate_data(x,dims);
37      display_data(x,dims);
38      delete[] x;
39      return 0;
40  }
```

В данном примере вместо вычисления массива шагов применяется конструкция, схожая со схемой Горнера вычисления значения многочлена, чтобы не хранить массив шагов, ограничившись только массивом измерений, и оставить число операций в вычислении смещения аналогичным таковому для скалярного произведения.

Эта схема хранения данных даже позволяет относительно легко работать со структурами, у которых не только размерность измерений, но и само их число определяется произвольно во время выполнения программы. Кроме того при работе с настоящими векторами шагов появляется возможность, как и с обычным многомерным массивом, писать функции работающие как с полным объёмом данных, так и с любой частью такого многомерного массива, имеющей вид гиперпрямоугольника размерности m, $m \leqslant n$, в общем виде.

8.3.2. Массивы указателей и «рваные» массивы

Другой способ организации работы с многомерной структурой данных покажем на примере, выполняющим те же операции, что и предыдущий:

```
1   #include <iostream>
2
3   namespace
4   {
5       void free_data(double*** x,const std::size_t dims[3])
6       {
7           for(std::size_t i=0;i<dims[0];++i){
8               for(size_t j=0;j<dims[1];++j)
9                   delete[] x[i][j];
10              delete[] x[i];
11          }
12          delete[] x;
13      }
14
15      double*** allocate_data(const std::size_t dims[3])
16      {
17          double*** x = new double**[dims[0]];
18          for(std::size_t i=0;i<dims[0];++i){
19              x[i] = new double*[dims[1]];
20              for(std::size_t j=0;j<dims[1];++j)
21                  x[i][j] = new double[dims[2]];
22          }
23          return x;
24      }
25
26      void generate_data(double * const * const *x,const std::size_t dims[3])
27      {
28          for(std::size_t i=0;i<dims[0];++i)
29              for(std::size_t j=0;j<dims[1];++j)
30                  for(std::size_t k=0;k<dims[2];++k)
31                      x[i][j][k] = 3.5+i*(2.8-j*(0.1+2.5*k));
32      }
```

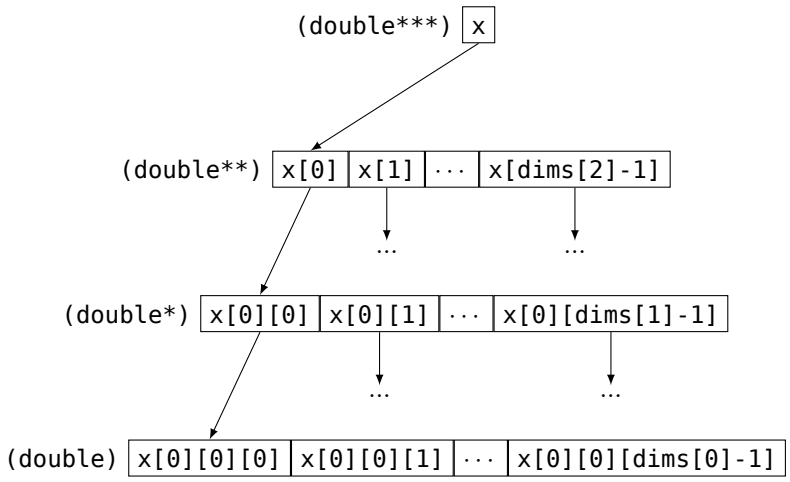


Рис. 8.3: «Рваный» массив

```
33
34      void display_data(const double * const * const *x,const std::size_t dims[3])
35      {
36          std::cout << "Table of f(i,j,k) = 3.5+i*(2.8-j*(0.1+2.5*k))\n"
37                       "for integer i in [0;" << dims[0] << "), j in [0;"
38                    << dims[1] << ") and k in [0;" << dims[2] << "):\n";
39          for(std::size_t i=0;i<dims[0];++i)
40              for(std::size_t j=0;j<dims[1];++j)
41                  for(std::size_t k=0;k<dims[2];++k)
42                      std::cout << "f(" << i << ',' << j << ',' << k << ") = "
43                                << x[i][j][k] << '\n';
44      }
45  }
46
47  int main()
48  {
49      std::cout << "Enter 3 dimensions: ";
50      std::size_t dims[3];
51      if(!(std::cin >> dims[0] >> dims[1] >> dims[2])){
52          std::cerr << "Invalid input.\n";
53          return 1;
54      }
55      double*** x = allocate_data(dims);
56      generate_data(x,dims);
57      display_data(x,dims);
58      free_data(x,dims);
59      return 0;
60  }
```

Данная программа внешне ведёт себя аналогично предыдущей.

Устройство структуры данных, которую использует этот пример, показано на рис. 8.3. В отличие от подхода с шаговым доступом, она состоит из множества отдельных блоков памяти. На схеме показаны только блоки, необходимые для доступа к нулевому элементу по всем измерениям, блоки памяти показаны прямоугольниками, разбитыми на элементы, для каждого из которых показан в скобках тип элемента, а далее выражение, по которому к нему можно получить доступ. Верхний уровень, соответствующий самому последнему измерению, является простым одномерным массивом указателей на адреса своих элементов, каждый из которых расположен в своём блоке памяти. Элементами всех уровней, кроме последнего, являются указатели на блоки памяти более низкого уровня. Блоки памяти последнего уровня, наконец, содержат последовательности хранимых в такой структуре данных элементов. Доступ к элементу такой структуры данных составляет последовательное разыменование n уровней указателей, где n — число измерений в массиве. Такую структуру данных (обычно только для двумерного случая, хотя понятие можно расширить) называют ***рваным массивом (jagged array)***.

Сравним два рассмотренных нами подхода:

- Шаговый доступ выделяет ровно один блок памяти размером ровно со все требуемые элементы. В «рваном» массиве сами данные разбиты на множество отдельных блоков, плюс требуется память на хранение всех указателей, заполняющих все измерения, кроме младшего. Таким образом, рваный массив требует дополнительных накладных расходов по памяти сверх самих хранимых данных, в то время как для шагового доступа их нет.
- При шаговом доступе требуется чтение массива шагов или измерений, чтобы вычислить адрес элемента, а при доступе к рваному массиву нужно сделать $n - 1$ операций чтения адресов, по которым необходимо считать адрес следующего уровня, плюс один доступ на чтение или запись к самому элементу. Массив шагов или измерений занимает тривиальный объём, а массивы указателей разбросаны по всей памяти. За счёт оптимизаций, которые может сделать транслятор, и устройства работы кеш-памяти процессора шаговый доступ ощутимо быстрее.
- Код выделения и освобождения памяти под более сложно устроенный «рваный» массив сложнее.
- Оба подхода позволяют работать со структурами данных с переменным числом измерений, но за счёт отсутствия кодирования информации об их количестве в типах используемых данных, шаговый доступ позволяет реализовать это несколько проще.
- Синтаксис доступа к элементу рваного массива является «естественным», он совпадает с таковым для массива определённого без использования динамического времени хранения. При шаговом доступе требуется написание некоторой формулы.
- «Рваный» массив носит такое название потому, что число элементов в каждом отдельном массиве следующего уровня, на который указывает родительский, вовсе не обязано быть одинаковым для всех. Типичным примером является массив строк: это массив указателей на массивы символов, каждый из таких массивов символов является строкой с длиной, не зависящей от других. При использовании двумерного массива символов с шаговым доступом (по сути, прямоугольной таблицы символов) под каждую строку было бы выделено одинаковое количество памяти, т.е. память для строк, которые короче строки максимальной длины, была бы выделена впустую и не использовалась. «Рваный» массив хранит каждую строку такой таблицы в отдельном блоке памяти своей длины, так что правый край таблицы получается образно неровным, «рваным». Таким образом, рваный массив является более гибкой и мощной структурой данных.

8.4. Введение в классовые типы

Рассмотренные примеры работы с динамическим массивом показывают, что он концептуально является тройкой значений: указателем на блок памяти, числом используемых и выделенных элементов. Передача всех этих значений функциям, реализующим алгоритмы над этой структурой данных, особенно при передаче по указателю для возможности изменения их, весьма громоздка. Для решения этой проблемы введём понятие классового типа или просто класса.

Класс (class) — категория типов, которая полностью контролирует своё представление в памяти и семантику операций. В этом разделе мы познакомимся с простейшим использованием классовых типов для задания агрегатов с произвольным содержимым. В отличие от массива, подобъекты которого имеют одинаковый тип, класс может содержать произвольное число подобъектов любого типа, включая 0. В такой ситуации доступ к ним по номерам был бы неудобен, и вместо этого каждый из них имеет собственное имя, задаваемое программистом. Классовый тип является производным типом, но понятие базового типа к нему не применимо. Это минимальные возможности, предоставляемые классовыми типами — их полную функциональность мы будем вводить по частям по мере надобности.

Классовые типы вводятся в программу специальной формой спецификатора типа, состоящей из ***ключа (key)*** классового типа, за которым следует определение содержимого класса в фигурных скобках — это ***спецификатор класса (class specifier)***. Такой спецификатор является определением классового типа. В качестве ключа мы будем использовать ключевое слово `struct`. Между фигурными скобками записывается последовательность описаний, задающих все сущности программы, относящиеся к данному классу. Класс является областью видимости, которую получают все описания, указанные в его определении, поэтому их также называют ***членами (member)*** класса. Вначале мы ограничимся описаниями объектов без

инициализаторов и спецификаторов класса памяти. Эти описания и задают список подобъектов класса с их типами и именами:

```
struct
{
    int x,y;
    unsigned z;
} s1,s2;
```

Данное определение выделяет память под объекты `s1` и `s2` классового типа, содержащие по два подобъекта типа `int`, именуемые `x` и `y`, и по подобъекту типа `unsigned`, именуемому `z`. Так же как элементы с одним и тем же индексом в разных объектах-массивах, пускай даже одинакового типа, — разные объекты, так же и в каждом объекте классового типа свой набор подобъектов с именами, соответствующими описаниям в его определении.

Такая форма классового типа является агрегатом, располагающим подобъекты в памяти по порядку их описаний, поэтому она может быть инициализирована стандартным для агрегатов способом:

```
struct
{
    int a;      // = 1
    struct {
        int b, // = 2
            c; // = 0 (инициализирован по значению - не хватило инициализаторов)
    } d;
    int e;      // = 3
} s = {1,{2},3};
```

Для доступа к подобъектам классовых типов используются две операции: прямой и косвенной выборки. ***Операция прямой выборки `.` (точка) осуществляет доступ к подобъекту указанного объекта классового типа по его имени***. Это инфиксная бинарная операция. Левый операнд этой операции должен иметь классовый тип, а правый должен быть именем одного из его подобъектов. Это имя ищется только в области видимости класса, определяемого типом левого операнда. Результат этой операции есть значение соответствующего подобъекта, его категория значения совпадает с таковой для левого операнда.

Операция косвенной выборки `->` (стрелка) осуществляет доступ к элементу структуры по указателю на неё. Выражение `A->B` эквивалентно `(*(A)).B`, т.е. операция косвенной выборки осуществляет разыменование левого операнда, имеющего тип «указатель на классовый тип», а далее действует аналогично прямой выборке, т.е. является синтаксическим удобством. Она введена в язык, поскольку указатели на классовые типы встречаются часто, а использование операции прямой выборки требует неудобных лишних скобок вокруг разыменования левого операнда из-за большого приоритета операции выборки.

Приведём примеры обращения к членам структур:

```
// Описываемый класс состоит из двух подобъектов:
// x типа int и немодифицируемого y типа unsigned.
// s - объект типа такой структуры,
// ps - указатель на такую структуру.
struct
{
    int x;
    const unsigned y;
} s = {1,2},*ps;

// Присвоить подобъекту x объекта s значение 10.
// Поскольку s - леводопустимое выражение, то и
// s.x тоже, и оно может использоваться в качестве
// левого операнда операции присваивания.
s.x = 10;

// Записать адрес s в ps.
ps = &s;

// Считать значение члена структуры по указателю на неё.
// Выводит 10.
```

```
22  std::cout << ps->x << '\n';
23
24  // ОШИБКА: y описан как немодифицируемый,
25  //         поэтому s.y - немодифицируемое
26  //         (хоть и леводопустимое) выражение.
27  //         Этот подобъект всегда будет сохранять
28  //         своё значение, заданное при инициализации
29  //         всего классового типа.
30  s.y = 42;
31
32  // ОШИБКА: то же самое, что и выше.
33  ps->y = 18;
34
35  // ОШИБКА: имена x и y не описаны.
36  x = y = 0;
```

> При трансляции программы операции доступа к подобъектам классовых типов генерируются аналогично уже рассмотренным агрегатам — массивам. Вместо арифметики указателей к адресу начала структуры сразу прибавляется смещение требуемого члена, известное компилятору по построению её типа.

8.4.1. Имена классовых типов

Каждый раз при использовании спецификатора класса в известной нам форме, создаётся новый тип, даже если содержимое описаний одинаково:

```
1  struct { int a; } x;
2  struct { int b; } y = {0};
3
4  // ОШИБКА: тип y отличен от типа x.
5  x = y;
```

Чтобы иметь возможность говорить об одном и том же классовом типе в нескольких описаниях, можно дать ему имя. В спецификаторе класса после ключа `struct` может указываться идентификатор, который задаёт имя определяемого классового типа с указанным содержимым. Такое определение часто используется без описателей, чтобы ввести в программу новый тип без одновременного создания, например, объектов этого типа для наглядности. Имена классовых типов являются спецификаторами типов и могут использоваться в дальнейших описаниях:

```
1  // Определение классового типа s.
2  // В этом описании только один спецификатор класса
3  // на месте спецификатора типа, и ни одного описателя,
4  // поскольку задача такого описания - введение имени s.
5  struct S
6  {
7      int a;
8      unsigned b;
9  };
10
11 // Объект классового типа с описанным выше содержимым.
12 S x;
13 // То же самое.
14 S y = {1,2};
15
16 // Ошибки нет - тип одинаковый.
17 x = y;
18
19 // Определение классового типа и объекта этого типа в одном описании.
20 struct S2
21 {
22     char a;
23     char b;
24 } u;
```

Есть и другой способ дать классовому типу имя — через псевдоним типа. Этот приём в форме с ключевым словом `typedef` повсеместно применяется в языке C, но в C++ не нужен.

Имя классового типа и другое описание того же имени, не являющееся спецификатором типа, могут быть даны в одной области видимости без конфликтов в любом порядке. При этом имя классового типа является скрытым. Когда имя классового типа скрыто другим именем, не являющимся спецификатором типа, дополнительным способом обойти это скрытие является использование ***детального (elaborated)*** имени класса — такое имя состоит из ключа класса и его имени. Заданное таким образом имя при поиске имён игнорирует все описания, не являющиеся описаниями классовых типов:

```
1  // Определение классового типа.
2  struct S { int x; };
3
4  // Скрывающее описание класса объектом
5  // допустимо, поскольку вводит имя, не
6  // являющееся спецификатором типа.
7  int S;
8
9  // Здесь S относится к объекту.
10 S = 3;
11
12 // Использование ключа struct перед именем S
13 // пропускает в поиске имён описания, не являющиеся
14 // описаниями классовых типов и позволяет найти
15 // скрытое от поиска обычных имён определение класса S.
16 struct S s1,s2;
```

Поскольку классы являются именованными областями видимости, они могут именоваться в частях квалифицированного имени, чтобы получить полные имена подобъектов класса. Поскольку такое имя в известных нам случаях не соответствует никакому объекту, а лишь описывает структуру объектов классового типа, единственная применимая к таким именам операция — `sizeof`:

```
1  struct S
2  {
3      int x;
4  };
5
6  void f()
7  {
8      // Вывод размера подобъекта класса.
9      std::cout << sizeof(S::x) << '\n';
10 }
```

8.4.2. Влияние квалификаторов на структуру и её члены

Как нам известно, квалификаторы типов могут использоваться в описаниях членов классов, кроме того сами значения объектов классового типа могут иметь квалификаторы. При доступе к подобъекту классового типа результат имеет набор квалификаторов, являющихся объединением квалификаторов из описания члена класса и выражения, доступ к члену которого осуществляется:

```
1  struct S
2  {
3      int a;
4      const int b;
5      volatile int c;
6  };
7
8  S x,x2 = {0};
9  const S y;
10
11 // Можно.
12 x.a = 1;
```

```
13
14 // ОШИБКА: член b описан как немодифицируемый,
15 //         поэтому x.b - немодифицируемое.
16 x.b = 1;
17
18 // ОШИБКА: y - немодифицируемое, поэтому и
19 //         y.a - немодифицируемое.
20 y.a = 1;
21
22 // y.c - леводопустимое выражение типа
23 // const volatile int
24 // (const от y, volatile от c)
25
26 // ОШИБКА: x содержит немодифицируемый член,
27 //         который нельзя изменить после создания
28 //         объекта, поэтому и вся структура в целом
29 //         считается не модифицируемой.
30 x = x2;
```

8.4.3. Выравнивание объектов

Мы выяснили, что к объектам классовых типов, помимо специфичной для них операции выборки, применимы в обычном смысле некоторые из уже известных нам операций. Операция `sizeof` тоже работает с классами, но позволяет заметить некоторые неочевидные особенности их устройства.

Рассмотрим пример:

```
1  #include <iostream>
2
3  int main()
4  {
5      struct S
6      {
7          char a;
8          int b;
9          char c;
10     };
11     std::cout << sizeof(S) << '\n';
12     return 0;
13 }
```

Поскольку объект классового типа содержит в себе подобъекты, заданные в определении этого типа, размер его никак не может быть меньше суммы размеров всех подобъектов, в нашем случае (для архитектуры x86) `sizeof(char)+sizeof(int)+sizeof(char) == 6`. Однако программа выводит вдвое большее значение — 12.

Оказывается, что подобъекты внутри классового типа расположены не обязательно непосредственно друг за другом, как элементы массива. Между ними могут находиться ***заполняющие байты (padding bytes)***, которые не используются и имеют неопределённые значения. Это сделано для того, чтобы удовлетворить требования по выравниванию различных типов данных.

Выравнивание (alignment) типа данных — неотрицательная целая степень числа 2, которой должны быть кратны значения адресов всех объектов рассматриваемого типа. Численно большие выравнивания называют более ***строгими (strict)***. Такое требование введено в языке C++ и соблюдается, даже за счёт использования дополнительной памяти под заполняющие байты впустую, исходя из особенностей устройства аппаратных средств. Современные центральные процессоры без проблем осуществляют операции доступа к памяти только тогда, когда между адресом операции и объёмом обрабатываемых за эту операцию данных имеется некоторая связь. Причины этого кроются в организации механизмов доступа к памяти и кеширования. Если это условие не выполняется, то в лучшем случае он оказывается медленнее, чем в выровненном случае (процессор или операционная система выполнила две выровненных операции, смежные с местом не выровненной, и объединила результаты), а в худшем — приводит к аварийному завершению

программы (не выровненный доступ не поддерживается). Конкретное поведение зависит от среды исполнения, но стандарт языка за счёт введения требований на выравнивание объектов ограждает в большинстве случаев программиста от связанных с доступом к не выровненным данным проблем. ***Выравнивание не является эффектом, специфическим для членов классов — ему подчиняются все объекты.*** Рассматриваем мы его только сейчас, потому что до рассмотрения структур мы практически не могли столкнуться с видимыми проявлениями этих ограничений — если только любопытный читатель не обнаружил, что байты заполнения могут присутствовать и в аппаратном стеке между объектами с автоматическим временем хранения.

Попробуем разобраться с тем, где и почему расположены подобъекты в рассматриваемой нами структуре. Изучение, например, ассемблерного листинга покажет, что подобъекты `a`, `b` и `c` расположены по смещениям 0, 4 и 8 байт относительно начала структуры. Это означает, что объект, имеющий такой тип устроен как показано на рис. 8.4 (байты заполнения выделены серым).

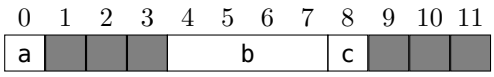


Рис. 8.4: Устройство структуры с байтами заполнения

Как видно, байты заполнения могут находиться как внутри структуры, так и в её конце. При расположении очередного члена транслятор оставляет неиспользуемыми под заполнение столько байтов после предыдущего, чтобы его смещение относительно начала структуры было кратно его выравниванию. При этом сама структура получает выравнивание, равное наибольшему выравниванию из всех её подобъектов, что вместе с рассмотренной схемой размещения обеспечивает необходимое выравнивание для подобъектов на всех уровнях иерархии сложного агрегатного типа. Добавление заполняющих байтов в конце объекта, чтобы его размер стал кратен его выравниванию, позволяет выполнить все требования по выравниванию в массивах структур, где каждая структура-элемент расположена непосредственно вслед за предыдущей. Таким образом, массив не вносит никаких дополнительных требований по выравниванию, и его выравнивание равно выравниванию его элемента.

Выяснить требования по выравниванию того или иного типа можно с помощью операции `alignof`, после которой в круглых скобках записывается имя интересующего типа, аналогично `sizeof` (вариант с выражением в качестве операнда в случае у `alignof` отсутствует). Она возвращает выравнивание своего операнда-типа в виде значения типа `std::size_t`.

Располагая сведениями о выравнивании типов, можно уменьшить число байтов заполнения в структуре, повысив эффективность использования памяти. Например, для рассмотренной выше структуры можно поменять местами два последних члена:

```
1 struct s
2 {
3     char a;
4     char c;
5     int b;
6 };
```

При этом в памяти она примет вид, показанный на рис. 8.5, который занимает 8 байтов в памяти вместо 12 для прошлого варианта.

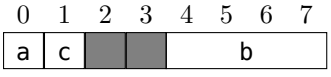


Рис. 8.5: Устройство структуры с байтами заполнения, вариант 2

Все значения выравнивания, поддерживаемые конкретной реализацией во всех контекстах, называются ***основными (fundamental)***. Тип с выравниванием, являющимся

максимальным из основных, определён в виде псевдонима типа `std::max_align_t` в заголовке `cstddef`. ***Для большинства базовых типов, размер которых является степенью числа 2, выравнивание совпадает с размером типа.*** Гарантируется, что символьные типы имеют минимальное выравнивание из возможных (обычно 1, т.е. без ограничений). Дополнительно реализация может в некоторых случаях разрешать применение бóльших выравниваний, называемых ***расширенными (extended)***.

Запросить более строгое выравнивание объекта, чем требуется исходя из его типа, можно с помощью ***спецификатора выравнивания (alignment specifier)***, относящегося к так называемым ***атрибутам (attribute)*** и являющимся исключением из их общего синтаксиса. Мы будем рассматривать его аналогично другим спецификаторам описания. Его синтаксис — ключевое слово `alignas`, за которым в круглых скобках указывается требуемое выравнивание в виде целочисленного константного выражения. Вместо него можно указать имя типа, что эквивалентно использованию выражения в первой форме с операцией `alignof`, применённой к указанному типу. Этот спецификатор может применяться только в описаниях объектов. Его можно применять к членам структуры, что позволяет более гибко управлять её внутренним устройством. Если в описании оказываются несколько спецификаций выравнивания, применяется самая строгая. В качестве исключения, спецификация выравнивания со значением 0 игнорируется полностью, любые другие попытки запросить итоговое выравнивание объекта менее строгое, чем было бы без использования спецификаторов выравнивания, ошибочно — выравнивание можно делать только более строгим. В случаях, когда для объекта имеется несколько описаний, применяются следующие правила: если определение не имеет спецификатора выравнивания, все описания тоже не должны его иметь, если же в определении он присутствует, остальные описания должны иметь такой же или не иметь спецификаторов выравнивания вовсе. Если описания одного объекта с внешней связанностью в разных единицах трансляции имеют разные выравнивания, поведение не определено.

В качестве примера, где может потребоваться такое увеличение естественного выравнивания, можно привести определение массива, работа с которым будет осуществляться с помощью SIMD-инструкций архитектуры x86. Эти инструкции используют так называемые XMM-регистры процессора размером 16 байтов, которые могут трактоваться как одно или несколько значений. Например, они могут рассматриваться и обрабатываться как массив из 4 значений типа `float`. Такой массив в языке C++ будет обладать выравниванием, равным выравниванию своего элемента, т.е. 4. Однако процессор для эффективного обмена с памятью значений таких регистров рассматривает их как 16-байтные величины с соответствующим выравниванием. Таким образом, массив, используемый в таком качестве, следует определять как

```
alignas(16) float x[4];
```

Операция `new` возвращает блоки памяти, адреса которых удовлетворяют любому основному выравниванию.

8.4.4. Использование классовых типов с функциями

Классы, в отличие от массивов, никакими особыми свойствами разложения не обладают и могут передаваться функциям в качестве параметров и возвращаться из них напрямую. Тем не менее, поскольку многие классы имеют немалый объём, копирование такого объёма информации на стек не целесообразно, и ***напрямую обычно передают классы размером не более двух машинных слов***. В других случаях вместо самого значения классового типа передают указатель на объект, хранящий его, так же по указателя в виде out-параметра реализуется и возврат значения классового типа.

Вернёмся к примеру с функциями добавления и вставки в динамический массив. Как уже было сказано, структура данных «динамический массив» состоит из трёх значений: указателя на выделенный блок памяти, его размера и числа используемых элементов. Функциям, работающим с этой структурой данных обычно нужны все или многие из этих значений. Логичным шагом является объединить их все в структуру. Пойдём дальше и вынесем этот классовый тип вместе с функциями его обработки в отдельную единицу трансляции с заголовочным файлом. Задокументируем при этом назначение функций и их параметров в комментариях.

```
1  // int_dynarray.hpp
2
3  #ifndef INT_DYNARRAY_HPP
```

```
#define INT_DYNARRAY_HPP

#include <cstddef>

namespace ida
{
    struct int_dynarray
    {
        int* p;
        std::size_t n,capacity;
    };

    // Инициализация динамического массива в пустое состояние.
    void init(int_dynarray* a);
    // Вставка мест под count элементов, начиная с позиции where, расположенной
    // внутри хранимых данных или примыкающей с конца.
    void insert_space(int_dynarray* a,std::size_t where,std::size_t count);
    // Удаление count элементов, начиная с позиции where.
    void delete_elements(int_dynarray* a,std::size_t where,std::size_t count);
    // Вывод содержимого динамического массива.
    void print(const int_dynarray* a);
    // Возвращает число хранимых в динамическом массиве элементов.
    std::size_t size(const int_dynarray* a);
    // Возвращает значение хранимого элемента с индексом i.
    int read_element(const int_dynarray* a,std::size_t i);
    // Записывает значение value в хранимый элемент с индексом i.
    void write_element(int_dynarray* a,std::size_t i,int value);
    // Освобождает ресурсы динамического массива.
    void destroy(int_dynarray* a);
}
#endif
```

```
// int_dynarray.cpp

#include "int_dynarray.hpp"

#include <cassert>
#include <iostream>

namespace ida
{
    namespace
    {
        inline std::size_t max(std::size_t x,std::size_t y)
        {
            return x>y?x:y;
        }
    }

    void init(int_dynarray* a)
    {
        a->p = nullptr;
        a->n = a->capacity = 0;
    }

    void insert_space(int_dynarray* a,std::size_t where,std::size_t count)
    {
        assert(a->n<=a->capacity);
        assert(where<=a->n);
        if(!count)
            return;
        std::size_t new_n = a->n+count;
        if(new_n>a->capacity){
            a->capacity = max(max(a->capacity*3/2,4),new_n);
            int *new_a = new int[a->capacity];
            for(std::size_t i=0;i<where;++i)
                new_a[i] = a->p[i];
```

```
            for(std::size_t i=where;i<a->n;++i)
                new_a[i+count] = a->p[i];
            delete[] a->p;
            a->p = new_a;
        }else{
            if(where<a->n){
                std::size_t i = a->n-1;
                do
                    a->p[i+count] = a->p[i];
                while(i--!=where);
            }
        }
        a->n = new_n;
    }

    void delete_elements(int_dynarray* a,std::size_t where,std::size_t count)
    {
        assert(where+count<=a->n);
        if(!count)
            return;
        for(std::size_t i=where+count;i<a->n;++i)
            a->p[i-count] = a->p[i];
        a->n -= count;
    }

    void print(const int_dynarray* a)
    {
        std::cout << a->n << '(' << a->capacity << "): ";
        for(std::size_t i=0;i<a->n;++i)
            std::cout << ' ' << a->p[i];
        std::cout << '\n';
    }

    std::size_t size(const int_dynarray* a)
    {
        return a->n;
    }

    int read_element(const int_dynarray* a,std::size_t i)
    {
        assert(i<a->n);
        return a->p[i];
    }

    void write_element(int_dynarray* a,std::size_t i,int value)
    {
        assert(i<a->n);
        a->p[i] = value;
    }

    void destroy(int_dynarray* a)
    {
        delete[] a->p;
    }
}
```

```
// main.cpp

#include "int_dynarray.hpp"

int main()
{
    ida::int_dynarray a;
    ida::init(&a);
    ida::print(&a);

    ida::insert_space(&a,0,6);
```

```
12      for(std::size_t i=0;i<6;++i)
13          ida::write_element(&a,i,i);
14      ida::print(&a);
15
16      ida::insert_space(&a,2,1);
17      ida::write_element(&a,2,100);
18      ida::print(&a);
19
20      ida::delete_elements(&a,4,2);
21      ida::print(&a);
22
23      ida::destroy(&a);
24  }
```

В этом примере мы пошли ещё дальше и оформили полный интерфейс операций над динамическим массивом в виде функций, включая тривиальные чтение и запись элементов. Это может показаться излишним, но тот факт, что основной программе не требуется прямой доступ к членам данных класса означает, что её внутреннее устройство может быть изменено произвольным образом, и при сохранении функционального интерфейса остальные части программы не потребуют изменений. Скрытие деталей реализации сложных объектов называют ***инкапсуляцией (incapsulation)***. Это одно из основных понятий объектно-ориентированного программирования, к рассмотрению которого мы постепенно приближаемся. Пока она обеспечена только честным словом программистов, использующих эту структуру данных, не осуществлять доступ к подобъектам классового типа ей соответствующего из единиц трансляции, кроме той, в которой реализованы алгоритмы её интерфейса — это хрупкое состояние мы постараемся закрепить на уровне языка дополнительными средствами, с которыми познакомимся позднее.

Особо отметим роль функций `read_element` и `write_element` — они поддерживают const-корректность на уровне, который самому языку C++ не соответствует. С точки зрения пользователя данного класса, он владеет своими элементами. В таком случае логично предположить, что у неизменяемого объекта динамического массива элементы изменить нельзя. Однако если просто осуществить доступ к члену `p` объекта типа `const int_dynarray`, будет получено леводопустимое значение типа `int* const`, которое ***позволит*** изменить объекты, адрес последовательности которых оно содержит. Об этом аспекте языка говорят, что const-корректность в языке C++ является ***поверхностной (shallow)*** — не распространяется на нижние уровни типов членов агрегатов. В данном примере при доступе через функции интерфейса за счёт const-корректности гарантируется, что ilcwrite_element может быть вызвана только на изменяемом объекте, что позволяет углубить const-корректность объекта до ожидаемой по логическому смыслу глубины.

8.5. Неполные типы

Неполным (incomplete) называется тип, размер которого не известен. Например, фундаментальный тип `void` относится к таким типам. Над неполными типами и объектами таких типов можно совершать только ограниченное число операций, которые не требуют знания размера и содержания такого объекта. Все они сводятся к работе с адресом этого объекта. В первую очередь к неполным типам не применима операция `sizeof` (по определению), а во-вторых, невозможно определить объект (или описать член структуры) неполного типа — определение выделяет память, чтобы это сделать, надо знать, сколько выделять. Неполный тип (кроме `void`) может быть ***дополнен (completed)*** информацией о размере и содержимом и перестать быть таковым для конкретного объекта или всего типа в целом.

Массив, размер которого не известен (не указан), является неполным типом. В рассмотренных нами ранее ситуациях, где размер массива не указывался, тип был полным: число элементов определялось инициализатором (в описании), или тип разлагался в указатель и массивом не являлся (в параметрах функции):

```
1  // Реальный тип параметра int * - полный.
2  void f(int x[])
3  {
4      // y имеет полный тип int [3].
5      int y[] = {1,2,3};
6  }
```

Рассмотрим пример из нескольких единиц трансляции, где используется неполный тип категории «массив».

```
// Файл tu.hpp

#ifndef TU_HPP
#define TU_HPP

#include <cstddef>

// Описание массива с неполным типом
// и внешней связанностью.
extern int x[];

// Функция, возвращающая корректный
// индекс массива.
std::size_t get_index();

#endif
```

```
// Файл tu.cpp

#include "tu.hpp"

// Совместимое с описанием из заголовочного
// файла определение x, которое дополняет
// его неполный тип int [] до полного int [4].
int x[] = {1,2,3,4};

std::size_t get_index()
{
    return 1;
}
```

```
// Файл main.cpp

#include <iostream>
#include "tu.h"

int main()
{
    // x имеет неполный тип, но для разложения
    // массива в указатель на нулевой элемент
    // знать число элементов в нём не нужно.
    std::cout << x[get_index()] << '\n';
    return 0;
}
```

В этом примере объект, идентифицируемый `x`, описан с неполным типом в заголовочном файле, что ошибки не вызывает, т.к. описание объекта не выделяет под него память. В единице трансляции `tu.cpp` содержится определение этого объекта, связываемое с неполными описаниями, которое дополняет его тип до полного информацией о числе элементов в нём и выделяет под его хранение память. В единице трансляции `main.cpp` тип этого объекта остаётся неполным, но для выполняемых с ним операций в ней информации о размере не требуется.

Больший интерес представляют неполные классовые типы — классы, содержимое которых не известно. Такой случай возникает в результате описания, вводящего имя спецификатора класса без указания его содержимого, такое описание считается для него не являющимся определением. Последующее полноценное определение спецификатора класса может дополнить этот тип для всех сущностей, в которые он входит.

```
void f()
{
    // Неполное описание класса S.
    // Имя S описано, но содержимое этого класса не известно.
    struct S;
```

```
6
7       // Формирование указателя на неполный тип разрешено, т.к.
8       // указатель хранит адрес начала объекта - размер и содержимое ему не важно.
9       S *ps;
10
11      // ОШИБКА: нельзя обратиться к члену структуры,
12      //         содержимое которой не известно.
13      ps->a = 3;
14
15      // Определение, включающее содержимое структуры S, дополняет этот тип.
16      struct S
17      {
18          int a;
19      };
20
21      // Теперь содержимое структуры известно, и обратиться к её члену можно.
22      ps->a = 3;
23  }
```

8.5.1. Абстрактные типы данных и инкапсуляция в стиле C

Показанный выше пример оформления интерфейса класса в виде набора функций, реализующих операции над ним, приводит к созданию ***абстрактных типов данных (abstract data type (ADT))*** — типов, заданных формально в виде семантики операций над ними, без указания их конкретных реализаций.

C++ как язык, напрямую поддерживающий объектно ориентированную парадигму программирования, имеет встроенные средства по ограничению доступа к деталям реализации объекта. Эти средства будут рассмотрены в дальнейшем, а сейчас покажем, как технически реализуется инкапсуляция уже известными нам средствами — так, как это делается в языке C.

Неполные классовые типы являются основой механизма реализации абстрактных типов данных в языке C. При таком подходе определение классового типа переносится из заголовочного файла в единицу трансляции, где реализуется этот интерфейс, а в заголовочном файле оставляют только его неполное описание. Теперь в других единицах трансляции доступ к членам класса невозможен, поскольку его устройство неизвестно, и эта ошибка будет обязательно обнаружена компилятором.

При таком подходе определить объекты неполного типа в других единицах трансляции невозможно, и задача выделения памяти под сам объект перекладывается с определения на функцию инициализации. Поскольку необходимо выделение памяти под произвольное число объектов, которые должны существовать после возврата из этой функции, приходится прибегать к динамическому времнени хранения для выделения отдельных объектов.

Чтобы ещё сильнее показать, что значение, соответствующее указателю на объект, не может быть никак осмысленно использовано, за исключением передачи его в функции интерфейса класса, этот указатель на неполный классовый тип можно скрыть за псевдонимом типа, который с точки зрения пользователя будет олицетворять «значения» данного класса. Такие указатели иногда называют ***непрозрачными (opaque)***. Для соблюдения const-корректности потребуется ввести отдельный псевдвоним для указателя на неизменяемый объект, т.к. добавление const к исходному псевдониму приведёт к иному результату.

Для рассмотренного ранее примера эти изменения выглядят следующим образом:

```
1   // int_dynarray.hpp
2
3   #ifndef INT_DYNARRAY_HPP
4   #define INT_DYNARRAY_HPP
5
6   #include <cstddef>
7
8   namespace ida
9   {
10      // Псевдоним типа, который является типом "динамического массива"
11      // для его пользователя. Это указатель на неполный классовый тип,
12      // который описан в этом же описании за счёт упоминания его с ключом struct.
13      using int_dynarray = struct int_dynarray_t*;
14      // В языке C это обычно выглядит как
15      // typedef struct int_dynarray_t* int_dynarray;
```

```

    // Псевдоним типа "неизменяемый динамический массив".
    // Для пользователей этого псевдонима достаточно знать,
    // что к нему возможно неявное приведение из изменяемого,
    // но не наоборот. Ключ struct уже не нужен - тип int_dynarray_t уже описан.
    using const_int_dynarray = const int_dynarray_t*;

    // Инициализация теперь не принимает указатель на уже созданный
    // объект, а возвращает его, уже инициализированный.
    // В процессе возврата происходит передача владения.
    int_dynarray init();
    // В остальных функциях также используются псевдонимы.
    void insert_space(int_dynarray a,std::size_t where,std::size_t count);
    void delete_elements(int_dynarray a,std::size_t where,std::size_t count);
    // Для указания const-корректности теперь используется псевдоним,
    // т.к. const int_dynarray это int_dynarray_t * const.
    void print(const_int_dynarray a);
    std::size_t size(const_int_dynarray a);
    int read_element(const int_dynarray a,std::size_t i);
    void write_element(int_dynarray a,std::size_t i,int value);
    // При уничтожении теперь также происходит передача владения.
    void destroy(int_dynarray a);
}
#endif
```

```
// int_dynarray.cpp

#include "int_dynarray.hpp"

#include <cassert>
#include <iostream>

namespace ida
{
    // Опеределение содержимого объекта, хранящего
    // данные, соответствующие динамическому массиву
    // теперь только в единице трансляции, где он реализован.
    struct int_dynarray_t
    {
        int* p;
        std::size_t n,capacity;
    };

    int_dynarray init()
    {
        // Выделить память под новый объект,
        // задать его начальное значение и вернуть.
        // Теперь выделение памяти и инициализация
        // выполняются в одном месте, и так как нам
        // нужны нули везде, всё можно сделать
        // инициализацией по значению.
        return new int_dynarray_t{};
        // В более сложных случаях можно было бы
        // всё делать последовательно:
        // int_dynarray a = new int_dynarray_t;
        // a->p = nullptr;
        // a->n = a->capacity = 0;
        // return a;
    }

    void destroy(int_dynarray a)
    {
        delete[] a->p;
        // Уничтожения объекта теперь ответственно
        // за освобождение не только памяти, владеемой
        // объектом, но и за его собсвтенную.
        delete a;
```

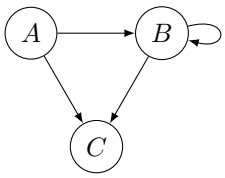


Рис. 8.6: Пример изображения орграфа на плоскости

```
43      }
44
45      // Реализация остальных функций идентична.
46  }
```

Такой подход несколько сложнее в реализации и может быть медленнее за счёт дополнительных динамических выделения памяти и того, что теперь любые обращения к объекту идут через указатель, поэтому не все библиотеки на языке C используют его — некоторые оставляют инкапсуляцию на формальном уровне.

8.6. Графовые структуры данных

До сих пор мы рассматривали структуры данных, в которых в каждом блоке памяти хранилось несколько однотипных элементов в виде массива. Перейдём к рассмотрению структур данных, где каждый хранимый элемент рассматривается независимо от остальных и может располагаться в отдельном блоке динамической памяти. В таком случае типом данных такого элемента будет являться класс, содержащий, помимо самих данных элемента, некоторое количество указателей на тот же (или другой) классы. Описание такого класса возможно, поскольку имя класса становится (неполно) описанным сразу после его упоминания после ключевого слова `struct`, поэтому в описании содержимого класса возможно задание члена типа «указатель на класс, в котором это описание содержится». Допустим именно указатель, член просто типа самого класса невозможен технически, поскольку до завершения описания содержимого этот тип является неполным, и противоречит здравому смыслу.

В общем случае такая структура данных является набором независимо выделенных в динамической памяти элементов, также называемых ***узлами (node)*** содержащих указатели друг на друга. За счёт хранения элементов в отдельных блоках памяти такие структуры данных хуже взаимодействуют с кеш-памятью, но не требуют одного большого непрерывного блока памяти в адресном пространстве программы. В большинстве случаев все операции изменения внутреннего устройства таких структур данных не приводят к изменению адресов памяти элементов в ней хранящихся, за исключением явно затрагиваемых. Это свойство может быть полезным для упрощения взаимодействия с такой структурой.

Рассматриваемые структуры данных могут рассматриваться как представления ориентированных ***графов***. ***Ориентированный граф (орграф) (directed graph (digraph))*** — упорядоченная пара $G = (V, A)$, где V — множество ***вершин (vertex)*** или ***узлов (node)***, а A — множество упорядоченных пар вершин, называемых ***дугами (arc)*** или ***ориентированными рёбрами (oriented edges)***. Граф является математической моделью множества объектов (вершин) и связей между ними (дуг). Например, орграф $G = (\{A, B, C\}, \{(A, B), (B, B), (B, C), (A, C)\})$ можно изобразить в виде рис. 8.6, где вершинам соответствуют окружности с их именами, расположенные на плоскости, а дугам — стрелки, соединяющие их.

Применительно к реализации структуры данных в языке C++, вершина является объектом, а наличие дуги из вершины A в вершину B соответствует хранению в одном из членов-указателей (или иначе связанных с ним объектов) объекта A адреса объекта B. ***Граф (graph)*** $G = (V, E)$ отличается от орграфа тем, что пары вершин E не упорядочены и называются ***рёбрами (edge)***. Такие связи можно трактовать как двунаправленные, в структурах данных они реализуются в виде пары дуг, начало одной из которых совпадает с концом другой и наоборот. Графически их изображают либо в виде линий без стрелок, либо в виде пар дуг, мы будем пользоваться последним представлением, поскольку оно отражает реализацию дуг в виде указателей. Дуги или ребра, у которых начало и конец совпадают ((B, B) на рис. 8.6), называется ***петлями (self-loop)***. Граф (ориентированный или нет), в котором допустимы несколько идентичных дуг, называется ***мультиграфом (multigraph)***, вместо множества дуг или рёбер он включает их ***мультимножество (multiset)***: упорядоченную пару (A, m), где $m : A \to \mathbb{N}$ — функция, сопоставляющая каждому элементу мультимножества

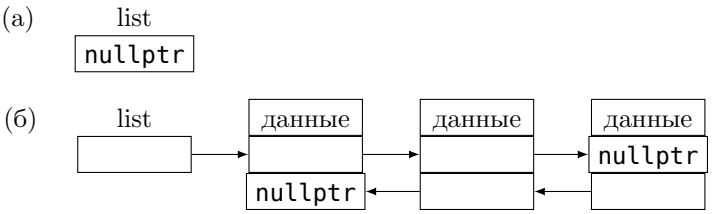


Рис. 8.7: Двусвязные списки.

натуральное число, называемое его ***кратностью (multiplicity)***. В мультиграфах, таким образом, возможны кратные дуги. Некоторые крайние случаи в структурах данных приводят к возникновению с формальной точки зрения кратных дуг и петель.

Для представления дуг, исходящих из вершины, могут использоваться разные подходы:

- Если число дуг фиксировано, соответствующее количество членов (или массив указателей) может быть включено в тип-структуру элемента.
- Если число дуг переменно, но не превышает известной малой величины, можно поступить как и в предыдущем случае, выделив в структуре место под максимально возможное число дуг.
- Для большого, включая неограниченное, количества дуг может использоваться отдельный динамический массив.
- Конкретные структуры данных могут допускать альтернативные представления, с которыми работать проще.

Асимптотическую сложность операции выделения динамической памяти оценить непросто. При анализе динамических структур данных в простом случае предполагают, что она не более амортизированной $O(1)$, хотя скрываемый асимптотической нотацией коэффициент может быть немалым.

8.6.1. Связанные списки

Простейшей графовой структурой является ***двусвязанный список (doubly linked list)***: каждый элемент хранит в себе указатель на следующий и предыдущие элементы хранимой последовательности, один из двух указателей в крайних элементах является нулевым. Сам объект, представляющий эту структуру данных, хранит только указатель на начальный элемент списка, или нулевой указатель, если список пуст. Этот указатель обычно обозначают `head`, а указатели на предыдущий и последующий элементы — `prev` и `next` соответственно. На рис. 8.7 показано устройство двух таких списков: пустого (а) и с тремя элементами (б). Обратите внимание, что речь идёт о связях между узлами, каждый из которых является классом из нескольких членов, поэтому концы стрелок на рисунке указывают не на отдельный член, а на весь узел.

Такая структура данных, как и динамический массив, может хранить последовательность элементов в заданном порядке не ограниченной теоретически длины. ***Основным недостатком связанного списка является то, что поиск элемента по индексу в такой структуре данных имеет не константную, а линейную сложность*** — требуется заданное количество раз перейти по указателю к следующему элементу, начиная с начального. Тем не менее, найти предыдущий или последующий элементы относительно данного можно за константную сложность, просто пройдя по соответствующему указателю. В отличие от динамического массива, связанный список всегда занимает столько памяти, сколько требуется для его элементов, выделенного, но не используемого места в нём нет. С другой стороны, помимо самих данных элементов, в нём хранятся ещё и указатели на соседние элементы, а динамический массив хранит только полезные данные. ***Имея указатель на элемент, перед или после которого нужно вставить новый, вставку можно осуществить за время*** $\Theta(1)$: для этого достаточно выделить память под хранение нового узла и изменить значения некоторых указателей этого узла и его соседей, которых всегда не более двух, независимо от числа элементов во всей структуре данных. Аналогичную сложность имеет и удаление элемента. Это — основные достоинства связанного списка, динамический массив имеют схожую сложность операций вставки и удаления только в конец, а не в любое место.

Существуют различные вариации этой структуры данных:

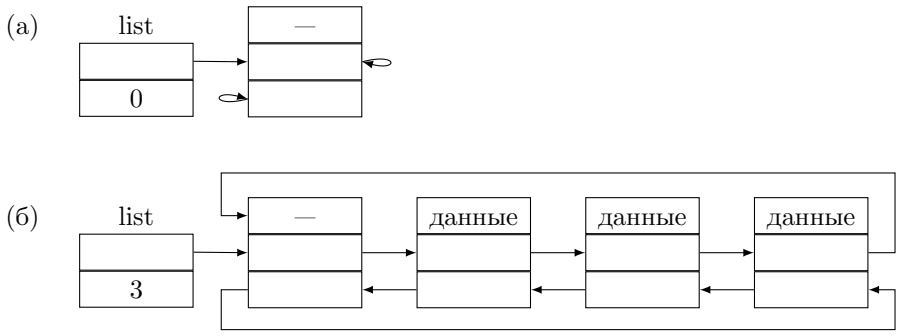


Рис. 8.8: Двусвязные циклические списки с ограничителями и хранением числа элементов.

- В ***Односвязном списке (singly linked list)*** каждый узел содержит указатель только на следующий элемент. Это позволяет сэкономить память в объёме один указатель на хранимый элемент, но поскольку операциям вставки и удаления необходимо будет изменить значение указателя в предыдущем узле, потребуется либо дополнительно отслеживать его или найти, просканировав весь список от начала и увеличив сложность этих операций до $O(n)$. Также константную сложность по перемещению к соседнему элементу сохраняют в такой структуре данных только перемещения вперёд, но не назад. Односвязный список является ***рекурсивной (recursive)*** структурой данный: каждый его элемент содержит указатель на по сути ещё один односвязный список. Это может упростить реализацию некоторых алгоритмов и позволить, например, хранить одинаковые «хвосты» нескольких списков в памяти всего в одном экземпляре, сославшись на него из любого числа других списков.
- В ***кольцевых (circular)*** связанных списках первый и последний элементы не содержат нулевых указателей, а ссылаются друг на друга, замыкая цепочки указателей в кольцо (элемент кольцевого списка из одного элемента ссылается сам на себя). Этот вариант может быть естественным для структур данных, циклических по смыслу. Он также позволяет просмотреть все элементы списка, начиная с любого, даже если список односвязный.
- Связанный список с ***ограничителем (sentinel)*** всегда содержит минимум один фиктивный элемент. Он имеет тот же тип, что и остальные узлы, но его полезные данные не используются. Его наличие позволяет несколько упростить алгоритмы операций за счёт исключения частных случаев с нулевыми указателями за счёт дополнительного места под хранение одного узла.
- Подсчёт числа элементов в обычном связанном списке потребует его полного обхода и имеет сложность $\Theta(n)$. Можно, помимо указателя на начальный элемент списка, сохранить их число в объекте, представляющем сам список. За счёт небольших накладных расходов по обновлению этого значения в операциях вставки и удаления, можно снизить сложность операции вычисления размера списка до $\Theta(1)$. К сожалению, при этом увеличится до линейной сложность операции переноса цепочки узлов между списками, если её длина заведомо не известна.

Примеры двусвязных кольцевых списков с ограничителями и хранением числа элементов, аналогичные рассмотренным выше, приведены на рис. 8.8.

Все опции устройства связанного списка имеют как свои плюсы, так и минусы (включая не рассмотренные нами), поэтому их следует выбирать, исходя из требования решаемой задачи.

Как и динамический массив, связанный список может лежать в основе реализаций стека, очереди и других линейных структур данных. Связанный список применяется чаще всего не в чистом виде, а как часть более сложной структуры данных.

8.6.2. Хеш-таблицы

Хеш-таблицы применяют в случае, когда от структуры данных, хранящей элементы, не требуется сохранения порядка элементов. Так структура данных соответствует понятию динамически меняющегося множества: можно добавлять в неё и извлекать из неё элементы, а также проверять, находится ли в ней элемент с заданным ключом, и получить к нему доступ. Для простоты начнём рассмотрение именно с множеств, а не мультимножеств, т.е. запретим наличие нескольких элементов с одним ключом.

Если всё множество ключей достаточно невелико, то организовать хранение такого множества можно, пронумеровав все возможные ключи некоторым образом, начиная с нуля, и используя массив указателей на элементы размером с общее число ключей. В таком случае каждый элемент массива указывает на элемент, если таковой входит в множество, или в нём хранится нулевой указатель. Для совсем малых по объёму типов элементов, когда среди их значений можно выделить одно особое, вместо указателей можно напрямую хранить элементы в массиве, помечая отсутствующие выделенным значением. Такая структура данных называется ***таблицей с прямой адресацией (direct-access table)***, её реализация труда не представляет. Она позволяет выполнять все требуемые нами операции за константное время. К сожалению, в большинстве случаев множество всех ключей настолько велико, что таблица с прямой адресацией не поместится ни в какую память. Даже если это не так, обычно число элементов, содержащихся во множестве, много меньше всего числа элементов во множестве ключей, и огромное количество памяти в таком представлении будет расходоваться впустую, поэтому эта структура данных имеет довольно ограниченное применение.

Хеш-таблица (hash table) является развитием идеи таблицы прямого доступа, позволяющим сократить объём требуемой памяти до приемлемого и при этом сохранить сложность операций в рамках $O(1)$. Для хеш-таблиц, правда, это сложность в среднем, а не в худшем, как для таблицы прямого доступа, случае. Основной идеей является уменьшение числа элементов массива путём использования не напрямую номера ключа, а некоторой функции от него. Каждый такой элемент массива в хеш-таблице называется ***ячейкой (bucket)***.

Пусть U — множество всех возможных ключей. Определим функцию $h : U \to \{0, 1, \ldots, m-1\}$, отображающую множество всех возможных ключей в m целых значений, начиная с 0, $m < |U|$. Будем использовать эту функцию для выбора ячеек, которым соответствуют хранимые элементы с заданными ключами. Функцию h называют ***хеш-функцией (hash function)***, а процесс её вычисления — ***[хешированием] (hashing)***. Наличие семейства хеш-функций для различных m позволяет сократить размер массива до любого нужного нам размера.

Очевидно, что в таком случае несколько различных ключей могут соответствовать одной ячейке — такая ситуация называется ***коллизией (collision)***. Предотвращение их по возможности и обработка в неизбежных случаях — основные детали реализации хеш-таблиц.

Одним из способов борьбы с коллизиями является правильный выбор используемой хеш-функции. Хорошая хеш-функция должна стремиться к тому, чтобы для каждого ключа выбор ячейки был равновероятен из всех m вариантов, независимо от того, сколько других ключей соответствует той же ячейке. В общем случае удовлетворить этому условию трудно, хотя, если распределение используемых ключей в конкретном случае известно, хеш-функцию можно подстроить под него. Например, если ключом являются равномерно распределённые на полуинтервале $[0; 1)$ вещественные числа, то хеш-функция $h(k) = \lfloor km \rfloor$ удовлетворяет указанному свойству.

В общем случае можно постараться определить хеш-функцию таким образом, чтобы её значения минимально зависели от закономерностей во входных данных. Одним из таких подходов является метод умножения:

$$h(k) = \lfloor m\{kA\} \rfloor, \tag{8.4}$$

где $\{x\}$ — дробная часть x и $0 < A < 1$ — некоторая константа. В [4] предлагается $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$, на практике это значение даёт хорошие результаты. Этот метод также выгоден тем, что одинаково хорошо работает для любого значения m.

При нумерации ключей необходимо каким-то образом преобразовать значение ключа в целочисленное числовое, что можно сделать разными способами, при этом следует учитывать особенности хранимых ключей. Например, при хранении таблицы идентификаторов в памяти транслятора, не следует использовать для вычисления хеш-функции малое начальное число символов, поскольку в библиотеках часто используется большое количество идентификаторов с одинаковыми префиксами.

В общем случае, поскольку $m < |U|$, коллизии неизбежны. Рассмотрим способ их обработки, называемый ***методом цепочек (chaining)***. Идея метода проста — каждой ячейке соответствует связанный список элементов, хранимых в хеш-таблице с соответствующим значением хеш-функции. С хеш-таблицей связано значение, называемое ***коэффициентом заполнения (load factor)*** $\alpha = n/m$, где n — число хранимых элементов в хеш-таблице. Для хорошей хеш-функции за счёт равномерного распределения элементов по ячейкам, это значение соответствует средней длине списка, связанного с каждой ячейкой. За счёт увеличения числа ячеек таблицы (путём создания новой с повторным занесением всех элементов)

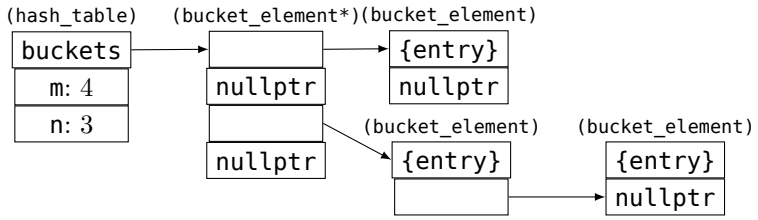


Рис. 8.9: Устройство хеш-таблицы в примере, не поддерживающем удаление элементов.

при превышении выбранного коэффициента заполнения можно поддерживать его не более желаемого, и тем самым обеспечить амортизированную константную сложность операций с хеш-таблицей, поскольку число шагов в просмотрах списка будет зависеть от α, но не от n. Такая операция называется ***перехешированием (rehashing)***. При геометрическом росте m операция перехеширования, имеющая сложность $\Theta(n)$, будет, как и в случае с динамическим массивом, не влиять на асимптотическую сложность операций кроме внесения амортизированности. При удалении большого числа элементов из хеш-таблицы её можно перехешировать на меньшее число m и тем самым уменьшить расход неиспользуемой памяти.

Пример, использующий хеш-таблицу с цепочками в виде односвязных списков, показан на рис. 8.9. В этом примере 4 ячейки и 3 элемента, занесённых в хэш-таблицу: один в нулевой ячейке, два — в третьей.

Основным свойством хеш-таблицы является константное амортизированное время операций над ней в среднем. В то же время в худшем случае (значение хеш-функции для всех ключей одинаково) она вырождается в один связанный список с линейной сложностью операций.

Другим способом представления множества значений с одним значением хеш-функции является ***открытая адресация (open addressing)***. В этом случае каждая ячейка таблицы хранит не более одного элемента, но для каждой из них определена последовательность, в которой должны просматриваться остальные ячейки таблицы, если искомое значение в начальной не найдено. Эта ***последовательность поиска (probe sequence)*** может быть задана различными способами, начиная от просто последующих ячеек по порядку, до повторных вычислений одной или нескольких функций хеширования. Открытая адресация несколько усложняет операцию удаления из хеш-таблицы, поэтому в случаях, когда такая операция требуется, метод цепочек проще.

В случае, когда множество ключей, хранимых в хеш-таблице, известно заранее, можно избежать коллизий полностью и построить структуру данных со сложностью поиска в точности $\Theta(1)$ (без амортизированности). Такой метод называется ***идеальным хешированием (perfect hashing)***.

Наличие у хеш-таблицы худшего случая с нежелательно большой асимптотической сложностью может быть использовано злоумышленниками в атаках, если им известна используемая хеш-функция. Для борьбы с этим применяют ***универсальное хеширование (universal hashing)*** — выбор случайной хеш-функции из специального большого класса, чтобы злоумышленник не мог знать, какой набор входных данных является наихудшим для данной программы.

8.6.3. Деревья

Путь (path) длины n в графе $G = (V, E)$ из вершины v_0 в вершину v_n — упорядоченная последовательность вершин $(v_0, v_1, \ldots, v_n)$, $v_i \in V$, для которой существуют все рёбра вида (v_{i-1}, v_i), $i = 1, 2, \ldots, n$. ***Простой (simple)*** путь — путь, в котором все вершины разные. Пути из трёх и более вершин, в которых начало совпадает с концом, называются ***циклами (loop)***. Граф называется ***ациклическим (acyclic)***, если он не содержит циклов. Одна вершина называется ***достижимой (reachable)*** из другой, если существует путь из второй в первую (вершина считается достижимой из самой себя). Граф называют ***связанным (connected)***, если каждая его вершина достижима из каждой. Эти понятия могут быть расширены и на ориентированные графы.

Дерево (tree) — ациклический связанный граф. Также под этим термином будем понимать графовую структуру данных, связи которой удовлетворяют данному определению. ***Корневое (rooted)*** дерево — дерево, в котором одна из вершин выделена и называется ***корнем (root)***. Для каждого узла в дереве существует один простой путь из вершины в него. Максимальная

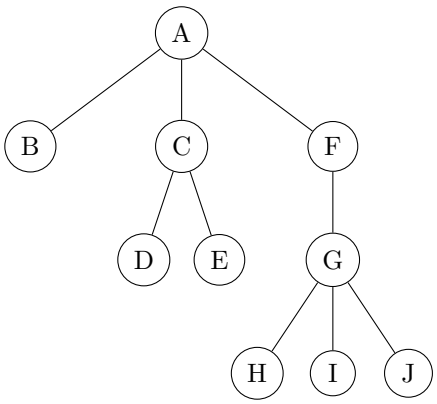


Рис. 8.10: Дерево.

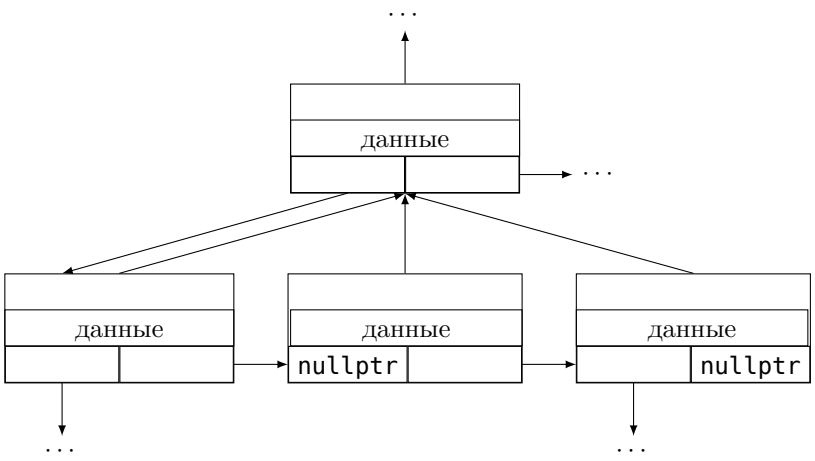


Рис. 8.11: Хранение дерева в памяти с применением связанных списков.

длина таких путей в дереве называется его ***высотой (height)***. Все узлы на таком пути в вершину (включая её саму), называются её ***предками (ancestor)***. Если v_1 — предок v_2, то v_2 — потомок v_1. Предпоследняя вершина такого пути называется ***родительской (parent)*** для рассматриваемой, а она сама — ***дочерней (child)*** для родительской (корневая вершина не имеет родителя и не является дочерней ни для какой). Дочерние элементы одного узла называются ***родственными (sibling)***. Узлы, не имеющие дочерних, называются ***листьями (leaf)***. ***Поддеревом (subtree)*** с корнем в указанной вершине называется часть дерева, состоящая из потомков данной вершины.

Пример дерева показан на рис. 8.10 (в графическом представлении корень дерева обычно располагают сверху). A — корень дерева высоты 3. A, F и G — предки G, из них F — родитель. G — дочерний элемент F, для которого потомки — это G, H, I и J. B, C и F — родственники. B, D, E, H, I, J — листья. Множество вершин C, D и E образует поддерево с корнем в C.

К хранению деревьев в памяти применимы все утверждения, касающиеся хранения произвольных графовых структур. Сама структура данных хранит в простейшем случае только указатель на корневой элемент. Поскольку дерево — ненаправленный граф, каждый узел, помимо полезных данных, хранит указатели на родительский и дочерние. Когда число дочерних элементов в дереве не ограничено (или этот предел слишком велик), можно представить дочерние элементы каждого узла в виде связанного списка. Для простого односвязного списка узел в таком дереве хранит три указателя: на родительский элемент, на первый дочерний и на следующий родственный (рис. 8.11). Таким образом, для деревьев любой сложности можно обойтись узлами с тремя указателями без дополнительных структур данных, хотя это не обязательно самое эффективное представление.

В ***позиционном (positional)*** дереве все дочерние элементы, различают, назначая им индексы из натуральных чисел. У узлов позиционного дерева может независимо от других присутствовать или иметься в наличии дочерний с данным индексом. Если имеется верхняя граница N на индекс дочерних элементов каждого узла, которые могут присутствовать, такое дерево называют ***N-арным (N-ary)***.

	Порядок элементов	Индексация	Поиск	Вставка/удаление
(Дин.) массив	заданный	$\Theta(1)$	$O(n)$, $O(\log n)$ в сорт.	O(n), O(1) аморт. в конце
Связ. список	заданный	$O(n)$	$O(n)$	$\Theta(1)$
Хеш-таблица	неопределённый	—	аморт. $O(1)$	аморт. $O(1)$
Дерево дв. поиска	отсортированный	—	$O(\log n)$	$O(\log n)$

Таблица 8.1: Характеристики основных структур данных.

Бинарное дерево (binary tree) — N-арное дерево для $N = 2$, каждый его узел может имееть левый и правый дочерний. Бинарное дерево может быть определено рекурсивно: это пустое множество узлов или корневой узел с левым и правым бинарными поддеревьями.

Бинарное дерево поиска (binary search tree) — бинарное дерево с элементами в узлах, каждый из которых удовлетворяет следующему свойству: все элементы левого поддерева имеют значение ключа меньше либо равное значению рассматриваемого узла, а из правого поддерева — больше либо равное. В таком дереве можно осуществить поиск элемента за время, не превышающее его высоту: можно двигаться от корня дерева к листьям, выбирая левое или правое поддерево на основании сравнения искомого ключа с текущим узлом, пока искомый элемент не будет найден, либо мы не дойдём до неудовлетворяющего нас листа. Эта операция идейно напоминает двоичный поиск, но сохраняет преимущества и недостатки графовых структур данных. С той же сложностью можно реализовать операции нахождения минимального и максимального элемента.

В вырожденном случае (цепочка из узлов с одним поддеревом) дерево превращается в связанный список, поэтому требуется ограничивать его высоту, чтобы сохранить логарифмическое время операций. Для этого существуют разные подходы, например, красно-чёрные или AVL-деревья. Основная их идея заклчюается в том, что для одного и того же набора элементов существует несколько деревьев, удовлетворяющих определению дерева двоичного поиска. При вставке или удалении элемента, таким образом, можно произвести некоторые преобразования структуры дерева, чтобы ограничить высоту дерева. Все эти подходы ограничивают высоту дерева значением, пропорциональным логарифму от числа элементов в нём, так что ***асимптотическая сложность основных операций с деревом двоичного поиска*** — $O(\log n)$.

К сожалению, соответствующие алгоритмы и их теоретические основы достаточно громоздки и в этом тексте рассматриваться не будут. Автор призывает читателей отметить основные свойства этой структуры данных, которые понадобятся, когда мы столкнёмся с готовыми реализациями в языке C++.

8.7. Сравнение динамических структур данных

Асимптотические сложности основных операций с рассмотренными нами динамическими структурами данных приведены в таблице 8.1.

Как видно из представленной таблицы, структуры данных, идеальной по всем параметрам не существует. Перечислим основные критерии выбора структуры данных, которые следует применять при решении задач.

- ***Массивы***, динамические или нет, являются самой простой структурой данных в реализации, которые предпочтительны по этой причине везде, где нет явных преимуществ у других структур данных. Массив, по сравнению с остальными рассматриваемыми структурами данных — единственная, поддерживающая индексацию за константное время, и применяется там, где такая операция требуется. Сортированный массив предпочтительнее дерева в случае, когда требуется сортированный порядок и число изменений элементов не велико, поскольку он имеет минимальные издержки по памяти. Для случаев, когда требуется много вставок или удалений в середину, это может быть не самая подходящая структура данных. Тем не менее, её всегда следует рассматривать, поскольку она обычно наилучшим образом взаимодействует с системой кеширования памяти.
- ***Связанный список*** — структура данных, применяемая тогда, когда элементы требуется хранить в заданном порядке и необходимы частые удаления и вставки из произвольных мест. Это нечастая ситуация, и как графовая структура данных она имеет плохую производительность на практике, так что редко обоснована.
- ***Хеш-таблицы*** — основная структура данных, применяемая в случаях, когда требуется поиск элементов по ключу. Её проблемой является то, что её хорошие скоростные

характеристики сильно зависят от качества реализации хеш-функции. Кроме того, рассмотренный нами способ устранения коллизий использует графовую структуру данных — связанный список. Если от хеш-таблицы требуется максимальная производительность, следует рассмотреть другие варианты обработки этих ситуаций.

- ***Деревья двоичного поиска*** применяются в случаях, когда требуется максимальная скорость вставки и удаления в последовательности упорядоченных элементов. Если порядок не важен, хеш-таблица предпочтительней, если имеется подходящая хеш-функция. Если вставок и удалений мало, сортированный массив проще в реализации и занимает меньше места в памяти.

Читатель, вероятно, уже отметил, что наши реализации динамических структур данных работают с конкретным типом элемента. Для программы, использующей несколько типов элементов, размножение реализаций одной и той же структуры данных, отличающихся только типом элемента нецелесообразно. Эта проблема получит удовлетворительное решение средствами обобщённого программирования языка C++.

8.7.1. Комбинирование структур данных

Рассмотренные нами хеш-таблицы являются, по сути, комбинированием динамического массива ячеек и связанных списков. Выделенные нами четыре основные структуры данных также можно комбинировать для получения более сложных. Рассмотрим несколько примеров:

- ***Развёрнутый список (unrolled list)*** — связанный список, в каждом элементе которого хранится массив из фиксированного числа m элементов. Работу с этой структурой данных можно организовать таким образом, что каждый массив, кроме максимум одного, заполнен как минимум наполовину — для этого в операциях вставки и удаления осуществлять необходимые переносы хранимых значений между элементами списка, их разбиение и слияние. Такая структура данных имеет те же асимптотические сложности операций, что и обычный связанный список, но с точки зрения точного числа операций в m раз быстрее и во столько же раз имеет меньше накладных расходов. Это особенно полезно, если размер хранимых значений особенно мал. Подобрав m таким образом, чтобы элемент списка имел размер, кратный размеру кеш-линии процессора, можно значительно улучшить взаимодействие такой структуры данных с кешем относительно обычного связанного списка.
- Хранение списка элементов с одним ключом в виде «значения» хеш-таблицы, как мы упоминали в соответствующем разделе, есть способ приспособить её для хранения мультимножества значений относительно их ключей.
- Если имеется множество пар значений, и требуется осуществлять быстрый поиск как по первому в роли ключа вторых, так и по вторым первых, можно сохранить сами пары в список или массив, и построить две хеш-таблицы, содержащие указатели на эти пары, хешируемые по двум разным ключам, связанным с обоими значениями соответственно.

Для многих прикладных задач прямого комбинирования готовых реализаций рассматриваемых структур данных достаточно. В случаях, когда это не так, часто не требуется разрабатывать принципиально новую структуру данных с нуля — достаточно модифицировать одну из имеющихся вместе с алгоритмами операций над ней. Такие структуры данных называют ***расширенными (augmented)***.

На этом мы закончим теоретическое обсуждение структур данных. Их готовые реализации мы встретим в стандартной библиотеке языка C++, где нам пригодятся сведения о их внутреннем устройстве для правильного применения. Различные примеры их комбинирования будут встречаться нам на протяжении всей оставшейся части книги.

Для интересующихся более глубокими теоретическими знаниями о структурах данных и алгоритмах работы с ними, можно порекомендовать книгу [7] — автор рекомендует именно оригинал, а не перевод.

8.8. Задачи на динамические структуры данных

Помимо стандартных требований предыдущих задач, начиная с данной задачи максимальные объёмы обрабатываемых данных не указываются, поскольку теперь нам известны средства работы со всей доступной памятью. Напомним ещё раз, что для корректного представления количеств и размеров элементов, охватывающего всё доступное реализации адресное

пространство, необходимо повсеместно использовать тип `std::size_t` или аналогичный по ширине.

Учтите, что даже в этом режиме программа получает на вход все символы вводимой строки только после нажатия Enter в её конце.

В рамках этой задачи также потребуется продемонстрировать умение использовать нескольких единиц трансляции и оформлять их интерфейсы в виде заголовочных файлов.

Данная задача решается в несколько этапов:

1. Этап 1: согласование интерфейса структуры данных.

   Выберите структуру данных, требуемую для наиболее эффективного решения данной задачи из рассмотренных линейных и многомерных. Дайте определение классового типа, соответствующего ей. Исходя из описания семантики требуемых функций запишите их описания. Все эти функции должны в качестве первого параметра принимать указатель на обрабатываемую структуру данных.

   Каждая структура данных имеет минимум один способ инициализации, который должен быть реализован функцией `init` (начальное состояние указано в конкретных вариантах, в случае нескольких таких функций имена должны начинаться с `init_`), и функцию `destroy`, освобождающую всю используемую структурой данных динамическую память, помимо перечисленных в каждом варианте функций. Для каждого описания приведите в комментарии перед ним документацию, объясняющую семантику параметров и возвращаемого значения функции. Обратите особое внимание на отражение фактов передачи владения при работе с указателями (отсутствие передачи владения можно подразумевать по умолчанию).

   Поместите определение структуры данных и работающих с ней функций в именованное пространство имён в заголовочном файле, который является результатом вашей работы на этом этапе.

2. Этап 2: реализация структуры данных и самой задачи.

   Реализуйте алгоритмы обработки структуры данных, описанные на прошлом этапе, в отдельной единице трансляции (имя должно совпадать с заголовочным файлом). Скройте реализацию вашей структуры данных с использованием неполных классовых типов.

   С помощью имеющегося интерфейса и его реализации решите задачу из своего варианта.

   Прямой доступ к членам данных классового типа из единиц трансляции, кроме той, в которой содержится реализация обрабатывающих его алгоритмов, запрещена! Если вы реализовали инкапсуляцию верно, этого сделать не удастся. Если вам кажется, что без этого не обойтись, согласуйте предварительно изменения в интерфейсе структуры данных. Эти интерфейсы изначально выбраны как более общие, чем требуется для решения конкретной задачи, но принципиальных затруднений вызывать не должны.

   Следующие этапы этой задачи будут даны в следующих разделах.

8.8.1. Варианты

0. Вводятся неотрицательные целые числа в десятичной системе счисления по одному на строку. Порядок их значений ничем не ограничен. Вывести в выходной файл сумму всех этих чисел.

   Пример ввода:

   ```
   1278936325783254390247237489237 4
   2323784263462390090010726361236271831234124534152 34
   2163178236123444453
   ```

   Соответствующий вывод:

   ```
   2323784263462390090138619993814618902041209517520 61
   ```

   Требуемый интерфейс для классового типа «неотрицательное целое число произвольной длины»:

   - Задание начального состояния, равного нулю.
   - Прибавление к данному объекту-числу другого числа, заданного его записью в виде нуль-терминированной строки.

- Вывод значения указанного объекта-числа.

1. Вводится последовательность инструкций по перемещению. Каждая инструкция состоит из буквы-направления (по первым буквам слов North, South, West, East) и положительного целого числа, задающего число шагов в данном направлении. Вместо инструкции по перемещению может быть введён восклицательный знак, при чтении которого необходимо вывести в выходной файл псевдографическое изображение пройдённого к этому моменту пути. Изображение должно занимать минимально необходимое место.

   Пример ввода:

   ```
   N5
   E10
   S3
   !
   W15
   S5
   !
   ```

   Соответствующий вывод:

   ```
   ^>>>>>>>>>>
   ^          v
   ^          v
   ^          v
   ^
   O

        ^>>>>>>>>>>
        ^          v
        ^          v
   <<<<<<<<<<<<<<v
   v    ^
   v    O
   v
   v
   v
   ```

   Требуемый интерфейс для классового типа «матрица символов»:

   - Задание начального состояния в виде матрицы 1×1 из одного указанного символа.
   - Возвращение ширины матрицы.
   - Возвращение высоты матрицы.
   - Запись данного символа по указанным координатам.
   - Чтение символа по указанным координатам.
   - Расширение матрицы на указанное число строк или столбцов в указанном направлении (одном из четырёх) с заполнением указанным символом.
   - Вывод матрицы.

2. Вводится последовательность предложений, оканчивающихся точкой. Предложения могут занимать более одной строки. По окончании их ввода вывести все введённые предложения ровно по одному на строку, заменяя переводы строк в исходном файле одним пробелом. Предложения должны выводиться в порядке увеличения их длин (порядок предложений одной длины может быть любым).

   Пример ввода:

   ```
   Alice's Adventures in Wonderland is a children's
   novel. The title is usually shortened to Alice in
   Wonderland. Charles Lutwidge Dodgson wrote the book.
   He wrote it using the pen name Lewis Carroll. John
   Tenniel drew the 42 pictures in the book. The book
   was published by Macmillan and Co in London. It was
   released on 26 November 1865. It has been adapted to
   numerous movies. In 2010, it has been adapted to
   Hollywood movie.
   ```

Соответствующий вывод:

```
It was released on 26 November 1865.
It has been adapted to numerous movies.
Charles Lutwidge Dodgson wrote the book.
He wrote it using the pen name Lewis Carroll.
John Tenniel drew the 42 pictures in the book.
In 2010, it has been adapted to Hollywood movie.
The book was published by Macmillan and Co in London.
The title is usually shortened to Alice in Wonderland.
Alice's Adventures in Wonderland is a children's novel.
```

Требуемый интерфейс для классового типа «массив нуль-терминированных строк»:

- Задание начального состояния: пустой массив.
- Добавление нуль-терминированной строки, выделенной в динамической памяти, в конец массива, массив принимает владение данной строкой.
- Сортировка строк массива по критерию предшествования, заданному указателем на функцию, принимающую указатели на две нуль-терминированных строки (критерий возвращает истину, если первая строка должна идти перед второй).
- Вывод массива.

3. Вводится информация об интервалах на числовой оси в виде последовательности строк из пар вещественных чисел, разделённых пробелами, задающих соответствующий отрезок, включая оба конца. Записать в выходной файл набор интервалов в том же формате по одному на строку, являющихся объединением интервалов входного файла. Выходные интервалы не должны иметь пересечений и должны располагаться по порядку.

   Пример ввода:

   ```
   1.2 3.4
   -1.8 0.1
   0.5 1.5
   1.3 2.8
   -2 -1
   ```

   Соответствующий вывод:

   ```
   -2 0.1
   0.5 3.4
   ```

   Требуемый интерфейс для классового типа «сортированный массив интервалов без пересечений»:

   - Задание начального состояния: пустой массив.
   - Добавление нового интервала в массив и его объединение с имеющимися, если это необходимо. После каждой из этих операций массив должен удовлетворять критериям из условий задачи.
   - Вывод массива.

4. Постройте двумерный растровый фрактал порядка N по данному шаблону размера $h \times w$. Введите значения параметров N, h и w, а затем h строк по w символов, содержащих изображение шаблона, состоящее из символов **#** и пробел (если строка короче w, считать оставшиеся символы пробелами). Фракталом порядка 1 является сам шаблон размера $h \times w$. Фракталом порядка N, $N > 1$ является двумерное изображение размера $h_{N-1} \cdot h \times w_{N-1} \cdot w = h^N \times w^N$, где h_{N-1} — высота, а w_{N-1} — ширина фрактала порядка $N-1$. Каждый элемент фрактала порядка N размером $w_{N-1} \times h_{N-1} N$ состоит из пробелов, если соответствующий элемент шаблона — пробел, и из содержимого фрактала порядка $N-1$, если в этом месте шаблона символ **#**.

   Пример ввода:

   ```
   3 2 3
   ##
   # #
   ```

   Соответствующий вывод:

```
## ##    ## ##
# ## #   # ## #
##    ## ##    ##
# #   # ## #   # #
## ##            ## ##
# ## #           # ## #
##    ##          ##    ##
# #   # #         # #   # #
```

Требуемый интерфейс для классового типа «окно в двумерный массив булевских значений с шаговыми доступом»:

- Создание окна в новую область памяти указанного размера. Окно, созданное этой функцией, владеет памятью, которую выделила.
- Создание подокна в другое окно по координатам левого верхнего угла, ширине и высоте. Окно, созданное этой функцией, не владеет данными, к которым даёт доступ.
- Чтение ширины окна.
- Чтение высоты окна.
- Чтение значения по заданным координатам.
- Запись значения по заданным координатам.
- Вывод окна в виде таблицы из символов решётки и пробела.

5. Имеется поезд из N вагонов, пронумерованных по порядку, начиная с 1. К поезду можно пристыковывать и отстыковывать вагоны спереди и сзади. Новые вагоны, добавляемые к поезду, получают следующие по порядку номера, после последнего использованного номера. В начале работы программы вводится начальное число вагонов, а затем — операции по стыковке и расстыковке по одной на строку. Каждая операция состоит из символа операции `'+'` — пристыковка или `'-'` — отстыковка, символа направления `'>'` — спереди или `'<'` — сзади и числа вагонов, участвующих в операции. Вывести состояние состава в начале и после каждой операции над ним в виде строки номеров вагонов по порядку от начала через пробел. Операции отстыковки, пытающиеся отстыковать вагонов больше, чем имеется, допустимы, в результате остаётся пустое место.

   Пример ввода:

```
3
+>2
-<1
+>3
-<2
+<2
+>1
+>2
-<4
```

   Соответствующий вывод:

```
1 2 3
5 4 1 2 3
5 4 1 2
8 7 6 5 4 1 2
8 7 6 5 4
8 7 6 5 4 9 10
11 8 7 6 5 4 9 10
13 12 11 8 7 6 5 4 9 10
13 12 11 8 7 6
```

   Требуемый интерфейс для классового типа «дек целых чисел»:

   - Задание начального состояния: пустой дек.
   - Возвращение числа элементов в деке.
   - Добавление элемента в начало.
   - Добавление элемента в конец.
   - Удаление заданного числа элементов с начала (считать удаление несуществующих элементов неопределённым поведением).

- Удаление заданного числа элементов с конца (считать удаление несуществующих элементов неопределённым поведением).
- Вывод содержимого дека.

6. В начале ночи на дереве живёт колония из n «дискретных светлячков». Все параметры жизни дискретного светлячка выбираются как целые числа в интервале от 1 до заданного максимального значения. Время жизни светлячка имеет максимум t_1, его яркость — l. Новые светлячки рождаются по одному через промежутки времени с максимальным значением t_2. Светлячкам при рождении даются номера по порядку, начиная с 1. Требуется вычислить минимальную и максимальную яркость всего дерева со светлячками в течение времени T, яркость дерева считать суммой яркостей светлячков. Параметры n, t_1, l, t_2 и T задаются пользователем. Вывести по одной строке на каждое событие рождения (включая n начальных в нулевой момент времени) и смерти светлячка с указанием времени события, порядкового номера светлячка, его яркости и общей яркости дерева после этого изменения. В конце вывести минимальную и максимальную яркость дерева за всё время.

   Пример вывода:

   ```
   0 : 0 + 12 = 12
   0 : 1 + 14 = 26
   0 : 2 + 13 = 39
   3 : 3 + 10 = 49
   5 : 2 - 13 = 36
   8 : 4 + 11 = 47
   9 : 0 - 12 = 35
   Min: 35, Max: 49
   ```

   Для хранения параметров светлячка используйте вспомогательный классовый тип, хранящий номер светлячка, его яркость и время его смерти в виде времени с начала моделирования. Требуемый интерфейс для классового типа «очередь погибающих светлячков»:

   - Задание начального состояния: пустая очередь.
   - Добавление светлячка с заданными характеристиками в очередь так, чтобы она осталась отсортированной по увеличению времени смерти.
   - Возвращение времени смерти светлячка, которому осталось жить меньше всех или 0, если очередь пуста.
   - Извлечение параметров светлячка, который погибнет следующим с удалением его из очереди (считать удаление из пустой очереди неопределённым поведением).

7. Вводится размер матрицы $m \times n$ целых чисел, а затем её значения. Отсортировать её строки по возрастанию значений столбца i, а затем отсортировать её столбцы по возрастанию значений строки j после первой сортировки. Значения i и j задаются пользователем, строки и столбцы нумеруются, начиная с 1. Вывести результирующую матрицу.

   Пример ввода таблицы 4×4:

   ```
   19 81 34 57
   33 46 18 44
   50 99 83 85
   34 55 66 12
   ```

   Требуемый вывод для значений $i = 1$, $j = 2$:

   ```
   34 19 57 81
   18 33 44 46
   66 34 12 55
   83 50 85 99
   ```

   Требуемый интерфейс для классового типа «матрица целых чисел»:

   - Задание начального состояния: матрица из нулей заданного размера.
   - Чтение значения по указанным координатам.
   - Запись значения по указанным координатам.
   - Сортировка строк матрицы по возрастанию значений указанного столбца.
   - Сортировка столбцов матрицы по возрастанию значений указанной строки.

- Вывод матрицы.

Для перестановки строк и столбцов, не затрагивая сами данные, можно помимо самой двумерной структуры ввести два дополнительных одномерных массива индексов строк и столбцов, изначально заполненных последовательными номерами. В процессе сортировок вместо самих данных сортируются соответствующим образом только эти массивы индексов. При доступе к элементу по координатам необходимо обратиться к этим массивам индексов, чтобы преобразовать логические координаты в физические, соответствующие реальному размещению данных.

8. Инопланетные агенты из воюющих миров десантируются в каньон планеты X. Каньон настолько узок, что для представления их положения в нём достаточно одной целочисленной координаты из интервала $[0; w]$. За счёт преимущества атаки с воздуха, в результате очередной высадки агента, приземляющегося в итоге в положение x, все остальные агенты из интервала $[x - d, x + d]$ уничтожаются. Всего в каньон высаживается n агентов строго по очереди, т.к. столкновения десантных кораблей в атмосфере никому не выгодны. Ввести параметры w, d и n. Вывести хронику событий: высадку агентов с указанием их номера по порядку и планируемой координаты приземления, выбираемой равновероятно с помощью генератора псевдослучайных чисел, а также номера уничтоженных ими в процессе десантирования врагов. После высадки вывести координаты и номера оставшихся в каньоне живых агентов.

   Пример вывода для $w = 100$, $d = 10$ и $n = 3$:

   ```
   Agent 1 landing at 17.
   Agent 2 landing at 25.
   Agent 1 killed by agent 2.
   Agent 3 landing at 70.
   Result:
   Agent 2 at position 25.
   Agent 3 at position 70.
   ```

   Для хранения информации об агенте введите вспомогательный классовый тип, хранящий его номер и координату. Требуемый интерфейс для классового типа «агенты в каньоне»:

   - Задание начального состояния: пустой каньон с заданным параметром d.
   - Высадка агента с данным номером по указанным координатам, соответствующие модификации и вывод сообщений.
   - Вывод текущего расположения агентов в каньоне.

9. Реализуйте буферное окно для обработки строки текста. Вводится последовательность печатных непробельных символов, набираемых на клавиатуре, некоторые из которых зарезервированы для обозначения специальных клавиш: `<` — стрелка влево, `>` — стрелка вправо, `B` — Backspace, `D` — Delete. После ввода значения каждой нажимаемой клавиши вывести новое содержимое виртуального редактора текста, отметив в нём положение курсора знаком `|`. «Нажатие» стрелки влево и Backspace в крайнем левом положении курсора и стрелки вправо и Delete в крайнем правом игнорировать и новое состояние буфера ввода не выводить.

   Пример ввода:

   ```
   abcdef<<<DD<<<<<<<<<<<<<gh>>>>>>>>>>>>>>ijk<BBB
   ```

   Соответствующий вывод:

   ```
   a|
   ab|
   abc|
   abcd|
   abcde|
   abcdef|
   abcde|f
   abcd|ef
   abc|def
   abc|ef
   abc|f
   ab|cf
   ```

```
a|bcf
|abcf
g|abcf
gh|abcf
gha|bcf
ghab|cf
ghabc|f
ghabcf|
ghabcfi|
ghabcfij|
ghabcfijk|
ghabcfij|k
ghabcfi|k
ghabcf|k
ghabc|k
```

Требуемый интерфейс для классового типа «буферное окно символов»:

- Задание начального состояния: пустое окно.
- Возвращение числа хранимых символов.
- Чтение символа из заданной позиции.
- Вставка символа в заданную позицию.
- Удаление символа из заданной позиции.

Корректные позиции вставки и удаления считать как для обычного массива.

Глава 9

Основы объектно-ориентированного программирования на языке C++

В этой главе мы рассмотрим основные возможности объектно-ориентированной парадигмы языка C++. Объединение данных с алгоритмами их обработки является основным способом превращения структур данных в полноценные самостоятельные сущности, являющиеся элементами построения объектно-ориентированной программы.

Вначале рассмотрим дополнительные полезные возможности языка, которые нам понадобятся.

9.1. Аргументы по умолчанию

Встречаются ситуации, в которых та или иная функция чаще всего вызывается с одними и теми же значениями нескольких параметров. В таких случаях C++ позволяет задать ***аргументы по умолчанию (default arguments)***.

Аргументы по умолчанию позволяют задать значения нескольких последних параметров функции, которые используются, если соответствующие аргументы не указаны при вызове. Требование отсутствия параметров без имеющихся аргументов по умолчанию после параметров, имеющих их, позволяет однозначно соотнести аргументы с параметрами. Значения аргументов по умолчанию задаются в одном из описаний функции, обычно первом, и имеют вид инициализации копированием:

```
1  // Прочертить линию по вектору (dx,dy) толщиной width,
2  // пунктирную, если dashed == true, сплошную иначе.
3  // Большинство линий имеют ширину 1 и являются сплошными,
4  // поэтому для удобства используются значения аргументов по умолчанию.
5  void draw(double dx,double dy,double width = 1.,bool dashed = false);
6
7  void f()
8  {
9      // Эквивалентно draw(+12.3,-45.6,1.,false),
10     // использованы оба значения по умолчанию.
11     draw(+12.3,-45.6);
12
13     // Эквивалентно draw(+12.3,-45.6,1.5,false),
14     // использовано второе значение по умолчанию.
15     draw(+12.3,-45.6,1.5);
16
17     // Эквивалентно draw(+12.3,-45.6,1.5,true),
18     // оба значения по умолчанию не использованы.
19     draw(+12.3,-45.6,1.5,true);
20
21     // Эквивалентно draw(+12.3,-45.6,1.,true),
```

```
22      // оба значения по умолчанию не использованы.
23      // Хотя значение третьего аргумента по умолчанию
24      // нам подходит, чтобы указать нестандартный
25      // четвёртый требуется указать и третий - опускать
26      // можно только в конце.
27      draw(+12.3,-45.6,1.,true);
28  }
29
30  // Определение draw, повторно давать значения аргументов по умолчанию
31  // ненужно и нельзя.
32  void draw(double dx,double dy,double width,bool dashed)
33  {
34      // ...
35  }
```

Если в значениях аргументов по умолчанию имеются имена, они ищутся в контексте описания, их содержащего, но эти выражения вычисляются при каждом вызове функции заново.

```
1   int x = 10;
2
3   // Имя x связывается с ::x
4   void f(int arg = x);
5
6   void g()
7   {
8       // Изменяет ::x после описания
9       // аргумента по умолчанию f.
10      ++x;
11      {
12          // Ещё один x.
13          int x = 20;
14
15          // В вызове f используется
16          // привязанный в описании
17          // объект ::x, со значением
18          // в момент вызова -
19          // эквивалентно f(11);
20          f();
21      }
22  }
```

Использование локальных переменных в этих выражениях запрещено.

```
1   // ОШИБКА: кажется удобным, но запрещено -
2   //         другой параметр есть локальная
3   //         переменная.
4   void f(int x,int y = x);
```

Формальные требования к аргументам по умолчанию таковы: в каждом описании они могут быть только для тех параметров, для которых ещё не заданы, и, если значение по умолчанию указано для конкретного аргумента, все последующие должны тоже иметь значения по умолчанию, указанные в этом или прошлых определениях.

```
1   // Описание со значением аргумента по
2   // умолчанию для последнего параметра.
3   void g(int x,int y = 5);
4
5   // Совместимое описание, задающее
6   // значение аргумента по умолчанию
7   // для первого параметра. Значение
8   // для второго указано ранее и здесь
9   // дублироваться не может.
10  void g(int x = 3,y);
```

Аргументы по умолчанию указываются только для аргументов в описании функций:

```
// ОШИБКА: аргумент по умолчанию для сущности,
//         не являющейся функцией.
void (*p)(int x = 5);
```

Аргументы по умолчанию могут задавать значения безымянных параметров, синтаксис тот же, но выглядит на первый взгляд странно:

```
// Второй параметр без имени типа double
// имеет значение по умолчанию 0.
void f(int a,double = 0.);
```

Основным следствием этих правил является то, что при использовании аргументов по умолчанию в описаниях функций в заголовочных файлах именно там все они и указываются, а в соответствующих определениях в единице трансляции — нет. Это частый источник ошибок, когда описание копируется из заголовочного файла в единицу трансляции для написания текста определения соответствующей функции.

9.2. Последовательность стандартных преобразований

9.2.1. Приведения типов

Во многих случаях выражение должно претерпеть несколько неявных преобразований, прежде чем будет получено значение с требуемыми в контексте его использования характеристиками. Все допустимые последовательности таких преобразований, состоящие из заданных языком преобразований, включая пустую, определены как ***стандартная последовательность преобразований (standard conversion sequence)***, которая включает в себя:

1. Не более одного из следующих преобразований:
   - Преобразование леводопустимого выражения;
   - Преобразование массива в указатель;
   - Преобразование функции в указатель;
2. Не более одного из следующих преобразований:
   - Целочисленные повышения;
   - Повышение с плавающей точкой;
   - Целочисленные преобразования;
   - Преобразования с плавающей точкой;
   - Преобразования между типами с плавающей точкой и целыми;
   - Преобразования указателей;
   - Логические преобразования;
3. Наконец, при необходимости производится преобразование квалификаторов.

Позднее мы познакомимся также с ***пользовательскими (user-defined)*** преобразованиями. Из известных нам конструкций, инициализация агрегата списком относится к пользовательским преобразованиям. В общем случае, максимально длинная цепочка неявных преобразований состоит из стандартной последовательности преобразований, пользовательского преобразования, и ещё одной стандартной последовательности.

```
int a[10];

// Преобразование массива в указатель (int [10], lvalue -> int*, prvalue),
// Логические преобразования (int*, prvalue -> bool, prvalue)
bool x = a;

double x = 3;

// Преобразование леводопустимого выражения (double, lvalue -> double, prvalue),
// Преобразования между типами с плавающий точкой и целыми
// (double, prvalue -> int, prvalue).
int y = x;
```

9.3. Перегрузка функций

В C++ допустимо несколько несовместимых описаний одного имени-функции в одной области видимости — такие функции называют ***перегруженными (overloaded)***. Это позволяет дать одно и то же имя одним и тем же по смыслу результата вариантам алгоритмов или их частей, использующих разные входные данные. При нахождении нескольких описаний функции в результате поиска имён — ***кандидатов (candidates)*** или перегрузок, транслятор осуществляет сопоставление аргументов в операции вызова функции с типами параметров этих описаний. Вначале отсеиваются описания, которые невозможно использовать в таком вызове. Если после этого осталось более одной функции, из оставшихся ***годных (viable)*** производится попытка выбрать одну, которая ***лучше (best)*** чем все остальные. В результате всего процесса ***разрешения перегрузок (overload resolution)*** должно остаться в точности одно описание, иначе программа не согласована.

Поскольку выбор из кандидатов осуществляется только по типам аргументов, запрещены перегрузки, отличающиеся только типом возвращаемого значения — из таких будет невозможно выбрать одну лучшую. При записи типов несколькими совместимыми способами (через псевдонимы типов, разложение массивов и функций и различий квалификаторов типов на верхнем уровне), если других отличий нет, дополнительные описания перегрузками не являются.

Годными считаются описания, которые подходят по числу параметров (с учётом значений аргументов по умолчанию и вариадических параметров), и возможны неявные приведения типов при инициализации копированием значениями аргументов параметров:

```
1  // Три перегрузки функции f.
2  void f();            // 1
3  void f(int x);       // 2
4  void f(int x,int y); // 3
5
6  void g()
7  {
8      // После отсечения по числу параметров
9      // остаётся ровно одно годное описание:
10     f();          // 1
11     f(42);        // 2
12     f(0.34,true); // 3
13 }
14
15 // Это формально корректный набор перегрузок.
16 // Вторая функция вызывается для двух аргументов.
17 // При вызове с одним окажется, что типы
18 // первого параметра в перегрузках идентичны
19 // и выбрать лучшую нельзя, так что первую
20 // перегрузку, как и весь набор перегрузок
21 // функции h, с одним аргументом вызвать нельзя.
22 void h(double);
23 void h(double,int = 0);
24
25 void f2(double x);   // 1
26 void f2(double* xp); // 2
27
28 void g2()
29 {
30     // В каждом случае только одна перегрузка
31     // годная из-за допустимости преобразований
32     // аргументов к типам параметров.
33     f2(3.45);    // 1
34     f2(nullptr); // 2
35 }
36
37 // Это не перегрузка, а два совместимых описания -
38 // все типы параметров совместимы, имена роли не играют.
39 void f4(int a[5],int b);
40 void f4(int* x,const int y);
41
42 int f5();
```

```
43 // ОШИБКА: перегрузка, отличающаяся только типом
44 //         возвращаемого значения.
45 bool f5();
```

Несколько перегруженных описаний-кандидатов могут быть найденными из разных пространств имён сразу за счёт описаний и директив `using`:

```
1  namespace A
2  {
3      namespace B
4      {
5          void f();
6      }
7
8      namespace C
9      {
10         void f(int);
11     }
12
13     namespace D
14     {
15         void f(int,int);
16     }
17
18     using B::f;
19
20     namespace E
21     {
22         using namespace C;
23         using namespace D;
24
25         void g()
26         {
27             // Поиск имён доходит до ::A,
28             // в котором видны все три имени
29             // ::A::B::f, ::A::C::f и ::A::D::f,
30             // образующие набор кандидатов.
31             f();
32             f(1);
33             f(1,2);
34         }
35     }
36 }
```

Для определения вызываемой функции из годных производится сравнение типов аргументов и параметров. В отношении каждого отдельно взятого параметра одна перегрузка может быть «лучше» другой или несравнима с ней. Если одна перегрузка для всех параметров не хуже другой и как минимум по одному лучше, то она лучше этой перегрузки в целом. Если из двух перегрузок ни одна не лучше другой, их называют ***неразличимыми (indistinguishable)***. Если одна перегрузка лучше всех остальных в целом, она и является результатом разрешения перегрузок.

Для каждого отдельно взятого параметра производится сравнение последовательностей неявных преобразований. Любые стандартные преобразования к известному типу параметра считаются лучше пользовательских преобразований. Для сравнения двух стандартных последовательностей преобразования каждому её шагу присваивается один из ***рангов (rank)*** (см. таблицу 9.1).

Ранг «точное соответствие» лучше ранга «повышение», который в свою очередь лучше «преобразования». Ранг всей последовательности преобразований определяется как худший из рангов входящих в неё преобразований («точное соответствие» для пустой последовательности). Помимо этого, если две последовательности имеют один ранг, применяются следующие правила:

- Если одна последовательность является частью другой, то более короткая лучше.
- Если различие только в преобразовании квалификаторов, лучше преобразование, добавляющее меньшее их число.

Преобразование	**Категория**	**Ранг**
(отсутствующее)	эквивалентность	точное соответствие
Преобр. леводоп. выражения Преобр. массива в указатель Преобр. функции в указатель	преобразование леводопустимого выражения	
Преобр. квалификаторов	квалификаторы	
Целочисленные повышения Повышения с плав. точкой	повышение	повышение
(все остальные)	преобразование	преобразование

Таблица 9.1: Ранги элементов стандартных последовательностей преобразования

- Цепочка преобразований, включающая преобразование из указателя или `std::nullptr_t` в `bool` хуже, чем та, которая без неё обходится.

В остальных случаях преобразования одного ранга неразличимы.

Общий смысл этих правил — лучше то, что наиболее «похоже».

```
1  void f1(int);    // 1
2  void f1(double); // 2
3
4  void g()
5  {
6      // Годны обе, но
7      // double->int (преобразование) хуже чем
8      // double->double (точное совпадение)
9      f1(3.5); // 2
10
11     // short->int (повышение) лучше чем
12     // short->double (преобразование)
13     f1(static_cast<short>(3)); // 1
14
15     // ОШИБКА: оба варианта неразличимы:
16     //         long->int (преобразование)
17     //         long->double (преобразование)
18     f1(1L);
19 }
20
21 void f2(const char* s); // 1
22 void f2(char *s);       // 2
23
24 void g2()
25 {
26     // Обе перегрузки являются точным совпадением:
27     // char [5] -> const char * -
28     //   преобразование массива в указатель и
29     //   преобразование квалификаторов,
30     // const char [5] -> char* -
31     //   преобразование массива в указатель,
32     // но вторая - часть первой.
33     char s[] = "test"; // 2
34     f2(s);
35
36     // Годна только первая перегрузка
37     // (вторая отбрасывает квалификаторы).
38     f2("other test"); // 1
39 }
40
41 void f3(const volatile char* s); // 1
42 void f3(volatile char *s);       // 2
43
44 void g3()
45 {
46     // Обе перегрузки являются точным совпадением
47     // и содержат одинаковые преобразования:
```

```
48      // char [5] -> const volatile char * и
49      // const char [5] -> char* - это
50      //   преобразование массива в указатель и
51      //   преобразование квалификаторов,
52      // но во втором случае надо добавить только
53      // один квалификатор, а в первом - два.
54      char s[] = "test"; // 2
55      f3(s);
56  }
57
58  void f4(bool);
59  void f4(char*);
60
61  void g4()
62  {
63      // Обе перегрузки включают преобразованием:
64      // std::nullptr_t -> bool -
65      //   логическое преобразование,
66      // std::nullptr_t -> char* -
67      //   преобразование указателей,
68      // но первое содержит преобразование
69      // std::nullptr_t в bool, а второе - нет.
70      f4(nullptr); // 2
71  }
72
73  void f6(int,const int*);
74  void f6(double,int*);
75
76  void g6()
77  {
78      // ОШИБКА: ни одна из перегрузок не
79      //         лучше другой:
80      // По первому параметру лучше первая
81      // (повышение против преобразования),
82      // по второму лучше вторая
83      // (точные соответствия, второе лучше, т.к. меньше
84      //  добавлений квалификаторов).
85      int x = 5;
86      f6('0',&x);
87  }
88
89  void f7(double,unsigned);
90  void f7(unsigned long long,unsigned long long);
91
92  void g7()
93  {
94      // По первому параметру перегрузки не различимы
95      // (конверсия против конверсии), по второму
96      // первая лучше, и, значит лучше в целом.
97      f7(0,5u); // 1
98  }
```

Особо отметим пример 2 выше, где показано, что перегрузка по наличию квалификатора `const` позволяет вызывать две разных перегрузки в зависимости от квалифицированности аргумента.

Если в качестве аргумента в вызове функции стоит список инициализации, он не является выражением и обрабатывается особым образом:

- Он может соответствовать параметру агрегатного типа, если может быть использован в качестве инициализатора для него, это пользовательское преобразование.
- Список из одного элемента соответствует параметру, если указанное значение преобразуемо к нему, и имеет связанный с этим преобразованием ранг.
- Пустой список может соответствовать инициализации параметра по значению, ранг при этом «точное соответствие».

```
1   struct S
```

```
{
    int a;
    double b;
};

void f(S);

void g(int);

void h()
{
    // Инициализация агрегата, пользовательское преобразование.
    f({12,3.4});

    // То же, что и g(5).
    g({5});

    // То же, что и g(0).
    g({});
}
```

Поскольку имя перегруженной функции не может быть сопоставлено с конкретной перегрузкой без наличия информации о типе параметров, взятие адреса перегруженной функции (явное операцией или неявное разложением) возможно только в некоторых контекстах: инициализаторах (включая аргументы функций и возвращаемые значения), присваиваниях и явных приведениях типов. В этих случаях известен требуемый тип результата, из которого можно взять типы аргументов для участия в разрешении перегрузок:

```
void f(int);
void f(double);

// Прямая инициализация указателя на функцию pf.
// Для нахождения требуемой перегрузки типы аргументов
// берутся из типа pf.
void (*pf)(int){f};

void g()
{
    // ОШИБКА: взятие адреса перегруженной функции
    //         в контексте, в котором нет информации
    //         для выбора перегрузки.
    &f;
}
```

9.4. Временные объекты

Рассмотрим ещё одно время хранения — временное. Объекты с таким временем хранения создаются непосредственно в выражениях, в том числе без использования какого-либо явного синтаксиса, и существуют до конца полного выражения, в котором были созданы. Временные объекты бывают только классовых типов или типа категории массив (очень редко). Категория значения выражений, соответствующих временным объектам — `prvalue`.

Простейшим примером временного объекта является значение классового типа, возвращённое из функции:

```
struct S { int a; };

S f()
{
    return {};
}

int main()
{
    // Результат вызова функции --- временный объект, который
    // после использования в выражении уничтожается.
```

```
12      int x = f().a;
13
14      // ОШИБКА: взятие адреса от не леводопустимого выражения.
15      &f();
16  }
```

Временные объекты будут создаваться в рассматриваемых нами далее в языке конструкций неявно, что требует обратить внимание на ситуации, когда это происходит, поскольку это может повлиять на производительность и корректность программ.

В некоторых случаях временные объекты необходимо создать явно, для чего используется синтаксис приведения типов в ***функциональном стиле (functional notation)***. Он состоит из имени типа создаваемого временного объекта, состоящего только из имён и ключевых слов, за которым следует один из видов инициализаторов, выполняющих прямую инициализацию. Для более сложных имён типов придётся предварительно определить псевдоним типа. Такая конструкция для типов, не являющихся классами или массивами создают просто чисто праводопустимые значения.

```
1   void f()
2   {
3       struct S
4       {
5           int a,b;
6       };
7
8       // Создание временного объекта, его прямая инициализация списком,
9       // чтение значения его подобъекта и использования его в инициализаторе
10      // объекта x, начальное значение 4.
11      int x = S{1,2}.a+3;
12
13      // Два чисто праводопустимых значения типа int,
14      // равных результату прямой инициализации по значению (0)
15      // складываются, результат отбрасывается.
16      int()+int();
17  }
```

Временные объекты пригодятся нам в обсуждении следующей важной конструкции создания производных типов: ссылок.

9.5. Ссылки

Ссылки (reference) — категория создания производных типов на базе других, позволяющая именовать другие объекты или функции (сами ссылки являются отдельным от них видом сущностей программы). В первом приближении они похожи на неизменяемые указатели, но отличаются от них синтаксисом использования и другими полезными свойствами.

В современном языке C++ имеется два типа ссылок: ***леводопустимые ссылки (lvalue reference)*** и ***праводопустимые ссылки (rvalue reference)***. Ссылки при их определении должны быть обязательно инициализированы, этот процесс называется их ***привязкой (binding)*** к сущности, которую они будут именовать, пока видны. Другое название процесса инициализации для ссылок подчёркивает его принципиальное отличие от последующих присваиваний. В выражениях ссылки, если не указано обратное, трактуются как сущности, к которым они привязаны. Они совпадают с ним по типу и значению, а категория значения — lvalue, кроме случая безымянных праводопустимых ссылок, которые являются первыми для нас и почти единственными представителя категории значений xvalue.

Ссылки используют синтаксис конструкции создания производного типа, аналогичный указателям, используя вместо пунктуатора * & для леводопустимой ссылки и && для правой:

```
1   // Функция f имеет два параметра:
2   // - леводопустимую ссылку на неизменяемый int и
3   // - праводопустимую ссылку на int.
4   void f(const int& a,int&& b);
```

Ссылки, как и указатели, могут быть произведены от любых типов, включая неполные. Ссылки используются для именования конкретных сущностей, поэтому ссылки на `void`

запрещены. Массивы и указатели на ссылки недопустимы, но ссылки могут выступать в качестве базового типа для функций, являясь их возвращаемым значением, а также в качестве их параметров и членов классов.

```
// Пока вопрос инициализации ссылок не рассмотрен,
// будем приводить примеры в виде членов структур -
// их инициализация производится позднее, в определении
// соответствующих объектов.
struct S
{
    // Ссылка на int - член структуры S
    int& a;

    // Леводопустимая ссылка на указатель на int.
    int*& b;

    // ОШИБКА: указатель на ссылку
    int&* c;

    // Леводопустимая ссылка на массив из 10 int'ов.
    int (&d)[10];

    // ОШИБКА: массив ссылок
    int &e[10];

    // Ссылка на функцию без параметров и возвращаемого значения.
    void (&f)();
};

// Функция, принимающая и возвращающая
// праводопустимую ссылку на int.
int&& f(int &&);

// ОШИБКА: ссылка на void.
void& v;

// ОЩИБКА: ссылка без инициализатора.
int& x;
```

Явное произведение типа ссылки от другой ссылки запрещено. В случае неявного создания такой конструкции с использованием псевдонимов типов создание ссылки на ссылку «коллапсирует» в одну ссылку. Она является леводопустимой, если хотя бы одна из двух изначальных была леводопустимой.

```
using rrint = int &&;

struct S
{
    // ОШИБКА: явная ссылка на ссылку
    int & & a;

    // Неявная праводопустимая ссылка
    // на праводопустимую ссылку, то же
    // самое что просто int&& или rrint.
    rrint && b;

    // Неявная леводопустимая ссылка
    // на праводопустимую ссылку, то же
    // самое что просто int&.
    rrint & c;
};
```

Поскольку ссылки не являются объектами, квалификаторы типов к ним не применимы. Их указание допускается, но игнорируется.

```
struct S
{
```

```
3       // Ссылка на неизменяемый int.
4       int const &x;
5
6       // const применён к ссылке и игнорируется:
7       // то же, что и int&.
8       int & const y;
9   };
```

9.5.1. Ссылки на объекты

Рассмотрим использование ссылок на типы объектов (не функций).

Чаще всего применяются леводопустимые ссылки. Они могут быть привязаны к леводопустимым выражениям того же типа — такая привязка называется ***прямой (direct)***. Такая привязка возможна также к ссылке, имеющей дополнительные квалификаторы, не имеющиеся у значения (отбрасывать квалификаторы, как обычно, нельзя). После привязки такая ссылка при использовании в выражениях выступает в роли леводопустимого выражения, идентифицирующего привязанную сущность. Это ещё один пример того, что в программе один объект может идентифицироваться разными выражениями — в данном случае, несколькими разными именами, через имя заданное в его описании и через имена ссылок на него:

```
1   int x = 15;
2
3   // Прямая привязка к леводопустимому значению.
4   int& rx = x;
5
6   // Изменяет объект, именуемый x, используя ссылку.
7   // Саму привязку ссылки изменить нельзя после её
8   // создания, операция присваивания работает с объектом,
9   // идентифицируемым ссылкой.
10  rx = 42;
11
12  // Истинно всегда, т.к. сравнивает адреса одного и того же объекта
13  // под разными именами.
14  assert(&x==&rx);
15
16  void f()
17  {
18      // Прямая привязка к леводопустимому значению,
19      // ссылка на более строго квалифицированный объект,
20      // чем привязываемое значение.
21      const int& rcx = x;
22
23      // Преобразование леводопустимого выражения с
24      // чтением значения объекта, именуемого x, rx и rcx.
25      std::cout << "x = " << rcx << '\n';
26  }
27
28  volatile int y;
29
30  // ОШИБКА: привязка не может отбросить квалификатор.
31  int& ry = y;
```

В приведённых примерах ссылка использовалась только в качестве альтернативного имени объекта, такое использование не требует хранения каких-либо данных в памяти среды выполнения, поэтому представление ссылок и само наличие у них такового определяется реализацией в каждом отдельном случае.

Если базовый тип леводопустимой ссылки имеет квалификаторы в точности `const`, она может быть привязана к праводопустимым значениям. Если они имеют категорию типа «класс» или «массив» и совпадают по типу с базовым типом ссылки, привязка также осуществляется напрямую, при этом время жизни временного объекта, соответствующего данному значению, расширяется на всё «время жизни» ссылки, которое определяется как для обычных объектов:

```
1   struct point
2   {
3       double x,y;
```

```
};

void f()
{
    // Привязка леводопустимой ссылки на неизменяемый
    // объект классового типа point к временному объекту.
    const point& p = point{1.2,3.4};
} // Здесь ссылка перестаёт быть видимой и связанный с ней
  // временный объект уничтожается.

void g()
{
    // Праводопустимое значение категории типа <<массив>> -
    // - редкий случай, чтобы создать его придётся использовать
    // псевдоним типа, т.к. в синтаксисе создания временного
    // значения не может быть конструкций создания производных типов:
    using array = int[5];

    // Создание временного значения типа <<массив из 5 int'ов>>
    // и привязка его к ссылке на неизменяемый объект того же
    // типа (записанного явно) также расширяет время жизни
    // временного объекта.
    const int (&a)[5] = array{1,2,3,4,5};
}
```

Наконец, остаются случаи, когда тип инициализатора ссылки не леводопустим и не является классом или массивом, или не совпадает с базовым типом ссылки. В таком случае осуществляется попытка создать временный объект и инициализировать его копированием из данного инициализатора — если это допустимо, ссылка привязывается к этому временному значению и расширяет его время жизни, как и в предыдущем случае — такая привязка прямой не является, т.к. привязывается не в точности к указанному объекту, а к дополнительному, созданному в рамках привязки объекту. Если и этот третий вариант привязки не применим, то программа не согласована.

```
// Непрямая привязка - int
// не класс и не массив,
// создаётся временный объект
// для хранения значения 42,
// который существует вместе с ссылкой.
const int& x = 42;

int y = 15;

// y - леводопустимое выражение,
// но его тип не требуемый double,
// создаётся временный объект типа double,
// инициализируемый копированием
// инициализатором y, и к нему привязывается
// ссылка.
const double& z = y;

double* p = nullptr;

// ОШИБКА: инициализация копированием
//         временного объекта типа
//         int * const из инициализатора
//         типа double * невозможна.
int * const & pr = p;
```

Первым практически полезным примером использования ссылок является замена ими указателей в параметрах функций — это избавляет от необходимости каждый раз разыменовывать их для доступа к требуемому значению. Такая замена допустима только для функций, которым не требуется приём нулевых указателей для обозначения опциональных параметров — создание «нулевой ссылки» в корректной программе невозможно, т.к. потребует разыменования нулевого указателя. Для функций, требующих передачи ненулевого адреса, использование ссылки как раз является индикацией этого факта, видимого в её описании,

что особенно полезно. Ссылками также не следует заменять использование указателей в параметрах функции, когда они по смыслу могут указывать более, чем на один объект — хотя такое использование корректно на уровне языка, оно не принято. Ссылки можно применить как для указателей на выходные параметры, так и на входные, которые передавались по указателю из-за своего большого размера.

```
1  struct S
2  {
3      int a;
4      double b;
5      char c[64];
6  };
7
8  // Функция ищет в последовательности из n объектов типа S, начиная с
9  // лежащего по адресу s, эквивалентный объекту, адрес которого указан
10 // key. Адрес первого найденного элемента возвращается, при этом,
11 // если index - не нулевой указатель, по его адресу записывается
12 // номер найденного элемента. Если такой элемент не найден,
13 // возвращается нулевой указатель.
14 const S* search(const S s[],std::size_t n,const S* key,std::size_t* index)
15 {
16     for(size_t i=0;i<n;++i)
17         if(s[i].a==key->a){
18             if(index)
19                 *index = i;
20             return s+i;
21         }
22     return nullptr;
23 }
24
25 // Вариант со ссылками:
26 // - параметр s указывает на несколько элементов и остаётся указателем;
27 // - параметр key был указателем потому, что тип S слишком велик для
28 //   передачи по значению и является обязательным - заменён ссылкой
29 //   на неизменяемый объект.
30 // - параметр index может быть нулевым указателем и потому ссылкой
31 //   не заменён.
32 // - Возвращаемое значение может быть нулевым указателем, и потому
33 //   ссылкой не заменено.
34 const S* search_with_refs(const S s[],std::size_t n,const S& key,
35                           std::size_t* index)
36 {
37     for(size_t i=0;i<n;++i)
38         if(s[i].a==key.a){
39             if(index)
40                 *index = i;
41             return s+i;
42         }
43     return nullptr;
44 }
45
46 // Функция обменивает значения
47 // двух объектов, принимаемых по ссылке -
48 // оба параметра являются и входными, и выходными.
49 void swap(int& a,int& b)
50 {
51     int t = a;
52     a = b;
53     b = t;
54 }
55
56 void f()
57 {
58     int x = 2,y = 3;
59     // Ссылки часто избавляют от операций взятия адреса -
60     // они бы здесь понадобились, если бы swap
```

```
61      // принимал указатели.
62      swap(x,y);
63  }
```

В языке C возвращаемое значение функции не могло быть леводопустимым значением. В C++ этого можно добиться, вернув из функции леводопустимую ссылку:

```
1   int& f(size_t i)
2   {
3       static int x[10];
4       // Вернуть ссылку на элемент массива
5       // со статическим временем хранения,
6       return x[i];
7   }
8
9   void g()
10  {
11      // Записать 5 в объект,
12      // ссылку на который вернула функция.
13      f(3) = 5;
14
15      // Проверить, что значение было записано.
16      assert(f(3)==5);
17  }
```

Таким образом, ссылки являются общим способом идентификации других значений, которые не обязаны сами иметь имя.

Так же, как и с указателями, при возврате ссылки из функции нет смысла возвращать ссылку на объект с автоматическим временем хранения, т.к. такая ссылка тут же станет висячей.

Использование ссылок в качестве параметров и возвращаемых значений функций требует наличия у них некоторого представления в памяти среды выполнения. В таких случаях их представление обычно не отличается от указателей на именуемые ими сущности.

При использовании с функциями расширение времени жизни временных объектов, связанных с ссылками, имеет некоторые особенности. Во-первых, временный объект, привязанный к параметру функции уничтожается по завершении вычисления полного выражения, в которое входит соответствующая операция вызова функции. Во-вторых, расширения времени жизни временного объекта, привязываемого к возвращаемому значению функции в операторе `return` вообще не происходит — из функции нельзя вернуть ссылку на временный объект, который был в ней создан.

Как следствие, могут возникнуть некоторые проблемы, когда функция возвращает ссылку на свои параметры, которые тоже являются ссылками, потенциально на временные объекты:

```
1   // Вернуть ссылку на свой параметр.
2   const int& pass(const int& x)
3   {
4       return x;
5   }
6
7   void f()
8   {
9       // Для привязки к параметру функции pass
10      // создаётся временный объект типа const int,
11      // инициализируемый значением 5.
12      // После возврата из функции её параметры уже
13      // не существуют, но временный объект, хранящий
14      // значение 5, ссылка на который была возвращена,
15      // существует до конца выражения-инициализатора
16      // и успешно в нём считывается, чтобы стать
17      // инициализатором объекта x.
18      int x = pass(5);
19
20      // ОШИБКА: висячая ссылка.
```

```
21      //          Как в предыдущем примере, но
22      //          у привязывается к временному объекту,
23      //          который был создан для хранения значения
24      //          параметра. Этот объект уничтожается сразу
25      //          после выполнения этого определения,
26      //          ссылка y не расширяет его время жизни,
27      //          т.к. она сама привязывается к возвращаемому
28      //          значению функции, которое леводопустимо,
29      //          т.к. само является леводопустимой ссылкой.
30      const int& y = pass(5);
31  }
```

Современный C++ имеет в своём арсенале и праводопустимые ссылки. Они были введены в язык для возможности определять при вызове функции, является ли привязанный к её параметру-ссылке объект временным или нет. Знание этого факта позволяет реализовать более эффективную перегрузку функции, избегающую копирования данных, осуществляя вместо этого эффективный перенос ресурсов из временного объекта.

Праводопустимые ссылки могут быть привязаны только к праводопустимым значениям по тем же правилам, что и леводопустимые ссылки без исключения для неизменяемых базовых типов. Поскольку цель введения праводопустимых ссылок требует модификаций объекта, ими идентифицируемым, квалификатор `const` на их базовом типе на практике не используется (хотя формально допустим).

```
1   void f()
2   {
3       // Непрямая привязка к праводопустимой ссылке.
4       int&& x = 5;
5
6       // В отличие от леводопустимых ссылок
7       // на неизменяемые типы, праводопустимые
8       // ссылки могут допускать модификацию привязанных
9       // к ним объектов, даже временных.
10      x = 42;
11  }
```

Операция приведения типов позволяет осуществлять приведение к типам категории «ссылка» по обычным правилам инициализации временного значения, возвращая результирующую безымянную ссылку. Для значений общей категории glvalue это позволяет явно изменить категорию значения с lvalue на xvalue или наоборот. В контекстах, где требуется трактовать не временный объект в качестве такового, возможно приведение к типу праводопустимой ссылки леводопустимого выражения того же типа:

```
1   struct S
2   {
3       int a,b,c;
4   };
5
6   // Функция требует аргумент типа S
7   // категории значения rvalue.
8   void f(S&& s);
9
10  void g()
11  {
12      // Вызов функции с привязкой временного объекта
13      // к параметру-праводопустимой ссылке.
14      f({1,2,3});
15
16      S s{4,5,6};
17
18      // ОШИБКА: леводопустимое выражение
19      //         не привязывается к праводопустимой
20      //         ссылке.
21      f(s);
22
23      // Явным приведением типов можно "отобрать"
24      // у выражения его леводопустимость.
```

```
25      f(static_cast<S&&>(s));
26  };
```

xvalue являются результатами показанной выше операции приведения типов к праводопустимой ссылке, возврата из функции праводопустимой ссылки или доступе к члену данных значения классового типа, которое не является lvalue. Эти значения могут быть напрямую привязаны к праводопустимым ссылкам или леводопустимым ссылкам на `const`. Поскольку такой статус имеют только безымянные значения, следует учитывать то, что после привязки к параметру функции, являющемуся праводопустимой ссылкой, истекающего или чисто праводопустимого значения, имя такого параметра в самой функции выступает в качестве леводопустимого выражения:

```
1   void f(int&& x);
2
3   void g(int&& x)
4   {
5       // К x привязан формально временный объект,
6       // но x в этой функции леводопустим, и для
7       // передачи его дальше в качестве
8       // временного приходится опять его приводить.
9       f(static_cast<int&&>(x));
10  }
11
12  void h()
13  {
14     g(5);
15  }
```

9.5.2. Ссылки на функции

Для полноты картины C++ допускает ссылки на функции. Поскольку не бывает квалификаторов функций и «временных» функций, привязка значений категории типа «функция» возможно к любым видам ссылок, независимо от категории этого значения.

```
1   void f();
2
3   // Привязка функции к левододопустимой ссылке.
4   void (&rf)() = f;
5
6   // Привязка функции (формально, леводопустимого
7   // выражения) к праводопустимой ссылке.
8   void (&&rrf) = f;
9
10  void g()
11  {
12      // Вызов функции через ссылку на неё.
13      rf();
14  }
```

Праводопустимым ссылкам на функции трудно найти применение, поскольку по свойствам они не отличаются от леводопустимых — они существуют разве только для полноты системы типов.

9.5.3. Ссылки и перегрузка функций

Чтобы описать поведение параметров-ссылок в механизме перегрузки функций, следует определить правила, насколько такие параметры соответствуют тем или иным аргументам.

Прямая привязка ссылки имеет ранг «точное соответствие», непрямая привязка имеет ранг соответствующий рангу преобразования, происходящего при инициализации соответствующего временного объекта:

```
1   void f(const int&); // 1
2   void f(double);     // 2
3
4   void g()
```

```
{
    short x = 5;
    // 1: непрямая привязка, ранг повышение
    // 2: преобразование
    f(x); // 1

    int y = 3;
    // 1: прямая привязка, точное соответствие
    // 2: преобразование
    f(y); // 1

    volatile int z = 5;
    // 1: привязка леводопустимого выражения
    //    к леводопустимой ссылке невозможна
    //    из-за отбрасывания квалификаторов.
    // 2: единственная годная перегрузка.
    f(z); // 2
}
```

Квалификаторы на базовом типе сравниваются так же, как и для указателей — лучше та перегрузка, которой требуется добавить меньшее их количество:

```
void f(const int&); // 1
void f(int&);       // 2

void g()
{
    int x = 5;
    // Годны обе, 2 лучше т.к.
    // не требует добавления квалификаторов.
    f(x); // 2

    // Годна только 1, т.к.
    // к не-const леводопустимой ссылке
    // привязываются только леводопустимые значения.
    f(8); // 1
}
```

В предыдущем примере показано, как перегрузки с разными квалификаторами позволяют различить некоторые категории значений. При перегрузке разными видами ссылок действует следующее правило: привязка праводопустимого выражения к праводопустимой ссылке лучше, чем к леводопустимой. Типичная перегрузка с целью выявления временных аргументов выглядит следующим образом:

```
struct S
{
    int a,b;
};

// К леводопустимой ссылке на неизменяемый
// тип можно привязать значение любой категории.
void f(const S&); // 1
// Вторая перегрузка привязывается только к
// временным значениям и лучше первой в таком случае.
void f(S&&);      // 2

void g()
{
    S s1{1,2};
    const S s2{3,4};

    f(s1);                   // 1
    f(s2);                   // 1
    f(S{1,2});               // 2
    f(static_cast<S&&>(s1)); // 2
}
```

Мы покажем использование такой перегрузки при рассмотрении основных новых свойств классовых типов в языке C++ далее.

9.6. Возможности отдельных классовых типов

Продолжим рассмотрение классовых типов, которые являются основой поддержки объектно-ориентированной парадигмы в языке C++ путём объединения в единое понятие типа данных устройства его представления и алгоритмов его обработки, включая его инициализацию, необходимые шаги для корректного освобождения используемых им ресурсов, а также создания его копий.

В C++ в определениях классов допускаются не только объекты, которые мы уже использовали, а все следующие виды описаний:

- Описания объектов и ссылок, называемых ***членами данных (data member)*** класса. Ссылки в представлении класса, как обычно, когда это требуется, представляются в виде указателей большинством реализаций.
- Описания и определения функций, называемых ***функциями-членами (member function)*** класса. В некоторых книгах упоминается также термин ***метод (method)*** — так называют функции-члены в некоторых других объектно-ориентированных языках, но не в C++.
- Псевдонимы типов.
- Другие описания классовых типов, называемых ***вложенными (nested)***.

Например:

```
1  struct S
2  {
3      // Член данных.
4      int x;
5
6      // Функция-член.
7      void f();
8
9      // Псевдоним типа.
10     using T = int;
11
12     // Определение вложенного класса.
13     struct S2
14     {
15         // ...
16     };
17 };
```

> Само имя класса становится видимым не только в области видимости, в которой определяется класс, но и в его собственной области видимости. Считается, что это описание даётся неявно самым первым в определении класса. Это описание называется ***внедрённым именем класса (injected class name)*** и, хотя обычно не влияет на результат поиска имён и может быть перекрыто обычным образом, играет специальную роль в некоторых конструкциях, которые будут рассмотрены далее. У безымянных классов его нет.

Не квалифицированные имена в определении класса, находящиеся вне тел функций-членов, аргументов по умолчанию и вложенных классов, ищутся до их использования в самом классе, окружающих его классах, если они есть, а затем в окружающих пространствах имён.

```
1  struct S
2  {
3      // Определение псевдонима ::S::my_int.
4      using my_int = int;
5
6      // Описание члена данных, имя псевдонима
7      // не требует квалификации.
8      /* ::S:: */ my_int x;
9  };
```

Вне определения класса для доступа к этим членам обычно требуется явное указание класса как области видимости при их использовании. Имена классов могут быть компонентами квалифицированного имени, например, псевдоним `my_int` из примера выше имеет полное квалифицированное имя `::S::my_int`. При доступе к членам данных с использованием операции прямой или косвенной выборки, их имена ищутся в пространстве имён класса, соответствующего типу левого операнда, как было показано ранее.

Описания функций и членов данных в классе могут содержать спецификатор `static`, назначение которого будет рассмотрено в соответствующем разделе — оно отличается от такового при использовании его не на членах класса. По его наличию или отсутствию в описании члены класса называют ***статическими (static)*** или ***не статическими (non-static)*** соответственно. Поведение не статических членов данных нам уже известно. Рассмотрим сначала не статические функции-члены классов, а затем отличие статических членов класса от не статических.

9.6.1. Функции-члены классовых типов

Функции-члены классовых типов являются основой всех нововведений в языке C++ по сравнению с использованием классовых типов в C. Это функции, которые описаны в определении класса, в отличие от остальных функций, называемых ***свободными (free)***.

Начнём с рассмотрения формальных свойств таких функций. Функции-члены могут быть описаны или определены в определении класса. Если они определены в нём, они считаются неявно встроенными. В дальнейшем для сокращения объёма примеров автор часто будет использовать определения функций в определении класса для краткости, но на практике следует использовать их так же, как и другие встраиваемые функции — определять в теле класса следует только простые функции, подходящие для встраивания. Если они лишь описаны в нём, их определение должно находится в любом окружающем пространстве имён с полностью квалифицированным именем. Так же как и в определениях, находящихся вне пространства имён, где они описаны, имена, используемые в определении членов класса, расположенных вне определения этого класса, ищутся в области видимости соответствующего класса только в части описания после упоминания квалифицированного имени, идентифицирующего данный класс.

```
1  struct S
2  {
3      // Определение функции-члена в классе,
4      // оно неявно встроенное.
5      /* inline */ void f()
6      {
7          // ...
8      }
9
10     // Описание функции-члена в классе.
11     void g();
12
13     using my_int = int;
14
15     // Параметр и возвращаемое значение
16     // имеют тип, видимый только в классе.
17     my_int h(my_int x);
18 };
19
20 // Определение функции-члена, описанной
21 // в классе, имя квалифицировано.
22 void S::g()
23 {
24     // ...
25 }
26
27 // Определение S::h.
28 // Тип параметра указывается до имени определяемой сущности
29 // и требует квалификации, чтобы найти его.
30 // Типы параметров расположены после этого имени и ищутся
31 // в первую очередь в области видимости класса без явной
32 // квалификации.
```

```
33  S::my_int S::h(/* ::S:: */ my_int x)
34  {
35      return x+42;
36  }
```

Связанность функций-членов класса внешняя (за исключением членов локальных классов). Спецификатор `extern` к функциям-членам не применим, а спецификатор `static`, как было сказано выше, имеет другое, не относящееся к связанности значение.

Доступ к не статическим функциям осуществляется так же как и к членам данных — через операцию выборки. Результатом такой операции является специальное чисто праводопустимое значение, пригодное только в качестве первого операнда операции вызова функции. Несмотря на его категорию значения, оно не только идентифицирует вызываемую функцию, но и определяет дополнительный аргумент в операции вызова. С точки зрения разрешения перегрузок при рассмотрении не статической функции-кандидата, она имеет дополнительный ***неявный параметр-объект (implicit object parameter)*** типа «ссылка на объект того типа, членом которого она является». Единственный способ задания значения этого параметра при вызове функции — использование операции выборки для доступа к функции через конкретный объект, задать его явно в списке аргументов нельзя, даже если функция поименовано верно, например, через её полное имя. При таком вызове говорят, что «функция вызвана *на* объекте» в смысле применения алгоритма к объекту. В теле не статической функции ключевое слово `this` является чисто праводопустимым выражением со значением адреса объекта, для которого была вызвана эта функция:

```
1   struct S
2   {
3       int x;
4
5       // Функция f - не статический член класса.
6       // Её параметры с точки зрения разрешения перегрузок:
7       // - неявный параметр-объект типа ссылка на S без имени -
8       //   объект, для которого она вызвана.
9       // - new_x типа int.
10      void f(int new_x);
11  };
12
13  // Определение S::f.
14  void S::f(int new_x)
15  {
16      // Изменение члена данных объекта,
17      // для которого вызвана функция.
18      this->x = new_x;
19  }
20
21  void g()
22  {
23      S s{};
24
25      // Вызов функции f для объекта s.
26      // Аргументы вызова:
27      // - Неявный параметр-объект - s
28      // - new_x - 42.
29      // В этом вызове этой функции
30      // this в её теле будет равен &s.
31      s.f(42);
32
33      // ОШИБКА: вызов не статической функции возможен
34      //         только с указанием объекта, на котором
35      //         она вызвана, с помощью выборки её имени
36      //         через объект.
37      S::f(s,42);
38  }
```

Если вспомнить примеры абстрактных типов данных, рассмотренных нами ранее, становится понятно, что явный параметр, использовавшийся для передачи адреса объекта, которым должна оперировать функция, можно опустить при описании функции как не статического члена класса и использовать вместо него `this`.

Поиск не квалифицированных имён в телах функций-членах класса и их аргументах по умолчанию имеет следующие отличия от поиска имён в других местах определения класса:

- В самом начале просматривается последовательность блоков до использования имени, как и в обычной функции (если они есть).
- Поиск имён-членов класса, в члене которого использовано это имя, может находить имена, описанные как до, так и после его использования.

Пример:

```
class X
{
    void f()
    {
        int x;
        {
            // x найден в окружающем блоке,
            // как и в обычной функции.
            x = 42;
        }
        // T найдено в классе X,
        // хотя описано после использования.
        T y = 0;
    }

    using T = int;
};
```

Если найденное имя в теле не статической функции соответствует не статической функции или члену данных, считается, что имя было найдено в результате косвенной выборки из `this`, т.е. применяется на или является частью того же объекта, который обрабатывает текущая функция. То же «приписывание `this->`» осуществляется и при квалифицированном поиске, если он ищет итоговое имя в том же классе, членом которого является текущая функция.

```
struct S
{
    using my_int = int;

    my_int x;

    void zero()
    {
        // x найдено как член S
        // для объекта, на котором
        // вызвана эта функция.
        /* this-> */ x = 0;
    }

    void f()
    {
        // zero найдено как член S
        // и вызвано на том же объекте,
        // что и текущая функция.
        /* this-> */ zero();

        // То же самое.
        ::S::zero();

        // my_int найдено в ::S.
        /* ::S:: */ my_int y = 42;
    }

    void set_x(int x)
    {
        // Имя члена структуры S x
        // скрыто в этой функции
```

```
33          // одноимённым параметром,
34          // и, чтобы именовать его,
35          // требуется явный доступ через this.
36          this->x = x;
37      }
38  }
```

Как показано в примере выше, в большинстве случаев использование ключевого слова `this` и многих квалифицированных имён не требуется, что упрощает тела функций-членов по сравнению с вариантом на языке C, где обрабатываемый функцией объект всегда требуется именовать явно.

Неявный параметр-объект, привязываемый к объекту типа `S`, на котором вызывается функция, имеет по умолчанию формальный тип `S&`, но, в отличии от обычной леводопустимой ссылки на не-`const`, может привязываться даже к значениям, не являющимся леводопустимыми выражениями. С другой стороны, как и для обычной ссылки отбрасывание квалификаторов при привязке недопустимо:

```
1   struct S
2   {
3       void f()
4       {
5           // ...
6       }
7   };
8
9   void g()
10  {
11      // Неявный параметр-объект, на котором
12      // вызывается функция может быть привязан
13      // к временному объекту.
14      S{}.f();
15
16      const S s{};
17
18      // ОШИБКА: отбрасывание квалификатора
19      //         при привязке неявного параметра-объекта.
20      s.f();
21  }
```

Чтобы, в том числе, разрешить вызов не статических функций-членов на неизменяемых или иначе квалифицированных объектах, квалификаторы базового типа для неявного параметра-объекта могут быть указаны в конце описания после списка параметров. Эта возможность позволяет иметь несколько перегрузок одной функции по типу этого параметра:

```
1   struct S
2   {
3       int x[10];
4
5       // Тип неявного параметра-объекта - S&.
6       // Эта функция может быть вызвана
7       // только на изменяемом объекте.
8       int& get(std::size_t i) // 1
9       {
10          return x[i];
11      }
12
13      // Тип неявного параметра-объекта - const S&.
14      // Эта функция может быть вызвана
15      // на объекте с любыми квалификаторами,
16      // кроме volatile.
17      // Для неизменяемых объектов только она
18      // и годна, и, поскольку члены данных
19      // в ней получают тот же квалификатор,
20      // возвращает ссылку на квалифицированное
21      // значение.
22      // Она также годна для изменяемых объектов,
```

```
23      // но предыдущая перегрузка для них лучше
24      // по правилам ранжирования привязок ссылок.
25      const int& get(std::size_t i) const // 2
26      {
27          return x[i];
28      }
29  };
30
31  void f()
32  {
33      S s1;
34      const S s2;
35
36      // Вызывает перегрузку 1.
37      s1.get(3) = 5;
38
39      // ОШИБКА: вызывает перегрузку 2,
40      //         возвращающую ссылку на
41      //         неизменяемый объект.
42      s2.get(4) = 6;
43
44      // Вызывает перегрузку 2.
45      int x = s2.get(4);
46  }
```

Необходимость учёта квалификаторов при вызове не статических функций-членов является причиной того, что в C++ значения классовых типов могут иметь квалификаторы, даже не являясь леводопустимыми:

```
46  // Продолжение примера выше.
47
48  // Квалификатор на возвращаемом значении классового
49  // типа осмыслен и сохраняется, несмотря на то,
50  // что оно не является леводопустимым.
51  const S g()
52  {
53      return {};
54  }
55
56  void h()
57  {
58      // Вызов перегрузки 2.
59      g().get(5);
60  }
```

В функции-члене, имеющей квалификаторы неявного объекта-параметра, они распространяются на базовый тип указателя `this` и, следовательно, на все не статические члены данных класса:

```
1   struct S
2   {
3       int x;
4
5       void modify_stuff()
6       {
7           // ...
8       }
9
10      int get_x() const
11      {
12          // ОШИБКА: при поиске имён
13          //         неявно подставляется
14          //         this, которое в этой
15          //         функции имеет тип const S*,
16          //         поэтому x - неизменяемое значение.
17          /* this-> */ x = 42;
```

```
18
19          // ОШИБКА: разыменование this даёт неизменяемое
20          //         значение, которое не может быть привязано
21          //         к ссылке на изменяемое значение,
22          //         выступающее в качестве неявного
23          //         объекта-параметра функции modify_stuff.
24          /* this-> */ modify_stuff();
25
26          // Читать неизменяемое значение можно.
27          return x;
28      }
29  };
```

По константности неявного объекта-параметра не статические функции класса концептуально делят на ***наблюдателей (observer)*** (с `const`) и ***мутаторы (mutator)*** (без `const`). ***Не забывайте давать всем функциям, которые не меняют состояния объекта, квалификатор неявного объекта-параметра `const` — это одно из важных проявлений `const`-корректности, обеспечивающее возможность работы функций-членов на неизменяемых объектах.***

Квалификатор `const` на функциях членах следует трактовать с логической, а не физической точки зрения — функции, его имеющие, не меняют *видимого* снаружи состояния объекта. Иногда требуется в такой функции изменить значение члена данных объекта, не являющегося частью видимого состояния объекта, например, кешированные данные. Отменить действие квалификатора `const` при доступе к не статическому члену данных из `const`-функций можно с помощью спецификатора `mutable` в его описании:

```
1   struct S
2   {
3       int x;
4       mutable int y;
5
6       void f() const
7       {
8           // ОШИБКА: x здесь не изменяем.
9           x = 42;
10
11          // Ошибки нет, mutable в описании
12          // y отменяет const.
13          y = 42;
14      }
15  }
```

Транслятор, в свою очередь, увидев хотя бы одно описание члена класса с модификатором `mutable`, узнаёт, что физическая неизменяемость блока памяти не соответствует понятию «неизменяемости» для объекта и никогда не будет размещать его в секции памяти с атрибутами только для чтения.

Хотя свойство привязки неявного параметра-объекта к значениям любых категорий обычно полезно, бывает необходимо различать их, как и для явных параметров функций. В таком случае все перегрузки соответствующей функции могут содержать ***квалификаторы ссылки (ref-qualifiers)***. Они также пишутся в конце описания функции и состоят из требуемых квалификаторов, за которыми следует пунктуатор `&` или `&&`, определяющий требуемый вид ссылки. Для таких функций применяются обычные правила привязки к неявному параметру-объекту без исключений.

```
1   struct S
2   {
3       // Тип неявного параметра-объекта - S&.
4       // Исключения на привязку к нему не леводопустимых
5       // значений нет.
6       void f() &
7       {
8           // ...
9       }
10
```

```
11      // ОШИБКА: квалификаторы-ссылки должны быть
12      //         либо у всех перегрузок функции,
13      //         либо ни у одной.
14      void f() const;
15
16      // Тип неявного параметра-объекта, const S&.
17      void g() const & // 1
18      {
19
20      }
21
22      // Тип неявного параметра-объекта - S&&.
23      // Эта перегрузка лучше прошлой
24      // для временных объектов.
25      void g() && // 2
26      {
27
28      }
29  };
30
31  void f()
32  {
33      // ОШИБКА: не леводопустимое значение временного
34      //         объекта не привязывается к
35      //         неявному параметру-объекту
36      //         типа S&.
37      S{}.f();
38
39      S s;
40
41      // Вызывается перегрузка 1.
42      s.g();
43
44      // Вызывается перегрузка 2.
45      S{}.g();
46  }
```

9.6.2. Статические члены классов

Статическими (static) являются функции-члены и члены данных класса, описанные со спецификатором `static`. Общий смысл статических членов в том, что они относятся не к конкретному объекту-экземпляру класса, а ко всему классу в целом.

Статические члены данных не участвуют в описании устройства классового типа в памяти. Вместо этого каждый из них описывает единственный объект со статическим временем хранения, который использует область видимости класса подобно пространству имён. Описания статических членов данных в классе никогда не являются определениями, которые для них должны быть даны отдельно вне определения самого класса:

```
1  struct S
2  {
3      // Описание статического члена данных.
4      static int x;
5  };
6
7  // Определение статического члена данных S::x.
8  int S::x = 42;
```

Доступ к статическим членам данных может осуществляться теми же способами, что и к не статическим, но при этом указания, к члену какого объекта следует обращаться, игнорируются, так как они не связаны ни с одним из объектов. Помимо этого доступ к ним можно осуществить вообще без указания объекта путём записи их квалифицированного имени, содержащего имя класса, членами которого они являются (или без него, если они будут найдены и не квалифицированным поиском):

```
9  // Продолжение предыдущего примера.
10
```

```
11  void f()
12  {
13      S obj;
14
15      // Доступ к статическому члену данных может быть
16      // получен как с указанием объекта (игнорируемым),
17      // так и без него.
18      assert(&obj.x==&S::x);
19  }
```

Статические функции-члены не имеют неявного параметра-объекта, поэтому ключевое слово `this` и квалификаторы неявного объекта-параметра с ними использовать нельзя. Они также могут быть вызваны как через операцию выборки (объект игнорируется), так и через отдельное имя. В статических функциях членах без указания объекта через операцию выборки доступны только статические члены класса. Таким образом, как и статические члены данных, статические функции-члены используют класс как отдельную область видимости без каких-либо других специальных возможностей, характерных для не статических функций-членов.

```
1   struct S
2   {
3       int a;
4
5       static int b;
6
7       void f()
8       {
9           // ...
10      }
11
12      static void g();
13
14      static void h(S& s)
15      {
16          // ОШИБКА: не статический член данных
17          //         в статической функции без указания объекта не доступен.
18          a = 5;
19
20          // Операция выборки работает как обычно.
21          s.a = 5;
22
23          // ОШИБКА: не статическая функция не может
24          //         быть вызвана из статической
25          //         без указания объекта.
26          f();
27
28          // Находит и вызывает ::S::g.
29          g();
30      }
31  }
32
33  void f()
34  {
35      S obj;
36
37      // Вызов статической функции, объект s
38      // игнорируется.
39      obj.g();
40
41      // Вызов статической функции без указания
42      // объекта с помощью квалифицированного имени.
43      S::g();
44  }
```

Перегрузки функций-членов класса, отличающиеся только тем, что одна статическая, а другая — нет, недопустимы, но в общем случае среди всех перегрузок одной и той же функции могут быть как статические, так и не статические. При этом действуют обычные правила

разрешения перегрузок, в контекстах, в которых неявный параметр-объект известен, он подставляется в качестве аргумента не статических функций, иначе они не являются годными. Неявный параметр-объект при сравнении его соответствия со статическими функциями, где он не фигурирует, считается не сравнимым.

```
1  struct S
2  {
3      void f(int);            // 1
4
5      // ОШИБКА: перегрузка отличается только
6      //         по отсутствию/наличию static.
7      static void f(int);
8
9      static void f(double); // 2
10     void f(void*);         // 3
11 };
12
13 void f()
14 {
15     S obj;
16
17     // Годны 1 и 2
18     // (для 1 подставлен неявный параметр-объект).
19     // По явному параметру 2 лучше, она и вызывается,
20     // объект игнорируется.
21     obj.f(4.2); // 2
22
23     // Вызов без указания неявного объекта-параметра,
24     // годны только статические функции, то есть 2.
25     S::f(3); // 2
26 }
```

9.6.3. Вложенные и локальные классы

Как было сказано ранее, определения классов могут содержать в себе другие описания классовых типов — такие классы называют ***вложенными (nested)***. Если такое описание не является определением, соответствующее определение может быть дано позднее в том же классе или в окружающем пространстве имён:

```
1  struct A
2  {
3      // Определение класса B,
4      // вложенного в A.
5      struct B
6      {
7          // ...
8      };
9
10     // Описания классов B1 и B2,
11     // вложенных в A.
12     struct B1;
13     struct B2;
14
15     // B1 видно, но является неполным типом.
16     B1* ptr;
17
18     // Определение вложенного класса B1,
19     // описанного ранее.
20     struct B1
21     {
22         // ...
23     };
24 };
25
26 // Определение вложенного класса A::B2,
27 // описанного в определении A.
```

```
28  struct A::B2
29  {
30      // ...
31  };
32
33  // ОШИБКА: в A нет описания B3.
34  struct A::B3
35  {
36      // ...
37  };
```

При не квалифицированном поиске имён в описаниях вложенного класса или его членов после просмотра его собственной области видимости и перед просмотром первого окружающего пространства имён просматривается область видимости класса, для которого данный является вложенным и далее все остальные классы, в которые может быть вложен только что просмотренный. Описания, находящиеся в классах, в которые вложен данный, расположенные после использования имени, находятся тогда и только тогда, когда подобное допустимо для членов самого класса:

```
 1  namespace A
 2  {
 3      struct B
 4      {
 5          using my_int = int;
 6
 7          struct C
 8          {
 9              void f();
10          };
11      };
12  }
13
14  void A::B::C::f()
15  {
16      // Поиск my_int просматривает по порядку:
17      // - блок-тело функции,
18      // - класс C, членом которого она является,
19      // - класс B - здесь находится my_int.
20      // Если бы описания в B не было, далее
21      // просматривались бы:
22      // - пространство имён ::A
23      // - глобальное пространство имён.
24      my_int x;
25  }
26
27  struct C
28  {
29      struct D
30      {
31          // Поскольку поиск имён в теле функции-члене
32          // класса может находить имена, описанные в
33          // содержащем её классе и после их использования,
34          // он также может находить описания из классов,
35          // в которые вложен этот класс после использования имени.
36          T x;
37      };
38
39      using T = int;
40  };
```

Вложенные классы используют для описания вспомогательных структур данных, которые не нужны вне класса. За счёт модифицированного поиска имён доступ к типам и статическим функциям класса, в который они вложены, в них упрощается.

Классы, описанные в блочной области видимости, называются ***локальными (local)***, мы уже использовали их в примерах, но на них наложен ряд ограничений. Такие классы должны содержать все определения своих членов в своём определении, поэтому статические члены

данных (которые требуют внешнего определения) в них полностью запрещены, а все описания функций-членов в них должны являться определениями и при этом в качестве исключения не имеют связанности. Классы, вложенные в локальные, могут быть определены в том же самом классе или в той же (блочной) области видимости, что и класс, в который они вложены, они сами тоже считаются локальными.

Не квалифицированный поиск имён в локальных классах дополнительно просматривает блоки, в которых определён локальный класс до поиска в окружающих пространствах имён. Несмотря на это, доступа к объектам с автоматическим временем хранения локальные классы не имеют.

```
1  void f()
2  {
3      int x = 0;
4      static int y = 0;
5
6      // Локальный класс A.
7      struct A
8      {
9          // Локальный класс B,
10         // вложенный в A.
11         struct B
12         {
13             void f()
14             {
15                 // Последовательность неквалифицированного
16                 // поиска имён в этом блоке такова:
17                 // - текущий блок-тело функции,
18                 // - класс B, членом которого является
19                 //   эта функция,
20                 // - класс A, в который вложен B,
21                 // - тело функции f, для которой
22                 //   этот класс является локальным
23                 // - глобальное пространство имён.
24
25                 // ОШИБКА: локальный класс не имеет
26                 //         доступа к автоматическим
27                 //         объектам функции, в
28                 //         которую вложен.
29                 x = 1;
30
31                 // Изменяет объект со статическим
32                 // временем хранения, описанный в f.
33                 y = 1;
34             }
35         };
36     };
37 }
```

Локальные классы бывают полезны для описания структур данных, необходимых всего в одной функции, чтобы это определение было недалеко в тексте от его использования.

9.6.4. Контроль доступа к членам классов

Чтобы реализовать скрытие реализации класса, мы пока прибегали исключительно к устной договорённости не осуществлять прямой доступ к членам данных класса, а работать только с функциями его интерфейса. В языке C++ есть встроенные средства контроля доступа к членам классовых типов.

Уровни доступа применимы ко всем возможным видам членов класса. Вначале мы познакомимся с двумя уровнями доступа: ***открытым (public)*** и ***закрытым (private)***. Открытые члены класса доступны всем. Это тот уровень доступа, которым мы пользовались до этого раздела, это единственно существующий уровень доступа в языке C (которых в нём формально нет). Закрытые члены класса могут использоваться только другими членами того же класса. Проверка на доступность осуществляется исключительно на уровне имён, а не смысла описания, и производится в конце процедуры поиска имён и после разрешения перегрузок, если речь шла о функции. Программа, пытающаяся именовать недоступное в месте использования

(явном или неявном) имя, не согласована. Порядок расположения в памяти не статических членов данных класса с разными уровнями доступа не уточняется.

Уровни доступа для членов класса определяются путём размещения между ними маркеров из ключевого слова, соответствующего названию уровня доступа, и двоеточия, все последующие описания до следующего такого маркера получают соответствующий уровень доступа. Секции описаний, соответствующие уровням доступа, могут быть пустыми и расположены в любом количестве и порядке. Расположенные до первого маркера уровня доступа описания получают уровень доступа, определяемый ключевым словом в заголовке определения класса: `private` для `class` и `public` для `struct` и `union`:

```
struct S
{
    int x;        // открытый
private:
    int y;        // закрытый

    void f(S& s); // закрытый
};

class C
{
    using my_int = int; // закрытый
public:
    // x - член C, поэтому имеет
    // доступ к закрытому определению my_int
    my_int x;           // открытый
private:
public:
public:
    // Ещё пустые секции.
};

void S::f(S& s)
{
    // Член класса имеет доступ к
    // именам с любым уровнем доступа.
    x = y = 42;

    // Поскольку проверка осуществляется
    // на уровне имён членов, объект,
    // к членам которого осуществляется
    // доступ, значения не имеет - f
    // имеет доступ к закрытым членам как
    // неявного параметра-объекта, так
    // и любого другого объекта типа S.
    s.x = s.y = 42;
}

void g()
{
    S s;

    // К открытому члену доступ имеют все.
    s.x = 42;

    // ОШИБКА: g - не член S
    //         и не имеет доступа к
    //         закрытому члену y.
    s.y = 42;

    // ОШИБКА: g - не член C
    //         и не имеет доступа к
    //         закрытому члену my_int.
    C::my_int i = 0;

    C c;
```

```
57
58      // К открытому члену my_int доступ
59      // имеют все. То, что его тип описан
60      // с использованием закрытого имени
61      // значения не имеет - проверяется
62      // доступ к самому имени x, а не его
63      // типу.
64      c.x = 15;
65  }
```

Больше ничем, кроме уровня доступа членов по умолчанию, в описаниях классовых типов ключевые слова `class` и `struct` не отличаются.

Разные перегрузки одной функции могут иметь разные уровни доступа, тот факт, что контроль доступа применяется только после выбора лучшей из них может приводить к некоторым неприятным эффектам:

```
1   struct S
2   {
3   private:
4       void f(int);    // 1
5   public:
6       void f(double); // 2
7   };
8
9   void g()
10  {
11      // ОШИБКА: выбрана перегрузка 1,
12      //         как лучшая, но она отсюда
13      //         не доступна.
14      //         Ошибка, несмотря на то, что
15      //         есть доступные годные перегрузки!
16      S{}.f(34);
17  }
```

Для членов, которые могут быть описаны несколько раз в определении класса, все описания должны иметь один уровень доступа:

```
1   struct A
2   {
3       class B;
4   private:
5       // ОШИБКА: уровень доступа отличается
6       //         от прошлого описания.
7       class B
8       {
9           // ...
10      };
11  }
```

Вложенные классы считаются членами класса, в котором описаны, и как его члены имеют доступ и к закрытым его членам. Когда говорят, что доступ имеет весь класс, имеют в виду доступ из всех его описаний и определений его членов. В обратном направлении никаких привилегий содержащий класс к своему вложенному не имеет.

```
1   class A
2   {
3       static int x;
4
5       void f();
6
7       class B
8       {
9            static int x;
10
11           void f();
12      }
```

```
};

void A::f()
{
    // ОШИБКА: x - закрытый член B.
    B::x = 42;
}

void B::f()
{
    // x - закрытый член A,
    // но B::f - член B, который
    // сам член A и потому имеет доступ.
    A::x = 42;
}
```

Классы с хотя бы одним не открытым не статическим членом перестают считаться агрегатными для целей инициализации их непустыми списками инициализации.

```
struct S
{
    int x;
private:
    int y;
};

void f()
{
    // ОШИБКА: объект класса с не открытыми
    //         не статическими членами не
    //         может быть инициализирован
    //         не пустым списком.
    S s{1};
}
```

Негласным соглашением является использование `struct` для классов, все члены которых открытые — это простые или для внутреннего использования другими частями программы типы, не имеющие или не заботящиеся о своих инвариантах, например, классовый тип точки на плоскости с двумя членами данных — вещественными координатами.

```
#include <cmath>

// Точка на плоскости. Забыв про возможность наличия
// специальных значений у вещественного типа
// (бесконечности, NaN, ...) будем считать,
// что все возможные значения членов этого класса
// корректны, т.е. инвариантов у него нет.
struct point
{
    double x,y;

    // Длина радиус-вектора.
    double rv_len() const
    {
        return std::sqrt(x*x+y*y);
    }
};
```

Большинство классов предназначены для общего использования и имеют инварианты, поддерживать которые является основной задачей инкапсуляции. В таком случае все члены данных класса делают закрытыми, а доступ к ним осуществляется через открытые ***функции доступа (accessor)***. В англоязычной литературе функции-наблюдатели для чтения членов данных называют ***getter*** («получатель»), а мутаторы для изменения — ***setter*** («установщик»).

```
class C
{
```

```
3       // Закрытый член данных.
4       int x;
5   public:
6       // Функции доступа к x.
7       int get_x() const
8       {
9           return x;
10      }
11
12      void set_x(int new_x)
13      {
14          x = new_x;
15      }
16  }
17
18  void f()
19  {
20      C c;
21
22      // Запись x.
23      c.set_x(42);
24
25      // Чтение x.
26      assert(c.get_x()==42);
27  }
```

Существуют несколько общепринятых стилей именования функций доступа:

- `get_`имя и `set_`имя — функции с именами из префикса get/set и имени члена данных, регистр и разделитель варьируются в соответствии с общим стилем именования идентификаторов.
- имя и `set_`имя — то же, но имя функции члена соответствует имени члена данных. Сам член данных в таком случае, чтобы не конфликтовать с этой функцией получает некоторый префикс или суффикс к имени, автор предпочитает суффикс `_`.
- Функции чтения и записи называются одинаково и являются перегрузками — перегрузка без параметров возвращает считанное значение, перегрузка с одним параметром меняет значение на указанное и возвращает `void`. Этот стиль используется, в основном, стандартной библиотекой языка C++.

Такое, на первый взгляд, значительное усложнение простейших операций чтения и записи, тем не менее, несёт с собой преимущества. Основной идеей является отделение интересующей величины, которую можно читать и/или записывать применительно к классу от конкретного объекта-члена данных, хранящего её значение. В этом смысле объект классового типа обладает ***свойствами (property)***, которые можно читать и/или писать, а где и как они хранятся (если вообще) — деталь реализации. Понятие «свойства» в некоторых объектно-ориентированных языках имеет синтаксическую составляющую, но в C++ оно существует только на концептуальном уровне.

Во-первых, можно реализовать «свойства только для чтения». Если просто объявить член данных неизменяемым, изменить его не сможет вообще никто, но если сделать его изменяемым и закрытым, предоставив только открытую функцию получения его значения, внешне оно будут доступно только для чтения, хотя функции-члены класса смогут его изменять (с сохранением инвариантов).

Во-вторых, контроль над операциями записи позволяет проверять корректность записываемых значений, исправлять их, или для поддержания инвариантов обновлять и другие члены класса, а также выполнять любые иные действия.

Имеется и множество других способов реализации функций доступа. В наиболее сложных случаях функции доступа вообще могут не читать/писать значения членов класса, а осуществлять доступ, например, к внешним устройствам, отражая их свойства. Даже в показанной в самом начале тривиальной форме функций доступа есть смысл, поскольку это позволит в будущем заменить простые операции чтения/записи в них более сложными, сохранив интерфейс класса без изменений, чтобы не пришлось исправлять все места в другом коде, где к нему осуществляется доступ, заменяя простые операции доступа к членам объекта вызовом функций доступа.

Таким образом, классы с нестатическими членами данных разных уровней доступа обычно не требуются. Если такие классы всё же имеются, для них порядок выделения памяти

подобъектов в представлении, соответствующий порядку описаний, гарантируется только отдельно в рамках каждого отдельного уровня доступа.

| Внедрённое имя класса всегда имеет открытый уровень доступа. |

9.6.5. Инициализация классов

Конструкторы (constructor) — особые функции-члены класса, отвечающие за инициализацию объектов его типа. Они вызываются автоматически сразу после выделения под хранение объекта памяти в результате определения объекта или использования операции `new`. Обычным образом, как функция-член объекта, они вызваны быть не могут.

Может существовать сколько угодно перегрузок конструкторов у одного класса, некоторые из них играют особую роль и имеют свои названия. Объект с хотя бы одним конструктором, предоставленным пользователем, перестаёт считаться агрегатом для целей задания начальных его значений списком инициализации — вместо этого все значения, указанные в таком списке или круглых скобках в различных синтаксисах инициализации, рассмотренных нами ранее, используются для выбора перегрузки конструктора, которому и передаются для инициализации объекта — это позволяет объектам полностью контролировать все возможные свои начальные значения, избегая мусора и некорректных в соответствии со своими инвариантами значений. Синтаксис прямой инициализации может содержать для объектов несколько аргументов. При инициализации по умолчанию или по значению после выполнения уже известных нам действий конструктор вызывается без аргументов. Если при вызове конструктора выбрать подходящую перегрузку не удаётся, то создание объекта с указанным инициализатором (или без него) невозможно. Конструктор с единственным параметром того же типа (без учёта квалификаторов), что и сам класс, в котором он содержится, недопустим, так как потребовался бы для инициализации собственного параметра, что повлекло бы за собой бесконечную рекурсию (то же касается случая, когда кроме этого параметра у всех остальных есть аргументы по умолчанию).

Конструкторы — не статические члены класса без квалификаторов неявного объекта-параметра. Они не имеют имён, в их описании вместо имени функции указывается имя, которое должно соответствовать самому классу, содержащему этот конструктор. Возвращаемое значение конструктора также не указывается, но внутри его тела может использоваться оператор `return` без выражения, как будто оно `void`.

```
1  struct S
2  {
3      // Определения трёх конструкторов класса S.
4      S() { /* ... */ }                  // 1
5      S(int,double = 4.2) { /* ... */ } // 2
6      S(char,float) { /* ... */ }        // 3
7  };
8
9  // Инициализация по умолчанию вызывает S::S().
10 S s1; // 1
11
12 // Инициализация по значению сначала инициализирует
13 // объект нулём, а потом вызывает S::S().
14 S s2{}; // 1
15
16 // Прямая инициализация вызывает конструктор,
17 // определяемый по заданным аргументам с
18 // помощью разрешения перегрузок.
19 S s3(5),         // 2
20   s4('x',5.7f); // 3
21
22 // То же для инициализации списком напрямую
23 // и копированием.
24 S s5{0,15.},         // 2
25   s6 = {'x',6.8f};  // 3
26
27 // Вызов конструкторов при создании временных
28 // объектов (явно или неявно) осуществляется по
29 // тем же правилам.
30 S(),S{},S(5.9),S{5.9},S('t',0.f),S{'t',0.f};
```

```
31
32 // ОШИБКА: нет подходящего конструктора для
33 //         инициализации объекта типа S
34 //         из трёх аргументов.
35 S(1,2,3);
```

Неявный параметр-объект в теле конструктора соответствует инициализируемому объекту. Основной задачей конструктора является инициализация подобъектов, для чего в синтаксисе его определения может применяться особая дополнительная конструкция, которой могут обладать только конструкторы — ***список инициализаторов членов (member initializer list)***.

Инициализация не статических членов класса происходит в порядке их описаний в определении класса следующим образом:

- Если для этого члена есть инициализатор в списке инициализаторов членов вызванного конструктора, он используется для прямой инициализации или инициализации по значению.
- Если в списке инициализаторов членов инициализатора нет, но в описании члена имеется инициализатор в одной из форм инициализации списком или копированием, используется он. Такие инициализаторы иногда называют специальной аббревиатурой ***NSDMI (Non-Static Data Member Initializers***, поскольку она появилась относительно недавно в C++11. Не квалифицированные имена в таких инициализаторах ищутся по правилам, аналогичным поиску имён в телах функций-членов.
- Если в двух указанных выше местах инициализатора нет, подобъект инициализируется по умолчанию.

После инициализации всех подобъектов выполняется тело конструктора.

Список инициализаторов членов — это разделённый запятыми список имён не статических членов класса с инициализаторами в форме круглых или фигурных скобок, опционально указываемый в определении конструктора между его описанием и телом. Имена членов ищутся сначала в области видимости конструируемого класса, а затем в области видимости, где находится определение конструктора (обычно эта возможность не используется). Имена в выражениях-инициализаторах ищутся по обычным правилам поиска имён в теле функции.

```
1  struct S
2  {
3      unsigned a;
4      unsigned b = 2;
5
6      // При использовании этого конструктора без
7      // списка инициализации членов,
8      // a инициализируется по умолчанию, а
9      // b инициализируется значением 2, указанным
10     // в его описании.
11     // После этого выполняется тело конструктора
12     // и выводится сообщение.
13     S()
14     {
15         std::cout << "Constructed without arguments.";
16     }
17
18     // При использовании этого конструктора
19     // a инициализируется значением x, а
20     // b инициализируется значением 2.
21     // После этого выполняется пустое тело конструктора.
22     S(unsigned x)
23         : a(x)
24     {}
25
26     // При использовании этого конструктора сначала
27     // a инициализируется по умолчанию, т.к. отсутствует
28     // в списке инициализаторов членов этого конструктора,
29     // а затем b инициализируется значением x+y,
30     // имеющим приоритет над данным в описании члена.
31     // Затем выполняется тело конструктора,
32     // записывающее в a новое значение поверх
```

```
33      // полученного им в результате инициализации.
34      S(unsigned x,unsigned y)
35          : b(x+y)
36      {
37          a = x-y;
38      }
39  };
```

Объекты классов, не имеющих конструктора, который можно вызвать с пустым списком аргументов, но не являющиеся агрегатами, а также неизменяемые объекты и ссылки — это примеры членов класса, которые не могут быть созданы без указания непустых инициализаторов. Для них обязательно должны быть даны инициализаторы в их описаниях в определении класса или в списке инициализации членов каждого из конструкторов, иначе их невозможно будет инициализировать, а вместе с ними и весь класс.

В общем случае, чтобы избежать лишней инициализации по умолчанию, которая может быть нетривиальной, принято задавать начальные значение членов именно списком инициализаторов конструктора, а не присваиваниями в его теле там, где это возможно.

Отметим также, что инициализация подобъектов происходит в порядке их описаний в определении класса и не зависит от порядка указания инициализаторов в списке инициализаторов членов конструктора.

```
1   struct A
2   {
3       int a,b;
4
5       // Описание конструктора.
6       A(int x);
7   };
8
9   // Определение конструктора вне класса.
10  A::A(int x)
11  // ОШИБКА: программист хотел сначала инициализировать
12  //         b значением x+1, а потом a значением b*2,
13  //         но порядок инициализации определяется
14  //         только порядком описаний членов в классе,
15  //         и первым инициализируется a, используя
16  //         ещё не инициализированное значение b.
17  //         Хороший компилятор выдаст предупреждение.
18      : b(x+1),a(b*2)
19  {
20  }
21
22  struct B
23  {
24      int& a;
25      const int b;
26
27      B(int& x)
28      // Ссылки, неизменяемые члены, и объекты классов
29      // требуют инициализаторов, которые в теле конструктора
30      // не указать.
31          : a(x)
32      // ОШИБКА: b в списке инициализации отсутствует, но
33      //         по умолчанию инициализирован быть не может,
34      //         так как неизменяем.
35      {}
36  };
```

Задание инициализаторов не статических членов непосредственно в их описаниях является одним из способов объединения одинаковой требуемой инициализации между несколькими конструкторами. Кроме этого, один конструктор может ***делегировать (delegate)*** инициализацию объекта другому конструктору. В этом случае его список инициализаторов содержит единственный элемент, именующий сам инициализируемый класс, с требуемыми аргументами для вызова другой перегрузки конструктора (вызов той же самой перегрузки, напрямую или

косвенно, некорректен, т.к. приводит к бесконечной рекурсии). После выполнения конструктора, которому была делегирована инициализация (вместе с его телом), выполняется тело текущего конструктора.

```
struct point
{
    double x,y;

    // Делегирующий конструктор.
    point()
        : point(0.)
    {
    }

    // Конструкторы могут делегировать инициализацию
    // по цепочке несколько раз, если это не ведёт
    // к циклам.
    point(double x)
        : point(x,0.)
    {
    }

    point(double x,double y)
        : x(x),
          y(y)
    {
    }

    /*
    В данном случае проще было обойтись одним конструктором
    вида
    point(double x = 0.,double y = 0.);
    но когда список инициализаторов членов и тело
    более сложные, этот подход имеет преимущества.
    */
};

// При инициализации p:
// - Нуль-арный конструктор делегирует
//   инициализацию унарному с аргументом 0.
// - Унарный конструктор делегирует
//   инициализацию бинарному с аргументами 0. и 0.
// - Бинарный конструктор инициализирует члены
//   объекта переданными ему значениями и выполняет
//   своё тело (пустое).
// - Выполняется тело унарного конструктора (пустое).
// - Выполняется тело нуль-арного конструктора (пустое).
point p;
```

Квалификаторы типов начинают влиять на объект только после того, как он инициализирован конструктором, что позволяет инициализировать обычным образом даже неизменяемые объекты:

```
struct S
{
    int a,b;

    S()
        : a(10)
    {
        b = 20;
    }
};

// Обычный вызов конструктора S::S
// с точки зрения которого квалификатора
```

```
14  // const на неявном объекте-параметре нет.
15  const S s;
```

Пока неразрешимой по-настоящему для нас является проблема сигнализирования об ошибках конструирования объектов — конструктор не имеет возвращаемого значения. В качестве временного решения можно использовать специальное «некорректное» состояние объекта, которое можно проверить после его создания, например, оставив нулевой указатель храниться в члене класса в качестве адреса не выделенного блока динамической памяти или структуры потока ввода-вывода. Для проверки класса на такое состояние можно создать дополнительную функцию `is_valid`, возвращающую логическое значение. Повторимся, что это не красивое и не правильное решение — настоящее решение этой проблемы мы узнаем позднее.

Конструкторы объектов со статическим временем хранения вызываются аналогично другим функциям в динамическую фазу инициализации.

9.6.6. Пользовательские преобразования типов

Без дополнительных указаний классовый тип может участвовать только в приведении его значений к `void`. Кроме этого классовые типы могут явно определять, как происходит приведение типов к их типу из других, и из них в другие, для чего используются конструкторы и функции преобразования.

Конструктор преобразования (conversion constructor) позволяет преобразовывать значения других типов к типу класса. Все применявшиеся нами до этого конструкторы по сути задают преобразование из списка своих аргументов (даже пустого) и потому считаются конструкторами преобразования.

Преобразования из классового типа в другой задаётся ***функцией преобразования (conversion function)***. Это не статическая функция-член класса без параметров, имеющая имя, состоящее из ключевого слова `operator`, за которым следует имя результирующего типа. Тип возвращаемого значения такой функции в описании отдельно не указывается, но фактически им является результирующий тип, включённый в имя этой функции. Такие функции чаще всего имеют квалификатор неявного объекта-параметра `const`, т.к. преобразование по смыслу не затрагивает сам объект, а создаёт новое значение.

Приведём пример:

```
1   class C
2   {
3       int x;
4   public:
5       // Конструктор преобразования из int.
6       C(int x)
7       : x(x)
8       {
9       }
10
11      // Функция преобразования в int.
12      operator int() const
13      {
14          return x;
15      }
16  };
```

Функции преобразования не могут осуществлять преобразования к типам категории «массив» или «функция», т.к. функции с такими возвращаемыми значениями недопустимы.

> Описания функций преобразования в `void` или сам тип класса допускаются, но использоваться никогда не будут.

При задании преобразования между двумя классовыми типами имеется выбор между реализацией в первом конструктора преобразования из второго, или реализация во втором функции преобразования в первый. Если один из этих классов не может быть изменён программистом (например, это класс из стандартной или другой библиотеки), или если необходимо преобразование в или из не классового типа, допустим только один из вариантов. В других случаях предпочитают использование конструктора преобразования, поскольку

передача его параметра по ссылке может быть эффективнее возврата значения из функции преобразования.

Конструкторы и функции преобразования задают ***пользовательские преобразование типов (user-defined conversion)***. В приведении типов, явном или неявном, если исходный или результирующий тип является классовым, последовательность преобразований в общем случае состоит из стандартной последовательности преобразований, пользовательского преобразования и ещё одной стандартной последовательности преобразований. Больше чем одно пользовательское преобразование в одной операции приведения типов задействовано быть не может.

```
struct X
{
    operator int*() const;
};

// Пустая стандартная последовательность X->X,
// пользовательское преобразование X->int*
// за счёт функции преобразования,
// стандартная последовательность int*->const int*.
const int* p = X();

struct A
{
};

struct B
{
    B(const A&);
};

struct C
{
    C(const B&);
};

// ОШИБКА: для приведения типа A к C нужно два
//         пользовательских преобразования.
C c1{A()};

// Первое преобразование в A->B задано явно,
// затем срабатывает неявное B->C.
C c2{static_cast<B>(A)};
```

Преобразование выбирается с помощью механизма выбора перегрузок и должно быть, как обычно, однозначным:

```
struct B;

struct A
{
    A(const B&);
};

struct B
{
    operator A() const;
};

// ОШИБКА: есть два пути с пользовательскими преобразованиями,
//         через A::A(const B&) и через B::operator A() const.
A a = B();
```

Как было показано выше, пользовательские преобразования могут использоваться как в явных, так и неявных приведениях типов. Это допустимо в случаях, когда такое преобразование происходит между семантически эквивалентными типами без потери данных, но может быть во многих случаях не желательным:

```
// Класс, соответствующий файлу на диске.
class stream
{
public:
    // Конструктор, создающий объект, связанный с указанным файлом.
    stream(const char file_name[]);

    // Другие члены.
};

void f(const stream& file,const char data[]);

void g()
{
    f("string","/home/bob/a.txt");
}
```

Приведённый пример корректен с точки зрения языка C++ и успешно транслируется, хотя очевидно, что программист перепутал местами порядок аргументов, указывая имя файла и строковые данные. Это произошло, потому что допускается неявное преобразование из `const char*` в `stream` за счёт конструктора преобразования, хотя по смыслу указатель на последовательность символов и поток ввода-вывода — разные вещи.

Конструкторы и функции преобразования могут содержать в описании спецификатор `explicit`, который разрешает их использование только в процессе прямой инициализации, включая явную операцию приведения типов. Такие конструкторы не считаются конструкторами преобразования. Это единственный случай в языке, где проявляется разница между прямой инициализацией и инициализацией копированием: первая может использовать любые пользовательские преобразования, а вторая —- только не `explicit`. Напомним, что передача аргументов функции является инициализацией копированием. С помощью этого средства может решить описанную проблему:

```
class stream
{
public:
    // Явный конструктор не может использоваться в большинстве
    // неявных преобразований.
    explicit stream(const char file_name[]);

    // Другие члены.
};

void f(const stream& file,const char data[]);

void g()
{
    // ОШИБКА: нет преобразования из const char[] в stream для
    //         первого аргумента в единственной перегрузке f.
    f("string","/home/bob/a.txt");

    // Программист вынужден обратить внимание и явно сконструировать
    // объект stream там, где он думает это надо сделать.
    // (Прямая инициализация, так что explicit конструктор допустим.)
    f("string",stream("/home/bob/a.txt"));
    // всё равно ОШИБКА - теперь оба аргумента не подходящего типа.

    // Теперь программист уточняет описание функции и, наконец,
    // понимает, что перепутал аргументы местами.

    // Теперь всё транслируется и работает верно:
    f(stream("/home/bob/a.txt"),"string");
}
```

Хотя может показаться, что явные пользовательские преобразования требуют дополнительного кода, эта цена оправдывается логической корректностью и автоматическим нахождением нежелательных преобразований типов компилятором. По этой причине почти все конструкторы, принимающие один аргумент, описываются как `explicit`.

В качестве другого примера рассмотрим класс рационального числа:

```
class rational
{
    int num,den;
public:
    rational(int num,int den = 1)
    : num(num),
    den(den)
    {
    }

    explicit operator double() const
    {
        return static_cast<double>(num)/den;
    }
};
```

В данном случае преобразование `int` в `rational` можно оставить неявным, поскольку оно никогда не ведёт к потере данных: целые числа — подмножество рациональных. А вот обратное преобразование сделано явным, поскольку в большинстве случаев результат операции деления не представим в точности в типе `double`, но в ситуации, когда эта гарантия есть или не требуется, это преобразование можно выполнить явным приведением типов.

В общем случае, чтобы избежать проблем с неоднозначностью выбора перегрузок, неявным оставляют максимум одно направление преобразования между двумя типами, с примерами в поддержку этого совета мы познакомимся непосредственно в следующем разделе.

Операция приведения к `bool` является одним из ярких примеров, где необходимо сделать функцию преобразования явной во избежание несуразностей:

```
    // Класс с ошибочно неявным преобразованиеми к bool.

    struct A
    {
        operator bool() const;
    };

    void f()
    {
        A a;
        // Компилируется за счёт неявного приведения
        // a к bool, хотя над объектами типа A ничего
        // похожего на арифметику делать нельзя.
        a+3;
    }
```

Помимо прямой инициализации и явного приведения типов имеются несколько контекстов, в которых также допускается использование `explicit` преобразований к типу `bool`. В них говорят, что соответствующее выражение должно быть ***контекстно преобразуемо (contextually convertible)*** к `bool`. Контролирующие выражения условного оператора и операторов цикла, операнды логических операций и первый операнд тернарной операции являются контекстами, где допустимы даже явные преобразования в `bool` без использования явного синтаксиса такого преобразования. Многие классы по этой причине имеют явную функцию преобразования в `bool`, возвращающую истину, если они находятся в некотором штатном или непустом состоянии. Тип давно используемого нами объекта `std::cin` имеет именно такую функцию, что позволяет в соответствующих контекстах трактовать его как логическое значение.

Конструкторы и функции преобразования считаются пользовательскими преобразованиями. Напомним, что при ранжировании перегрузок пользовательское преобразование считается хуже, чем стандартная последовательность преобразований. При сравнении двух неявных последовательностей преобразования, содержащих одно и то же пользовательское преобразование, сравниваются стандартные последовательности преобразования после пользовательских.

```
struct A
{
    operator short() const;
```

Выражение	Член класса	Свободная функция
`@a`	`a.operator@()`	`operator@(a)`
`a=b`	`a.operator=()`	
`a@b`	`a.operator@(b)`	`operator@(a,b)`
`a[b]`	`a.operator[](b)`	
`a->`	`a.operator->()`	
`a@`	`a.operator@(0)`	`operator@(a,0)`
`a(...)`	`a.operator()(...)`	

Таблица 9.2: Вид функций-перегрузок операций

```
4  } a;
5
6  int f(float); // 1
7  int f(int);   // 2
8
9  // Обе перегрузки требуют пользовательское
10 // преобразование A::operator short() const,
11 // но преобразование результата по стандартной
12 // последовательности преобразования для первого
13 // случая short->float хуже, чем short->int для второго.
14 int i = f(a); // 2
```

9.6.7. Перегрузка операций

Ещё одним шагом в сторону придания классовым типам свойств, характерных для других, является возможность задания семантики применения к классовым типам операций языка C++. Читатель уже мог заметить, что между операциями и функциями есть много общего: и те и другие возвращают результат, операнды и аргументы являются входными данными, по сути, единственным различием остаётся синтаксис.

Эта схожесть не случайна. Если хотя бы одним из операндов операции является значение классового или перечисляемого типа, это использование операции рассматривается как вызов функции с именем, состоящим из ключевого слова `operator` и знака операции, после чего осуществляется поиск перегрузок такой функции в виде свободной функции, а если левый операнд имеет классовый тип, то и среди членов этого класса. Кроме этого, рассматриваются встроенные в язык операции, например, сложения двух значений типа `int`.

В языке C++ эти функции называют ***перегрузками операций (operator overload)***. Они допускаются для всех операций, кроме прямой выборки `.`, пока не известной нам операции выборки по адресу члена `.*`, операции разрешения области видимости `::` и условной операции `?:`. Операции взятия адреса `&` и «запятая» `,` имеют определения и для классовых объектов без их явного определения в стандартном смысле, но эти стандартные реализации можно заменить, дав свои. Операции, обозначаемые `+`, `-` и `*`, могут быть перегружены как унарные и как бинарные. Формы поиска перегрузок для операций приведены в таблице 9.2, где `@` — знак операции, а `a` и `b` — операнды.

Из приведённой таблицы видно, что не все операции могут быть перегружены в виде свободных функций. В то же время, если для требуемой перегрузки левый операнд не имеет классового типа, или этот класс не подлежит изменению, необходимо использовать перегрузку в виде свободной функции. При перегрузке в виде функций-членов неявный объект параметр привязывается к левому или единственному операнду, а остальные, если есть, передаются в качестве аргументов.

Операция косвенного доступа при перегрузке считается унарной — при вызове её на объекте, имеющем описание её перегрузки, она вызывается без дополнительных аргументов, после чего её возвращаемое значение вновь подставляется на место левого операнда `->`, это свойство пригодится нам позднее. Постфиксные унарные операции инкремента и декремента имеют дополнительный аргумент 0, чтобы отличить их перегрузки от префиксных версий.

Все вышеперечисленные операции имеют фиксированное число аргументов, которое должно соответствовать требуемому. Для операции вызова функции может существовать сколько угодно перегрузок с любым числом параметров (и их аргументов по умолчанию). Эта операция позволяет «вызывать» объект, словно он является функцией — такие объекты называют ***функциональными объектами (functional object)*** или ***функторами (functor)***.

Перегрузки операций позволяют полностью задать семантику операций независимо друг от друга, например, если имеется перегрузка бинарной операции сложения, перегрузка составной операции присваивания со сложением может выполнять произвольные действия, не имеющие отношения к стандартному поведению для фундаментальных типов, если она вообще имеется. Тип возвращаемого значения перегруженной операции также определяется программистом. В то же время нельзя изменить приоритет и ассоциативность операций, число их операндов и синтаксис записи. Эти свойства с одной стороны дают большую гибкость перегрузке операций, а с другой стороны требуют большого труда для поддержания стандартной семантики и дают возможность легко запутаться, если этого не делать.

Приведём пример:

```
#include <iostream>

// Рациональное число.
// Проверки на переполнение, деление на ноль и прочие ошибки для простоты опущены.
class rational
{
    int num_,den_;
public:
    rational(int num,int den = 1)
    : num_(num),
    den_(den)
    {
    }

    explicit operator double() const
    {
        return static_cast<double>(num_)/den_;
    }

    int num() const
    {
        return num_;
    }

    int den() const
    {
        return den_;
    }

    // Перегрузка операции префиксного инкремента
    // как члена класса.
    rational& operator++()
    {
        num_ += den_;
        return *this;
    }

    // Постфиксный инкремент вызывается с лишним
    // аргументом 0, чтобы отличить перегрузку
    // от префиксной формы.
    rational operator++(int)
    {
        // Эта стандартная реализация
        // постфиксного инкремента через префиксный
        // годится в большинстве случаев.
        rational t(*this);
        ++*this;
        return t;
    }
};

// Перегрузка операции умножения как свободной функции.
rational operator*(const rational& a,const rational& b)
{
    return {a.num()*b.num(),a.den()*b.den()};
```

```
56  }
57
58  void f()
59  {
60      rational a{1,2},b{3,5};
61      // Вызов operator*(frac,frac).
62      rational c = a*b;
63      std::cout << "a*b = " << c.num() << '/' << c.den() << '\n';
64
65      // Преобразовывает 3 из int в rational конструктором
66      // преобразования и вызывает operator*(const rational&,const rational&).
67      // Если бы преобразование rational в double было неявным,
68      // то вызов был бы ошибочно неоднозначным - был бы ещё вариант
69      // привести a и 3 к double и воспользоваться встроенным
70      // operator*(double,double).
71      rational d = a*3;
72
73      // Перегруженные операции можно вызвать и полным
74      // их именем с использованием операции вызова функции,
75      // чего делать со встроенными в язык нельзя.
76      c = operator*(a,b);
77  }
```

Читатель, вероятно, уже догадался, что операции `>>` и `<<` имеют перегрузки для классов потоков ввода-вывода, которыми мы пользовались, начиная с нашей первой программы.

9.6.8. Аргументо-зависимый поиск имён

Если в качестве операнда операции вызова функции, идентифицирующего вызываемую функцию, указано неквалифицированное имя, то его поиск осуществляется по особым правилам, называемым ***аргументо-зависимым поиском имён (Argument-Dependent Lookup, ADL)***. Его цель — расширить список областей видимости, где может быть найдено имя вызываемой функции, исходя из типов её аргументов.

Вначале выполняется обычный не квалифицированный поиск имени. Если среди его кандидатов есть функции-члены, описания функций с блочной областью видимости, не являющиеся описаниями `using` или сущности, не являющиеся функциями, аргументо-зависимый поиск не применяется. Иначе формируется список ассоциированных пространств имён и классов, исходя из типов аргументов:

- С аргументом фундаментального типа ничего не связано.
- С аргументом классового типа связан его класс, классы, в которые он вложен (если вложен), и ближайшее окружающее пространство имён.
- С аргументом типа категории «указатель» или «массив» связано то же, что и с их базовым типом.
- С аргументом типа категории «функция» связано то же, что и с типами её параметров и возвращаемого значения. Если аргумент — имя перегруженной функции, то берётся объединение связанных классов и пространств имён от всех перегрузок.

Объединив все связанные классы и пространства имён, в этом списке областей описания ищется необходимое имя. Поиск осуществляется строго в этих областях видимости, при этом игнорируются директивы `using` и описания, не являющиеся функциями. Все найденные описания добавляются к списку кандидатов, найденному обычным не квалифицированным поиском, после чего начинается обычное разрешение перегрузок.

Смыслом существования аргументо-зависимого поиска является то, что функции, обрабатывающие параметры классового типа могут логически входить в его интерфейс, даже не являясь его членами. Приведём пример:

```
1  #include <cmath>
2
3  namespace my_math
4  {
5      class rational
6      {
```

```
7           // Смотри предыдущий пример.
8       };
9
10      // Функция возвращает число бит, достаточное для хранения числителя и знаменателя др
11      // Это свободная функция, по смыслу входящая в интерфейс класса rational.
12      std::size_t weight(const rational& f)
13      {
14          int a = std::abs(f.num());
15          int b = std::abs(f.den());
16          int c = a>b?a:b;
17          return static_cast<std::size_t>(std::ceil(
18          std::log2(static_cast<double>(c))));
19      }
20  }
21
22  void f()
23  {
24      my_math::rational f{11,375};
25
26      // ADL позволяет найти weight в пространсте имён my_math,
27      // так как оно связано с типом аргумента f, без явной
28      // квалификации.
29      std::size_t w = weight(f);
30
31      // Чтобы отключить ADL принудительно, достаточно взять имя
32      // в скобки - суть не меняется, но первый операнд вызова
33      // функции уже не просто неквалифицированное имя.
34      // ОШИБКА: weight не найдено.
35      std::size_t w2 = (weight)(f);
36  }
```

Важность ADL в первую очередь в том, что оно является средством, обеспечивающим функционирование перегруженных в виде свободных функций операций, лежащих в пространствах имён не просматриваемых обычным поиском в точке использования:

```
1   namespace my_math
2   {
3       class rational
4       {
5           // Смотри предыдущий пример.
6       };
7
8       // Перегрузка операции сложения
9       // в виде свободной функции.
10      rational operator+(const rational&,const rational&);
11  }
12
13  void f()
14  {
15      my_math::rational a{1,2},b{2,3};
16
17      // Требумая перегрузка операции сложения
18      // описана в пространстве имён, который
19      // неквалифицированным поиском отсюда
20      // не просматривается. Если переписать вызов
21      // в форме вызова функции operator+(a,b),
22      // то можно будет указать квалификацию имени,
23      // но теряются все синтаксические преимущества
24      // от перегрузки операции. В синтаксисе записи
25      // операций квалификация не предусмотрена, и
26      // этот пример работает только за счёт ADL.
27      my_math c = a+b;
28  }
```

9.6.9. Друзья классов

Нами были рассмотрены случаи, когда функции, не являющиеся членами класса, логически входят в его интерфейс. В таком случае может оказаться, что им также требуется доступ к закрытым членам класса. Нарушить стандартные правила защиты членов класса позволяют описания ***друзей (friend)*** класса.

Описание функции или класса со спецификатором `friend` в любой секции классового типа объявляет его другом класса, для которого доступны члены с любым уровнем защиты. Эти описания не являются описаниями членов самого класса, а лишь именуют внешние сущности, имеющие особые привилегии при доступе к членам класса. Если другом является функция, этот доступ можно осуществлять в её теле. При указании дружественной функции указывается полное описание, т.е. одна конкретная перегрузка. Если это первое описание функции, оно получает внешнюю связанность, иначе сохраняется предыдущая, явные спецификаторы связанности на дружественных описаниях недопустимы. Друзьями могут быть функции-члены другого класса. Если другом является класс, доступ предоставляется в определении самого класса и всех его членов, включая функции-члены и их тела. Дружба является односторонней и не транзитивной.

Приведём примеры:

```
1  class A
2  {
3      // Функция f без параметров - друг класса A.
4      // Поскольку она не была описана ранее, это описание свободной
5      // функции в глобальном пространстве имён с внешней связанностью.
6      friend void f();
7
8      // Закрытый член.
9      int x;
10 };
11
12 void f()
13 {
14     A a;
15
16     // Поскольку f - друг a,
17     // доступ к закрытом члену разрешён.
18     a.x = 5;
19 }
20
21 class B
22 {
23     void f();
24 };
25
26 class C
27 {
28     // Для указания в качестве друга
29     // функции-члена другого класса
30     // этот класс должен быть определён к
31     // в точке описания дружбы, чтобы
32     // квалифицированное имя было найдено.
33     friend void B::f();
34
35     int x;
36 };
37
38 void B::f()
39 {
40     C c;
41
42     // B::f() - друг C,
43     // доступ разрешён.
44     c.x = 5;
45 }
46
47 // Неполное описание класса D.
```

```
48  class D;
49
50  class E
51  {
52      // D - имя дружественного класса.
53      // Оно уже видно как имя класса, ключ в описании друга не обязателен.
54      friend D;
55
56      // Описание класса F другом, но поскольку пока
57      // сам он не описан, требуется детальное имя.
58      friend class F;
59
60      using my_int = int;
61      int x;
62  };
63
64  class D
65  {
66      // D - друг E и может использовать
67      // закрытое описание my_int.
68      E::my_int y;
69  };
70
71  class F
72  {
73      void f()
74      {
75          E e;
76
77          // f - член F, а он - друг E,
78          // поэтому F::f имеет доступ
79          // к закрытым описаниям в E.
80          e.x = 5;
81      }
82  };
83
84  class G
85  {
86      void f();
87      public:
88      // Секция уровня защиты для
89      // описания дружественной сущности
90      // значения не имеет.
91      friend class H;
92  };
93
94  class H
95  {
96      friend class I;
97
98      int x;
99  };
100
101 void G::f()
102 {
103     H h;
104
105     // ОШИБКА: то, что H - друг G
106     //         не означает автоматически,
107     //         что G - друг H.
108     h.x = 5;
109 }
110
111 class I
112 {
113     void f()
```

```
114     {
115         // ОШИБКА: то, что H - друг G,
116         //         а I - друг H не означает
117         //         автоматически, что I - друг G.
118         G().f();
119     }
120 };
```

Хотя дружба не является автоматически коммутативной или транзитивной, этого можно добиться в каждом конкретном случае соответствующими определениями.

> При указании дружественных классов квалификаторы типов игнорируются. Также допустимы и игнорируются указания в качестве дружественных типов, не являющихся классами. Кроме этого, описания дружественных функций в не локальных классах могут являться определениями, которые трактуются как определения встраиваемых функций-друзей класса, являющихся членами окружающей определение класса области видимости. Все эти странные, на первый взгляд, особенности полезны при использовании обобщённой парадигмы программирования C++.
>
> Описание дружественной сущности с неквалифицированным именем в локальном классе соотносится только к описаниями в областях видимости не шире, чем ближайшее окружающая область видимости, не являющаяся классом. Если это функция, и она не найдена, программа не согласована (ей негде больше быть определённый в случае локального класса). Если это класс, и он не найден, он соответствует классу в ближайшей окружающей области видимости, не являющейся классом, но не является его описанием, даже неполным.

9.6.10. Специальные функции-члены класса

Шесть конкретных функций-членов класса выполняют специальные функции, и называются ***специальными функциями-членами класса (special member functions)***. Это ***конструктор по умолчанию (default constructor)***, ***конструктор копирования (copy constructor)***, ***конструктор перемещения (move constructor)*** ***операция присваивания копированием (copy assignment)***, ***операция присваивания перемещением (move assignment)*** и ***деструктор (destructor)***. Их объединяет в эту группу то, что они в некоторых ситуациях могут описываться компилятором неявно в момент окончания определения класса, если программист сам не описал их. Когда программа требует вызова одной из таких неявно описанных функций, компилятор автоматически генерирует её определение в стандартной форме. Большинство из специальных функций-членов класса предназначены для управления ресурсами, которыми владеет объект класса. Эти специальные функции часто вызываются в контекстах, не содержащих явных операций вызова функций, и тем самым автоматизируют управление ресурсами. Необходимость в этом должна быть очевидна читателю после работы с динамической памятью — пока она требовала ручного освобождения, и уследить за этим во всех ветвях программы часто было непросто. С другой стороны, эти автоматически происходящие в программе действия делают язык менее прозрачным, чем C, где практически за каждым фрагментом исходного кода стоит строго определённый и предсказуемый эффект в виде машинных инструкций. Усложнение чтения программы в данном случае есть плата за повышение уровня абстракции и автоматизации, и программист, желающий воспользоваться эффективностью C++, должен чётко представлять, где происходят эти вызовы, чтобы строить верные суждения о корректности и эффективности своих программ.

Функции, описанные программистом явно, называются ***описанными пользователем (user-declared)***. Для таких специальных функций неявные описания созданы не будут. Могут возникать ситуации, когда компилятор генерирует неявное определение специальной функции из-за отсутствия версии, описанной пользователем, в ситуации, когда оно не треюбуется или некорректно. И напротив, компилятор может в соответствии с другими правилами не пытаться дать неявное описание со стандартной формой определения, которое требуется программисту. Чтобы исправить эти ситуации, применяют явные определения ***по умолчанию (defaulted)*** и ***удалённые (deleted)*** определения. Они задаются с использованием синтаксиса `= default;` или `= delete;` вместо тела функции.

Определение по умолчанию заставляет компилятор попытаться создать определение соответствующей функции в стандартной форме. Удалённое определение, являясь описанным пользователем, предотвращает неявное описание, но настоящим определением не является — выбор соответствующей функции как результата разрешения перегрузок ведёт к ошибке компиляции.

Таким образом, стандартная форма определения специальных функций может быть создана в результате неявного описания и определения или явного определения по умолчанию. Неявное описание будет являться встроенной функцией с открытым уровнем доступа, для явного определения по умолчанию встраиваемость и уровень доступа определяются по обычным правилам. За счёт этого явное определение по умолчанию может применяться для изменения этих атрибутов, даже если неявное было бы создано и неявно. Если попытка генерации стандартной формы приводит к некорректному коду, явное или неявное определение по умолчанию превращаются в удалённые. Если определение специальной функции находится вне определения класса отдельно от её описания, оно может быть дано явно по умолчанию, но если такое определение по факту окажется удалённым, программа некорректна. Определение явно удалённой функции должно быть первым её описанием, т.е. для члена класса должно быть дано непосредственно в нём. Во всех остальных случаях определения, когда оно не генерируется компилятором в стандартной форме, его называют ***предоставленным пользователем (user-provided)***.

Синтаксис явного определения по умолчанию допустим только на специальных функциях-членах классов, поскольку ни у каких других функций стандартной формы, предписываемой стандартом языка, нет. Удалённое же определение может использоваться на любых функциях, в том числе даже не на членах класса. Иногда это применяют в следующей ситуации:

```
1  // Проверка на нечётность.
2  bool is_odd(int x)
3  {
4      return x&1;
5  }
6
7  // Добавить во множество перегрузок
8  // "ловушку" для вещественных чисел,
9  // которая для них лучше, чем основная перегрузка.
10 bool is_odd(long double) = delete;
11
12 void f()
13 {
14     // Проверять целое на чётность осмысленно.
15     bool r1 = is_odd(3);
16
17     // Проверять вещественное число на чётность смысла нет,
18     // но без перегрузки с удалённым определением следующий
19     // вызов успешно скомпилировался бы из-за наличия неявного
20     // преобразования double->int.
21     // С нашим вторым определением получаем желаемую ОШИБКУ:
22     // использование удалённой функции.
23     bool r2 = is_odd(3.45);
24 }
```

Все рассмотренные выше свойства покажем на примере самой простой специальной функции, единственной, не отвечающей за управление ресурсами.

9.6.11. Конструктор по умолчанию

Неоднократно показанный нами в примерах конструктор, который может быть вызван с пустым списком аргументов, называется ***конструктором по умолчанию (default constructor)*** и является специальной функцией-членом класса. Если в классе нет описанных пользователем конструкторов, компилятор неявно описывает конструктор по умолчанию с пустым списком параметров. Стандартная форма его определения не содержит списка инициализаторов членов и имеет пустое тело. Эта функция специальная, потому что часто для создания объекта без аргументов никаких особых действий не требуется.

```
1  struct A
2  {
3      int x;
4
5      // Нет описанных пользователем конструкторов,
6      // Даётся неявное описание конструктора по умолчанию,
```

```
7       // эквивалентное
8       // public: inline A();
9   };
10
11  // Если этот конструктор потребуется, даётся его определение:
12  // A::A() {}
13
14  // Инициализация по умолчанию не делает ничего.
15  A a;
16
17  struct B
18  {
19      int x;
20
21      B(x) : x(x) {}
22
23      // Конструктор по умолчанию неявно описан не будет,
24      // т.к. есть описанные пользователем конструкторы.
25  };
26
27  struct C
28  {
29      const int x;
30
31      // Описанных пользователем конструкторов нет,
32      // даётся неявное описание конструктора по умолчанию.
33      // public: inline C();
34      // Если этот конструктор потребуется, стандартная форма
35      // C() {}
36      // окажется некорректной, т.к. неизменяемый подобъект x
37      // не может быть инициализирован по умолчанию, как результат
38      // определение будет дано в удалённом виде:
39      // C() = delete;
40  };
41
42  // ОШИБКА: выбранный для инициализация объекта конструктор удалён.
43  C c;
44
45  struct D
46  {
47      int& x;
48
49      D() = default;
50      // Явно запрошенная стандартная форма определения
51      // конструктора по умолчанию. Описана пользователем,
52      // но не предоставлена им.
53      // Поскольку стандартная форма некорректна из-за невозможности
54      // инициализировать ссылку-член по умолчанию, это то же самое, что
55      // D() = delete;
56  };
57
58  // Если бы определение конструктора по умолчанию класса D выше
59  // было дано вне класса в той же форме
60  // D::D() = default;
61  // то оно бы оказлось ошибочным, т.к. по факту было бы удалённым определением,
62  // которое для этой функции не является первым.
63
64  // Класс, имеющий имя и получающий
65  // уникальный номер по счётчику.
66  class counted
67  {
68  private:
69      // Счётчик объектов этого класса.
70      static size_t counter;
71  public:
72      // Номер объекта, назначаемый по
```

```
73      // счётчику. Открытый, т.к. всё равно не изменяем.
74      const n = counter++;
75
76      // Имя объекта, нулевой указатель,
77      // если не указано обратного во
78      // избежание мусора. Тоже без
79      // функций доступа из-за неизменяемости.
80      const char * const name_ = nullptr;
81
82      // Конструктор по заданному имени,
83      // номер назначается в инициализаторе выше.
84      counted(const char* name_)
85          : name(name_)
86      {
87      }
88
89      // Инициализаторы в описаниях членов класса
90      // создают приемлемую для нас инициализацию
91      // объекта по умолчанию, но соответствующий
92      // конструктор неявно компилятором не будет
93      // создан, т.к. имеются другие, описанные
94      // программистом. Затребуем явно форму по умолчанию.
95      counted() = default;
96  };
97
98  std::size_t counted::counter = 0;
```

9.6.12. Автоматическое освобождение ресурсов

Особый класс представляют объекты, владеющие ресурсами.

В примере динамического массива, который мы рассматривали ранее, за освобождение динамической памяти, используемой структурой данных, отвечала функция `ia_destroy`. Основной проблемой с данной функцией являлось то, что ответственность за её вызов лежала на пользователе, который мог легко забыть вызвать её в одном из путей выхода из сложного алгоритма. Чем с большим числом ресурсов одновременно работает программа, и чем сложнее её структура, тем проще допустить ошибку при ручном контроле за владением.

C++ содержит средство, позволяющее для классовых типов автоматически выполнять требуемые действия по окончании времени жизни объекта. Его использование для освобождения ресурсов в паре с их получением в конструкторе класса позволяет строго привязать существование ресурса к времени жизни владеющего им объекта. Главным улучшением является то, что пользователи таких объектов освобождаются от необходимости следить за освобождением соответствующего ресурса, код становится проще, а освобождение гарантировано выполняется в точности один раз и именно тогда, когда ресурс более не нужен. Это преимущество является одним из фундаментальных улучшений, кардинально влияя на структуру программы на языке C++, по сравнению с C — код освобождения ресурса пишется один раз создателем класса, им владеющим, а при использовании объекта никаких дополнительных действий не требуется, включая самого знания о существовании требующего освобождения ресурса, что благотворно сказывается на инкапсуляции. Идиома, соответствующая привязке времени жизни некоторого внешнего ресурса к времени жизни объекта называется ***«Получение ресурса есть инициализация» (RAII, Resource Acquisition Is Initialization)***. Она имеет и другие преимущества, с которыми мы познакомимся позднее.

Средство, о котором шла речь выше, называется деструктором. ***Деструктор (destructor)*** — специальная функция-член класса, автоматически вызываемая сразу перед окончанием времени жизни объекта классового типа. Для объектов разных времён хранения это означает следующее:

- Для объектов со статическим временем хранения деструктор вызывается сразу перед завершением работы программы (уже после возврата управления из `main`).
- Для объектов с автоматическим временем хранения деструктор вызывается сразу перед выходом из соответствующего блока. Выход может осуществляться как последовательным выполнением блока до конца, так и любыми операторами перехода. Это основное свойство, которое упрощает работу с ресурсами — не требуется нескольких мест освобождения ресурса на всём множестве путей выхода из блока.

- Для временных объектов деструктор вызывается сразу перед окончанием времени их жизни (сразу после вычисления значения полного выражения, в котором они созданы, при окончании времени жизни ссылки, к который они привязаны и т.д.).
- Для объектов с динамическим временем хранения деструктор вызывается операцией `delete` перед освобождением памяти. Необходимость знать число элементов при выделении массива и вызвать для каждого из них деструктор является причиной существования отдельной векторной версии операции `delete` и объяснением того, почему векторная операция `new` выделяет дополнительную память сверх требуемой для хранения массива — в них запоминается число объектов, для которых необходимо вызвать деструкторы.

Деструктор, также как и конструктор, в описании не имеет возвращаемого значения (можно использовать `return` без выражения в теле) и квалификаторов неявного объекта параметра (квалификаторы самого объекта внутри деструктора силы не имеют). Вместо его имени указывается токен `~` и имя содержащего класса. Деструктор не имеет параметров и не может быть перегружен. Взятие адреса деструктора, как и конструкторов, невозможно. Сразу после выполнения тела деструктора автоматически вызываются деструкторы всех не статических членов данных класса в порядке, обратном порядку вызова их конструкторов, т.е. описаний в определении класса.

Если класс не содержит описанного пользователем деструктора, компилятор даст ему неявное описание. Стандартная форма определения деструктора — пустое тело. Разумеется, даже в такой форме он после его выполнения уничтожает подобъекты класса. Покажем использование деструктора на простом примере класса, владеющего массивом целых чисел задаваемого при инициализации размера:

```
1  class fixed_int_array
2  {
3      // Указатель на данные, владеющий.
4      int* ptr;
5      // Число выделенных элементов.
6      std::size_t size_;
7  public:
8      // Конструктор получает ресурс --- динамическую память.
9      fixed_int_array(std::size_t size)
10         : ptr(new int[size]{}),
11           size_(size)
12     {}
13
14     // Деструктор освобождает данные, удовлетворяя требования по владению.
15     ~fixed_int_array()
16     {
17         delete[] ptr;
18     }
19
20     // Функции доступа к данным класса.
21
22     std::size_t size() const
23     {
24         return size_;
25     }
26
27     int& elem(std::size_t i)
28     {
29         return ptr[i];
30     }
31
32     int elem(std::size_t i) const
33     {
34         return ptr[i];
35     }
36 };
37
38 // Пара вспомогательных функций для демонстрации
39 // автоматического вызова деструктора.
40 bool failing_function();
41 bool other_failing_function();
```

```
42
43  void f()
44  {
45      for(int i=0;i<10;++i)
46      {
47          // Создание объекта с выделением
48          // ресурса - динамической памяти.
49          fixed_int_array fia(10);
50
51          // ...
52
53          if(!failing_function())
54              // Выход из блока оператором return,
55              // деструктор fia вызывается автоматически.
56              return;
57
58          // ...
59
60          if(!other_failing_function())
61              // Выход из блока оператором return,
62              // деструктор fia вызывается автоматически.
63              continue;
64      } // Если управление дошло до конца блока,
65        // деструктор fia вызывается здесь.
66
67      // Эффективнее было бы определить fia вне цикла,
68      // чтобы избежать его пересоздания на каждой
69      // итерации. Это просто пример того, что любой
70      // штатный выход из блока вызывает деструктор
71      // автоматического объекта.
72  }
```

Тот факт, что деструкторы не статических членов класса вызываются автоматически после выполнения его собственного, обеспечивает возможность лёгкого использования объектов даже с нетривиальными деструкторами в качестве членов класса без дополнительных усилий:

```
75  // Продолжение предыдущего примера
76
77  struct pair_of_arrays
78  {
79      // fixed_int_array не имеет конструктора по умолчанию и
80      // требует явных инициализаторов.
81      fixed_int_array a1{10},a2{20};
82
83      // Неявно определённый конструктор по умолчанию
84      // инициализирует оба члена данными в их описаниях
85      // аргументами.
86
87      // Неявно определённый деструктор
88      // автоматически вызывает деструкторы
89      // a1 и a2 при уничтожении объекта класса pair_of_arrays
90      // после выполнения своего пустого тела.
91  }
```

9.6.13. Копирование объектов классовых типов

Придание объекту состояния, семантически эквивалентного состоянию другого объекта, называется ***копированием (copy)***. Например, для арифметических типов оно соответствует копированию представления объекта в памяти. Для классовых типов смысл их «значения» может быть произвольно сложным. В простейших случаях, когда класс — агрегат, может быть достаточно почленного копирования значений подобъектов. Для объектов, владеющих ресурсами, копирование значения обычно означает создание новых ресурсов с эквивалентными значениями.

Конструктор, принимающий значение собственного типа не допустим, но можно принять ссылку на значение типа того же класса. Такой конструктор, независимо от квалификаторов

на базовом типе леводопустимой ссылки и наличия у него дополнительных параметров с аргументами по умолчанию, называется конструктором копирования. Поскольку копирование по смыслу не затрагивает исходный объект, квалификатор на базовом типе параметра конструктора копирования обычно `const`, это же позволяет копировать значения объектов любых категорий.

Дадим определение конструктора копирования для рассмотренного выше класса `fixed_int_array`:

```
1  class fixed_int_array
2  {
3      int* ptr;
4      std::size_t size_;
5  public:
6      // Конструктор копирования.
7      fixed_int_array(const fixed_int_array& other)
8      // Выделить память под массив того же размера,
9      // что и в другом объекте.
10         : ptr(new int[other.size_]),
11     // Задать размер этого объекта равным размеру
12     // другого объекта.
13           size_(other.size_)
14     {
15         // Скопировать сами данные.
16         for(std::size_t i=0;i<size_;++i)
17             ptr[i] = other.ptr[i];
18     }
19
20     // Другие члены класса
21 };
22
23 void f()
24 {
25     // Создать объект.
26     fixed_int_array fia1(5);
27
28     // Заполнить его данными.
29     for(size_t i=0i<5;++i)
30         fia1.elem(i) = i;
31
32     // Создать две его копии прямой инициализацией
33     // и инициализацией копированием. В обоих
34     // случаях вызывается конструктор копирования.
35     fixed_int_array fia2(fia1),
36                     fia3 = fia1;
37 }
```

Наиболее яркими примерами неявного вызова конструктора копирования является копирование аргумента классового типа в параметр функции и выражения в операторе `return` в возвращаемое значение функции.

Для придания объекту значения, семантически эквивалентному значению другого объекта в процессе его существования используется операция присваивания копированием. Это перегрузка операции присваивания с параметром, соответствующим правому операнду, того же типа, как и у конструктора копирования. Для сохранения обычной семантики тип её возвращаемого значения обычно леводопустимая ссылка на сам класс, и возвращаемое значение соответствует неявному параметру-объекту, чтобы над объектами этого класса можно было осуществлять присваивание по цепочке, как и с фундаментальными типами.

Эта функция может выполнять те же действия, что и конструктор копирования, но без использования списка инициализаторов, который допускается только в конструкторах. Кроме этого, перед копированием в объект состояния из другого объекта, она должна полностью уничтожить старое состояние или только его часть, воспользовавшись оставшимся для хранения нового, чтобы не освобождать ресурсы, которые тут же будут выделены заново. Таким образом, она семантически эквивалентна связке деструктора и конструктора копирования, но вместо полного уничтожения старого значения и создания нового как копии значения другого объекта с нуля, может воспользоваться имеющимися ресурсами. Для эффективности обычно проверяют, не происходит ли присваивание объекта самому себе, и пропускают всю логику

копирования в этом случае — часто это не только увеличивает производительность, но и необходимо для корректности программы.

```
class fixed_int_array
{
    int* ptr;
    std::size_t size_;
public:
    // Операция присваивания копированием.
    fixed_int_array& operator=(const fixed_int_array& other)
    {
        // Копируем, только если это не присваивание самому себе.
        if(this!=&other)
        {
            // Заменить буфер новым нужного размера,
            // если требуется.
            if(size_!=other.size_){
                delete[] ptr;
                ptr = new int[other.size_];
                size_ = other.size_;
            }
            for(std::size_t i=0;i<size_;++i)
                ptr[i] = other.ptr[i];
        }
        // Вернуть ссылку на неявный параметр-объект.
        return *this;
    }
};

void f()
{
    // Создать объект.
    fixed_int_array fia1(5);

    // Заполнить его данными.
    for(size_t i=0i<5;++i)
        fia1.elem(i) = i;

    // Создать ещё объект.
    fixed_int_array fia2{3};

    // Скопировать содержимое одного в другой,
    // уже существующий, вызывается операция
    // присваивания копированием.
    fia2 = fia1;
}
```

Рассмотрим следующий пример:

```
class C
{
    fixed_int_array fia{0};
public:
    const fixed_int_array& get_fia() const
    {
        return fia;
    }

    void set_fia(const fixed_int_array& fia)
    {
        this->fia = fia;
    }
};
```

В данном случае рассмотренный выше класс `fixed_int_array` используется в качестве закрытого члена класса с соответствующими функциями доступа. Поскольку этот тип не тривиально копировать, он принимается и возвращается по ссылке на неизменяемый объект.

Поскольку конструктор копирования и операция присваивания копированием — специальные функции, у них есть стандартные формы определения по умолчанию. Описания этих стандартных функций даются неявно всегда, за исключением случая, когда в классе есть описанная пользователем та же специальная функция. Стандартная форма определения конструктора копирования инициализирует все подобъекты создаваемого объекта значениями соответствующих подобъектов исходного объекта. Если подобъект — массив, то все его элементы тоже инициализируются из соответствующих элементов массива-подобъекта исходного объекта. Тело этой формы пусто. Стандартная форма определения операции присваивания копированием аналогичным образом присваивает значения подобъектов в своём теле, и возвращает неявный параметр объект как леводопустимое значение. Таким образом, стандартные формы функций копирования осуществляют почленное копирование значений. Они принимают исходный объект по леводопустимой ссылке на неизменяемое значение, если в такой форме корректны, иначе — по леводопустимой ссылке без квалификаторов.

Покажем, как можно записать стандартные формы этих специальных функций явно:

```
// A, B и C - имена типов.
using A = /* ... */;
using B = /* ... */;
using C = /* ... */;
struct S
{
    A a;
    B b[2];
    C c;

    S(const S& other)
    // Инициализировать копированием почленно, включая элементы массивов.
        : a(other.a),
          b{other.b[0],other.b[1]},
          c(other.c)
    {}

    S& operator=(const S& other)
    {
        a = other.a;
        b[0] = other.b[0];
        b[1] = other.b[1];
        c = other.c;
        return *this;
    }
};
```

Такая форма позволяет избежать самостоятельного написания функций копирования для простых классов, например, агрегатов, и копировать их как и значения фундаментальных типов.

9.6.14. Семантика переноса и другие оптимизации копирования

Воспользуемся классом из предыдущей главы и попробуем установить значение свойства `fia` в значение из миллиона нулей:

```
void f()
{
    C c;
    c.set_fia(fixed_int_array{1000000});
}
```

В функции `f` демонстрируется передача временного объекта в качестве аргумента setter'а, чтобы показать неэффективность его реализации: временный объект создаётся только для того, чтобы в теле setter'а произошло копирование его данных, которые будут тут же уничтожены по завершении его выполнения. Тем не менее эту ситуацию нельзя исключать, и она требует решения, т.к. часто использование временных объектов является неизбежным или более удобным, кроме того, возможны ситуации, в которых не временный объект заведомо не используется после такого вызова и его копирование также не обосновано.

Описанные ситуации можно охарактеризовать как случаи, в которых не важно, какое значение окажется в объекте, после того, как будет скопировано в другой. ***Перемещением (move)*** называют подвид копирования, которому разрешено оставлять исходный объект в неуточняемом состоянии. Перемещение является оптимизацией копирования и основной причиной появления праводопустимых ссылок в языке, начиня с C++11 — именно праводопустимые ссылки позволяют в перегрузках функций различать категории значения. С точки зрения перемещения, значения, для которых праводопустимые ссылки годны лучше, чем левые, это значения, значения которых разрешено уничтожать. Это временные объекты, которые имеют категорию prvalue, и объекты категории xvalue, являющиеся результатом явного приведения типов к праводопустимым ссылкам, т.е. явно помеченные программистом, как более не нужные. В случае перемещения можно обойтись без выделения новых ресурсов, позаимствовав имеющиеся у исходного объекта, который после перемещения из него своё значение теряет. Как минимум требуется обеспечить на таком объекте корректную работу деструктора и операций присваивания, полностью заменяющих его состояния новым. Данные соглашения носят название ***семантики переноса (move semantics)***.

Иногда вместо неуточняемых документируют, что объекты, из которых было совершено перемещение, получают конкретные, обычно пустые по смыслу значения. В некоторых случаях это может быть удобно, но стоит помнить, что весь смысл семантики переноса — оптимизация копирования, поэтому обычно в перемещение закладывают минимум необходимых действий.

Технически за реализацию семантики переноса отвечают две оставшиеся специальные функции: конструктор перемещения и операция присваивания перемещением. Их отличает от соответствующих функций копирования использование праводопустимых ссылок. В их случае квалификаторов на базовом типе параметра обычно нет, чтобы разрешить изменение привязанного объекта.

Покажем решение поставленной проблемы:

```
1  class fixed_int_array
2  {
3      int* ptr;
4      std::size_t size_;
5  public:
6      // Конструктор перемещения.
7      fixed_int_array(fixed_int_array&& other)
8      // Переносим указатель на данные из
9      // другого объекта в данный.
10         : ptr(other.ptr),
11     // Переносим размер.
12           size_(other.size_)
13     {
14         // Никакого ресурсоёмкого копирования
15         // всех данных нет.
16
17         // Другой объект более не владеет данными,
18         // занулим его указатель, чтобы он не
19         // удалили их в своём деструкторе -
20         // теперь за это отвечает наш объект.
21         // Это передача владения между объектами.
22         other.ptr_ = nullptr;
23
24         // Если требуется конкретное вместо неуточняемого
25         // состояние объекта, из которого произошло
26         // перемещение, занулим и размер, чтобы не нарушать инвариант
27         // "хранимый размер соответствует размеру
28         //  выделенных данных".
29         // other.size_ = 0;
30     }
31
32     // Операция присваивания перемещением.
33     fixed_int_array& operator=(fixed_int_array&& other)
34     {
35         // Действия, аналогичные конструктору перемещения.
36         assert(this!=&other);
37         ptr = other.ptr;
38         size_ = other.size_;
```

```
39          other.ptr = nullptr;
40          // other.size_ = 0;
41          return *this;
42      }
43
44      // Специальные функции-члены для копирования.
45      fixed_int_array(const fixed_int_array&);
46      fixed_int_array& operator=(const fixed_int_array&);
47
48      // Остальные функции-члены.
49  };
50
51  class C
52  {
53      fixed_int_array fia{0};
54  public:
55      const fixed_int_array& get_fia() const
56      {
57          return fia;
58      }
59
60      void set_fia(const fixed_int_array& fia)
61      {
62          this->fia = fia;
63      }
64
65      // Перегрузка setter'а для более не нужных
66      // объектов-аргументов.
67      void set_fia(fixed_int_array&& fia)
68      {
69          // Вызов операции присваивания переносом.
70          // Напомним, что приведение типов необходимо,
71          // т.к. даже праводопустимая, но именованная
72          // ссылка - леводопустимое выражение, хотя
73          // привязана к праводопустимому.
74          this->fia = static_cast<fixed_int_array&&>(fia);
75      }
76  };
77
78  void f()
79  {
80      C c;
81      c.set_fia(fixed_int_array{1000000});
82  }
```

В данном примере копирования миллиона целых чисел в вызове setter'а не производится. Поскольку копирование объектов может быть сколь угодно сложным, эта оптимизация критически важна для многих реализаций.

В некоторых случаях требуется явно указать, что значение не временного объекта больше не нужно и может трактоваться как временное для использования семантики переноса:

```
85  // Продолжение предыдущего примера
86
87  void g()
88  {
89      C c;
90      fixed_int_array fia{10};
91      // ...
92
93      // fia больше не будет использоваться в этой
94      // функции, его можно трактовать как более
95      // не нужный, что требуется сделать явно,
96      // т.к. он леводопустим.
97      c.set_fia(static_cast<fixed_int_array&&>(fia));
98
99      // ...
100 }
```

Стандартные формы функций перемещения аналогичны стандартным формам копирования, но не имеют квалификаторов на базовом типе ссылки-параметра, и все почленные присваивания или инициализации производятся из значений подобъектов, трактуемых как имеющие категорию значения rvalue. Покажем, как такая форма выглядит для конструктора перемещения:

```
1  // A и B - типы.
2  using A = /* ... */;
3  using B = /* ... */;
4
5  struct S
6  {
7      A a;
8      B b;
9
10     S(S&& other)
11         : a(static_cast<A&&>(other.a)),
12           b(static_cast<A&&>(other.b))
13     {}
14 };
```

Функции перемещения будут неявно описаны и определены только, если в классе пользователем не описана ни одна из пяти специальных функций, ответственных за управление ресурсами, включая их самих. Это следует трактовать как «если этот класс имеет что-то не стандартное, имеющее отношение к владению ресурсами, стандартная форма перемещений не годится». Было бы логично расширить это правило на всю пятёрку функций управления ресурсами, но этого сделано не было из-за проблем обратной совместимости со старыми версиями C++. Функции копирования реагируют на наличие функций перемещения следующим образом: если имеется хотя бы одна описанная пользователем функция перемещения, определения функций копирования, полученные в результате неявных описаний, становятся удалёнными. Это позволяет, например, явно удалить перемещение и явно затребовать копирование в стандартной форме. Наличие другой функции копирования или деструктора, описанного пользователем, функцию копирования «не смущает», но такое поведение в текущей версии стандарта объявлено устаревшим. Это лучшее, что может предложить текущий стандарт для совместимости с версией языка, где не существовало праводопустимых ссылок и удалённых определений.

Если класс получил каким-либо образом удалённое определение конструктора или операции присваивания перемещением, оно не участвует в разрешении перегрузок, чтобы не перехватывать случаи, которые могут вместо них быть обработаны функциями копирования, присутствующими в том или ином виде всегда.

Стандартом также определяются случаи, когда формально необходимые копирования объектов могут быть заменены перемещениями или опущены вовсе автоматически. Последний случай допускается, даже если соответствующие конструкторы копирования и деструктор имеют побочные эффекты — пара соответствующих им вызовов всегда считается «ненужной». Несмотря на то, что в случае использования этих оптимизаций соответствующие функции не вызываются, все проверки, связанные с возможностью их вызова всё равно выполняются. Эти оптимизации могут фактически изменять описания функций, использовать нестандартные соглашения о вызовах и осуществлять полное или частичное встраивание для достижения необходимого результата на уровне машинного кода. Перечислим эти возможности автоматической оптимизации:

- При возвращении из функции значения классового типа, соответствующего локальной переменной без квалификатора `volatile`, не являющейся параметром функции, компилятор может избежать её копирования в возвращаемое значение функции, сконструировав его напрямую в нём в её определении. Эта оптимизация известна также под названием ***NRVO (Named Return Value Optimization)***.

```
1  fixed_int_array f()
2  {
3      // Этот объект может быть создан
4      // напрямую в возвращаемом значении
5      // функции,...
6      fixed_int_array fia(10);
7      // ...
```

```
8       // чтобы избежать его копирования в него здесь.
9       return fia;
10  }
```

- Временный объект, не привязанный к ссылке и используемый для инициализации копированием или перемещением другого объекта того же типа с теми же квалификаторами, может быть сконструирован напрямую во втором объекте. Эта оптимизация в случае, когда вторым объектом является возвращаемое значение функции, известна под именем ***RVO (Return Value Optimization)***.

```
1   fixed_int_array f()
2   {
3       // Вместо копирования временного объекта
4       // в возвращаемое значение функции, он
5       // может быть сконструирован в нём напрямую.
6       return fixed_int_array(10);
7   }
8
9   void g()
10  {
11      // Вместо копирования возвращённого f значения
12      // в fia, fia может напрямую использоваться в
13      // качестве возвращаемого значения в вызове f().
14      fixed_int_array fia = f();
15  }
```

- В описанных выше случаях, а также когда выражение оператора `return` именует параметр функции, компилятор пытается сначала трактовать значение как праводопустимое, так что при наличии соответствующих специальных функций-членов в ситуации, когда оптимизатор не произвёл описанные выше ***избегания копирования (copy elision)***, вместо копирования будет выполнено более эффективное перемещение. В примере выше временное значение будет не скопировано, а перемещено в возвращаемое значение функции `f`, а из него — в объект `fia`. Возвращаемое значение в операции `return` не следует явно приводить к xvalue там, где оно трактуется как таковое автоматически, поскольку это отнимет у транслятора выбор между перемещением и избеганием копирования вообще.

Все рассмотренные в данном разделе оптимизации направлены на эффективное использование классов наравне с фундаментальными типами, чтобы минимизировать потери производительности на границах функций. Например, до C++11 вместо возврата из функций классов, копирование которых ресурсоёмко, применяли out-параметры через ссылки, что делало интерфейс функций неестественным и более сложным в использовании.

С учётом их оптимизаций, пару перегрузок функций, которым выгодно различать категорию передаваемого им значения, можно заменить одной чуть менее эффективной:

```
1   class C
2   {
3       fixed_int_array fia{0};
4   public:
5       void set1(const fixed_int_array& fia)
6       {
7           this->fia = fia;
8       }
9
10      void set1(fixed_int_array&& fia)
11      {
12          this->fia = static_cast<fixed_int_array&&>(fia);
13      }
14
15      // Новый вариант: передача по значению, без ссылок (!)
16      void set2(fixed_int_array fia)
17      {
18          this->fia = static_cast<fixed_int_array&&>(fia);
19      }
20  };
21
```

```
22  void f()
23  {
24      C c;
25      fixed_int_array fia{10};
26
27      // Одно копирование в setter'е.
28      // Избежать его нельзя логически, т.к.
29      // исходный объект нельзя изменять по
30      // смыслу функции и, следовательно,
31      // нельзя украсть его ресурс.
32      c.set1(fia);
33
34      // Одно перемещение в setter'е.
35      // Его избежать невозможно.
36      c.set1(fixed_int_array{10});
37
38      // Вариант set1 требует двух перегрузок,
39      // но даёт минимально возможное число операций
40      // копирования или переноса.
41  }
42
43  void g()
44  {
45      C c;
46      fixed_int_array fia{10};
47
48      // Копирование в параметр, перемещение в setter'е,
49      // одно лишнее перемешние по сравнению с оптимальным
50      // вариантом.
51      c.set2(fia);
52
53      // За счёт избегания копирования объект будет,
54      // вероятно, сконструирован напрямую в параметре,
55      // откуда будет перенесён. Если этого не произойдёт,
56      // будет ещё одно перемещение из временного значения
57      // в параметр.
58      c.set2(fixed_int_array{10});
59  }
```

Эта замена особенна полезна в случае, когда таких параметров у функции несколько, чтобы избежать комбинаторного взрыва числа требуемых перегрузок. Отметим, что хотя перемещения могут быть значительно эффективнее копирования, это не полностью бесплатные операции, а в худшем случае — такие же, как и копирование. Тот факт, что столь базовые действия как передача и возврат значений между функциями часто требуют вдумчивого подхода с ручной реализацией многих вариантов, в зависимости от свойств используемых типов, чтобы быть оптимальными или близкими к этому, до сих пор считают большим недостатком C++.

9.6.15. Классификация объектов по свойствам специальных функций-членов

В общем случае класс может реализовывать, удалять или не описывать специальные функции-члены класса в произвольных комбинациях, и это будет влиять на успешность создания неявных определений специальных функций в тех классах, которые содержат объекты исходного в качестве не статическиз членов данных:

```
1  // Неявные описания закомментированы.
2
3  struct strange
4  {
5      strange() = default;
6
7      // Объект со странным конструктором копирования,
8      // который разрешает копирование только из
9      // изменяемых леводопустимых значений.
```

```
10      strange(strange&) {}
11
12      // Для пущей странности явно удалено присваивание перемещением.
13      strange& operator=(strange&&) = delete;
14
15      // Неявного описания конструктора перемещения нет,
16      // так как есть описанный пользователем конструктор копирования.
17
18      // Неявное описание операции присваивания копированием удалено,
19      // поскольку есть описанный пользователем конструктор перемещения.
20      // strange& operator=(const strange&) = delete;
21  };
22
23  struct X
24  {
25      strange s;
26
27      // Реализация конструктора копирования по умолчанию принимает X по ссылке
28      // на изменяемый объект, т.к. вариант с неизменяемым объектом не корректен
29      // из-за члена s, который таким образом скопирован быть не может.
30      // X(X&)
31      //   : s(other.s)
32      // {}
33
34      // Конструктор перемещения есть, но удалён,
35      // т.к. попытка реализации в стандартной форме не корректна:
36      // не удастся сконструировать s из strange&&,
37      // в конструкторе strange(strange&) праводопустимое значение
38      // не привязывается к леводопустимой ссылке.
39      // X(X&&) = delete;
40
41      // Присваивание копированием удалено, т.к. стандартная
42      // реализация попытается вызвать удалённую неявно операцию
43      // присваивания копированием типа strange.
44      // X& operator=(const X&) = delete;
45
46      // Стандартная реализация присваивания перемещением наткнётся
47      // на ту же проблему, поскольку у strange нет описания присваивания
48      // перемещением, и вместо него разрешение перегрузок выберет удалённое
49      // присваивание копированием.
50      // X& operator=(X&&) = delete;
51  };
```

Показанные эффекты могут быть расширены или сужены собственными явными определениями.

Для приведения всех возможных комбинаций реализаций этих функций в систему, автор предлагает сделать два упрощения:

- Конструктор копирования или перемещения и соответствующая операция присваивания реализуются вместе: либо обе, либо ни одна. Нечасто требуется объект который можно копировать или перемещать только при создании или только после.
- Перемещать объект можно всегда, поскольку это лишь изменение размещения места в памяти, где хранится информация о том или ином ресурсе.

В рамках этих допущений все классы можно разделить на две категории:

- ***Перемещаемые, но не копируемые объекты***. Это объекты, которые по смыслу не могут быть скопированы. Для них определяют функции перемещения, функции копирования при этом автоматически удаляются. Это могут быть объекты, соответствующие конкретным аппаратным ресурсам.
- ***Копируемые объекты***. Эти объекты можно копировать, так что они должны иметь реализации функций копирования (возможно, неявные). Если для них возможно перемещение более эффективным, чем копированием, способом, у них должны иметься определения соответствующих функций перемещения. Назовём такие классы эффективно перемещаемыми. Если это не так, они не требуются — функции копирования будут вызваны вместо них, если где-то будет совершена попытка перемещения.

```
#include <iostream>

// Рассмотренный класс fixed_int_array
// копируем и эффективно перемещаем.
class fixed_int_array
{
public:
    // Все 4 функции копирования-перемещения.
    fixed_int_array(const fixed_int_array&);
    fixed_int_array(fixed_int_array&&);
    fixed_int_array& operator=(const fixed_int_array&);
    fixed_int_array& operator=(fixed_int_array&&);

    // Другие члены.
};

// Класс, объекты которого докладывают о своём создании и уничтожении.
class C
{
private:
    int a[1000];
public:
    C()
    {
        std::cout << "C constructed.\n";
    }

    ~C()
    {
        std::cout << "C destructed.\n";
    }

    // Хотя функции копирования и будут
    // описаны и определены неявно, такое их использование
    // более не принято, т.к. наличие
    // нетривиального деструктора - потенциальный
    // сигнал о владении этим классов ресурсов,
    // требующих ручного управления.
    // В данном случае это не так, и нам годятся
    // определения по умолчанию, затребуем их явно,
    // чтобы не использовать устаревшую возможность языка.

    C(const C&) = default;
    C& operator=(const C&) = default;

    // Т.к. пользователем описаны функции копирования,
    // функции перемещения неявно не описываются.
    // Нас это устраивает, т.к. класс нельзя перемещать эффективнее,
    // чем полным копированием.
};
```

Часто специальные функции перемещения и копирования не реализуют не потому, что это невозможно или бессмысленно, а просто потому, что не требуется в программе — в этом случае возможно некорректные их определения также следует удалить явно (достаточно удалить функции копирования, которые в отличие от перемещения не удаляются сами при наличии описанного пользователем деструктора).

Отметим, что у простого класса-агрегата с членами данных только фундаментальных типов и без описаний функций-членов есть неявные описания всех шести специальных функций. Конструктор по умолчанию и деструктор такого класса не выполняют никаких действий на уровне машинного кода, а функции копирования осуществляют по факту побайтовое копирование представления объекта как целового. Если допустить, что специальные функции есть и у не классовых типов в том же смысле, то их поведение будет аналогичным. В языке C это единственно возможное поведение объектов. С точки зрения языка C++, когда такое поведение гарантировано, т.е. является результатом написанного компилятором кода в стандартной форме определения, то соответствующая специальная функция называется ***тривиальной***

(trivial). Стандартные формы определены рекурсивно через почленную инициализацию по умолчанию/копирование/перемещение/вызов деструктора, поэтому чтобы это свойство считалось имеющимся, оно должно выполнятся на любую глубину подобъектов типа, пока не завершится на не классовых подобъектах, для которых выполняется всегда. Наличие любого подобъекта с квалификатором `volatile` отменяет это свойство.

```
1  struct S1
2  {
3      int x;
4
5      // Конструктор копирования не тривиален, т.к. предоставлен пользователем.
6      S()
7          : x(0)
8      {}
9
10     // Остальные специальные функции тривиальны, т.к. используют стандартную форму
11     // и сводятся к соответствующим <<специальным функциям>> не классового
12     // типа int подобъекта x.
13 };
14
15 struct S2
16 {
17     // Конструктор по умолчанию не тривиален, поскольку предоставлен пользователем,
18     // пускай и в форме, эквивалентной стандартной.
19     S2() {}
20
21     // Деструктор тривиален, т.к. используется (явно запрошенная) стандартная
22     // форма и нет подобъектов с нетривиальным деструктором.
23     ~S2() = default;
24 };
25
26 // Копируемый класс без эффективного перемещения,
27 // владеющий ресурсами.
28 struct S3
29 {
30     S3() { /* ... */ }
31     S3(const S3&) { /* ... */ }
32     S3& operator=(const S3&) { /* ... */ }
33     ~S3() { /* ... */ }
34     // Перемещения не описаны, остальное специальные функции не тривиальны.
35 };
36
37 struct S4
38 {
39     S3 s3;
40
41     // То же, что и в S3, т.к. специальные функции сводятся к операциям над ним.
42 };
```

Класс, все специальные функции которого тривиальны, сам называется тривиальным. Не классовые типы данных также относят к тривиальным. Как мы уже сказали, тривиальность соответствует простейшему поведению специальных функций и может помогать компилятору в оптимизациях. Отметим, что тривиальность формально возможна только в случаях использования стандартных форм определений, из чего следует, что, например:

```
1 X::X() {}         // короче, но не тривиально
2 X::X() = default; // чуть длиннее, но тривиально, и следует по возможности
3                   // предпочитать, если такое определение нужно дать явно.
```

9.6.16. Размещение описаний классов

Рассмотрим вопрос расположения описаний и определений классов и их членов.

В первую очередь напомним, что функции-члены, которые определены в определении класса являются неявно встроенными. Такое размещение следует применять только для простых функций, являющихся хорошими кандидатами для встраивания. Если считать определение

класса чистым описанием его интерфейса, где любым определениям не место, можно дать определения функций, которые следует встроить, вне класса со спецификатором `inline` — выбор из этих двух вариантов есть вопрос стилистический.

Классовые типы, имеющие имя данное в их спецификаторе типа или с помощью псевдонима типа, имеют связанность, определяемую по стандартным правилам. Если их область видимости — пространство имён, их связанность внешняя, если в полном квалифицированном имени не встречаются анонимные пространства имён, иначе внутренняя. Связанность класса распространяется на все его вложенные типы (включая вложенные классы), функции-члены и статические члены данных. Локальные классы, как и их члены, связанности не имеют. Благодаря возможности наличия внешней связанности классы могут входить в интерфейсы единиц трансляции.

Для классов, требуемых только в реализации конкретной единицы трансляции, определение класса и всех его членов вне его определения размещаются в ней напрямую. Определения классов, которые являются предоставляемыми для других единиц трансляции интерфейсами, размещают в заголовочном файле вместе с определениями встраиваемых функций, неявно в определении самого класса, или явно вне его. Определения всех остальных функций размещаются отдельно от класса в соответствующей ему единице трансляции. Там же размещаются определения всех статических членов данных класса для единственности их определения.

Приведём в качестве примера реализацию и использование класса динамического массива целых чисел, модернизировав рассмотренную нами ранее реализацию с учётом всего изученного с тех пор материала.

```
1  // Файл int_array.hpp
2  #ifndef UUID_51D5615B_A797_43AC_A198_87EF4D066764
3  #define UUID_51D5615B_A797_43AC_A198_87EF4D066764
4
5  #include <cassert>
6  #include <cstddef>
7
8  namespace examples
9  {
10     // Опишем size_t в нашем пространстве имён,
11     // т.к. он часто будет нужен.
12     using std::size_t;
13
14     // Класс динамического массива из объектов типа int.
15     // Исключения выделения динамической памяти не обрабатываются.
16     class int_array
17     {
18     public:
19         // Создаёт массив из initial_size нулевых элементов.
20         // Также является конструктором по умолчанию за счёт
21         // значения аргумента по умолчанию - создаёт пустой массив.
22         int_array(size_t initial_size = 0);
23         // Конструктор, инициализирующий массив count элементами
24         // по адресу data.
25         int_array(const int* data,size_t count);
26         // Конструктор копирования.
27         int_array(const int_array& other);
28         // Конструктор перемещения.
29         int_array(int_array&& other);
30         // Операция присваивания копированием.
31         int_array& operator=(const int_array& other);
32         // Операция присваивания перемещением.
33         int_array& operator=(int_array&& other);
34         // Деструктор.
35         ~int_array();
36         // Вставка count элементов по адресу data в позицию where.
37         // Если data - нулевой указатель, вставляется count нулей.
38         void insert(size_t where,size_t count,const int data[] = nullptr);
39         // Вставка элемента element в позицию where.
40         void insert(size_t where,int element);
41         // Удаление count элементов с позиции where.
42         // Хотелось называть функцию delete, но это ключевое слово.
43         void erase(size_t where,size_t count = 1);
```

```
        // Перегруженные операции доступа к элементам массива.
        // Встроенные неявно потому что часто используются и коротки.
        int& operator[](size_t i) { assert(i<size_); return p[i]; }
        const int& operator[](size_t i) const { assert(i<size_); return p[i]; }
        // Возвращает число элементов в массиве.
        size_t size() const { return size_; }
        // Явное приведение к bool, истина, если объект не пуст.
        explicit operator bool() const { return size_; }
    private:
        int *p;
        // С подчёрком, чтобы не конфликтовать с соответствующей
        // функцией чтения это члена.
        size_t size_,capacity;

        // Убедиться, что массив имеет ёмкость не меньше
        // required_capacity, и перевыделить блок памяти на больший
        // с сохранением данных, если это не так.
        void ensure_capacity(size_t required_capacity);
    };
}

#endif
```

```
// Файл int_array.cpp
#include "int_array.hpp"

// Вспомогательные функции.
namespace
{
    // Копирование интервалов int'ов по порядку.
    void copy_forward(int to[],const int from[],size_t count)
    {
        while(count--)
        *to++ = *from++;
    }

    // Копирование интервалов int'ов по в обратном порядке.
    void copy_backword(int to[],const int from[],size_t count)
    {
        while(count--)
        to[count] = from[count];
    }

    // Зануление int'ов.
    void zero(int data[],size_t count)
    {
        while(count--)
        *data++ = 0;
    }
}

namespace examples
{
    int_array::int_array(size_t initial_size)
    // Инициализировать члены в пустое состояние
        : p(nullptr),
          size_(0),
          capacity(0)
    {
        // Вставить initial_size нулей в позицию 0.
        insert(0,initial_size);
    }

    int_array::int_array(const int data[],size_t count)
    // Делегировать создание пустого массива.
    // Возможно, не самый эффективный вариант, но показывает,
    // как делегация может упростить реализацию.
```

```
45          : int_array()
46      {
47          // Вставить начальные элементы
48          insert(0,count,data);
49      }
50
51      int_array::int_array(const int_array& other)
52      // Выделить память под столько элементов, сколько
53      // в другом массиве
54          : p(new int[other.size_]),
55      // Этим же числом инициализировать свой размер
56            size_(other.size_),
57      // и ёмкость.
58            capacity(size_)
59      {
60          // Скопировать данные из другого массива.
61          copy_forward(p,other.p,size_);
62      }
63
64      int_array::int_array(int_array&& other)
65      // Украсть все члены из другого массива
66          : p(other.p),
67            size_(other.size_),
68            capacity(other.capacity)
69      {
70          // Занулить указатель, чтобы исходный
71          // объект больше не владел памятью,
72          // и не удалил её в деструкторе или операции присваивания,
73          // т.к. мы теперь владеем этими данными.
74          other.p = nullptr;
75
76          // Чтобы корректно работала операция присваивания,
77          // ёмкость тоже нужно исправить на актуальную.
78          other.capacity = 0;
79
80          // Опционально: чтобы состояние объекта,
81          // из которого было совершено перемещение,
82          // было полностью определённым - пустым - можно исправить size.
83          // Иначе оно не уточняется.
84          // other.size = 0;
85      }
86
87      int_array& int_array::operator=(const int_array& other)
88      {
89          // Если нас присваивают самим себе, ничего
90          // делать не надо. Так быстрее, и иногда
91          // код копирования (не в этом случае, но часто бывает
92          // что) некорректен для копирования себя в себя.
93          if(this!=&other){
94              // Убедиться, что элементы другого массива влезут.
95              ensure_capacity(other.size_);
96              // Установить новый размер, равный размеру
97              // другого массива.
98              size_ = other.size_;
99              // Скопировать элементы другого массива.
100             copy_forward(p,other.p,size_);
101         }
102         return *this;
103     }
104
105     int_array& int_array::operator=(int_array&& other)
106     {
107         // Проверка на присваивание самому себе не требуется,
108         // т.к. попытка использовать объект, идентифицируемый
109         // праводопустимой ссылкой в параметре функции в
110         // других её параметрах (здесь - в качестве неявного
```

```
111         // параметра-объекта) уже есть неопределённое поведение.
112         assert(this!=&other);
113         // Освободить свою старую память.
114         delete[] p;
115         // Украсть память, которой владел исходный объект.
116         p = other.p;
117         // Установить остальные характеристики.
118         size_ = other.size_;
119         capacity = other.capacity;
120         // Отнять владение у исходного объекта аналогично
121         // конструктору перемещения.
122         other.p = nullptr;
123         other.capacity = 0;
124         // other.size = 0;
125         return *this;
126     }
127
128     int_array::~int_array()
129     {
130         // Освободить динамическую память,
131         // в которой хранятся элементы.
132         // Если её украли, то удаление нулевого
133         // указателя ничего не делает.
134         delete[] p;
135     }
136
137     void int_array::insert(size_t where,size_t count,const int data[])
138     {
139         if(where>size_){
140             // Вставка после конца массива не впритык к нему -
141             // нужна ёмкость равная месту вставки плюс число элементов.
142             ensure_capacity(where+count);
143             // Заполним дырку между старыми и новыми элементами нулями.
144             zero(p+size_,where-size_);
145             // Новый размер такой же, как и потребовавшаяся ёмкость.
146             size_ = where+count;
147         }else{
148             // Вставка в смежное с существующими данными место,
149             // нужна ёмкость сколько было раньше плюс новые.
150             ensure_capacity(size_+count);
151             // Переместим старые данные справа от точки вставки,
152             // освобождая место для новых.
153             copy_backward(p+(where+count),p+where,size_-where);
154             // Увеличим логическое число элементов.
155             size_ += count;
156         }
157         if(data)
158             // Новые данные указаны, копируем.
159             copy_forward(p+where,data,count);
160         else
161             // Новых данных нет, зануляем.
162             zero(p+where,count);
163     }
164
165     void int_array::insert(size_t where,int element)
166     {
167         // Эта функция для удобства, она проста.
168         insert(where,1,&element);
169     }
170
171     void int_array::erase(size_t where,size_t count)
172     {
173         // Интервал удаления вне массива или он пуст,
174         // делать нечего.
175         if(where>size_||!count)
176             return;
```

```
        // Если интервал удаления выходит за конец массива,
        // сократим его.
        if(where+count>size_)
            count = size_-where;
        // Переместить влево элементы справа от удалённого
        // интервала.
        copy_forward(p+where,p+(where+count),size_-(where+count));
        // Уменьшить логический размер массива.
        size_ -= count;
    }

    void int_array::ensure_capacity(size_t required_capacity)
    {
        // Если хватает, делать нечего.
        if(capacity>=required_capacity)
            return;
        // Иначе мало.
        // Сделаем новую ёмкость в 1.5 раза больше старой,
        // или начнём с 8 элементов, если было пусто.
        size_t new_capacity = capacity?capacity*3/2:8;
        // Если этого не хватает для новых требований,
        // удовлетворим их.
        if(new_capacity<required_capacity)
            new_capacity = required_capacity;
        // Выделить новый блок памятм.
        int *new_p = new int[new_capacity];
        // Скопировать туда данные.
        copy_forward(new_p,p,size_);
        // Старый блок удалить.
        delete[] p;
        // Записать новый указатель на элементы
        // и новую ёмкость в члены объекта.
        p = new_p;
        capacity = new_capacity;
    }
}
```

```
// Файл main.cpp
#include "int_array.hpp"

#include <iostream>

using namespace std;
using namespace examples;

// Печать содержимого массива.
// Массив принимается по ссылке, чтобы избежать копий,
// неизменяемой, потому что его не нужно в ней менять.
void print_int_array(const int_array& a)
{
    std::cout << '{';
    if(a.size()){
        std::cout << a.elem(0);
        for(size_t i=1;i<a.size();++i)
            std::cout << ',' << a[i];
    }
    std::cout << "}\n";
}

int main()
{
    {
        int_array a1,a2{};
        std::cout << "Default-constructed array: ";
        print_int_array(a1);
        std::cout << "Value-constructed array: ";
        print_int_array(a2);
```

```
31      } // Здесь a1 и a2 уничтожаются и вызываются их
32        // деструкторы (которые впустую удаляют нулевые указатели).
33      {
34          int_array a{10};
35          a[2] = 3;
36          a[7] = 4;
37          a.insert(5,42);
38          a.erase(2);
39          std::cout << "Array with modifications: ";
40          print_int_array(a);
41      } // Память освобождается автоматически!
42      return 0;
43  }
```

9.7. Задачи на использование классов

Данные задачи являются третьим этапом задач, данных в разделе «Динамические структуры данных» и являются их доработкой с целью использования изученных в данном разделе возможностей языка C++.

Во всех вариантах необходимо сделать следующее:

1. Основной класс, соответствующий используемой вами структуре данных, следует превратить в класс с закрытой реализацией. Вместо инкапсуляции в стиле языка C с использованием неполных типов, использовавшихся на предыдущем этапе, следует использовать уровни контроля доступа к членам класса языка C++: сделать все члены данных закрытыми.

   Свободные функции, составлявшие на прошлых этапах интерфейс вашего типа данных, необходимо сделать функциями-членами. Функции инициализации (`init`) следует заменить конструкторами, а `destroy` — деструкторами. Достаточно простые функции следует сделать встраиваемыми.

2. Поскольку ваши классы владеют ресурсами, реализуйте для них все остальные специальные функции, отвечающие за их копирование и перемещение (хотя в имеющемся решении вы их и не используете).

3. Функции, отвечающие за вывод содержимого ваших классов, реализуйте в виде перегрузок операции `<<`, исходя из того, что тип объекта стандартного потока вывода `std::cout` — `std::ostream`. Сохраните при этом общепринятую семантику этих перегрузок, позволяющую использовать их по цепочке.

Следующие дополнительные доработки следует выполнить в соответствующих вариантах:

0. Функцию сложения замените перегрузкой операции `+=`.
1. Функции чтения/записи элементов замените парой перегрузок операции `()`, возвращающих ссылку на элемент для модифицируемых объектов и само значение для не модифицируемых.
2. Добавьте в ваш класс возможность использования его в контекстах, предполагающих контекстное преобразование к логическим значениям, возвращая истину для не пустых объектов.
3. См. вариант 2.
4. См. вариант 1.
5. См. вариант 2.
6. Функцию добавления светлячка в очередь замените перегрузкой операции `<<` над вашей структурой данных и объектом класса светлячка.
7. См. вариант 1.
8. Функцию высадки агента замените перегрузкой операции `<<` над вашей структурой данных и объектом класса агент.
9. Функции чтения/записи элементов замените парой перегрузок операции `[]`, возвращающих ссылку на элемент для модифицируемых объектов и само значение для не модифицируемых.

9.8. Наследование классов

9.8.1. Основы наследования

Классы — основное средство создания объектов с уникальным поведением. Тем не менее, часто требуется создать класс, который похож по поведению на уже имеющийся, и включает в себя его структуру и функциональность. Чаще всего, для включения одной функциональности в другой класс используется ***агрегация (aggregation)*** — включение других объектов в состав представления класса в виде не статических членов данных, то есть именованных подобъектов. Такое отношение также называют ***has-a*** — один объект просто включает другой в качестве одной из своих частей. Благодаря всем средствам автоматизации использования классовых типов, обеспечивающим их корректную инициализацию, уничтожение и прочие специальные операции, этим нетрудно воспользоваться просто сделав объект классового типа частью представления другого класса:

```
1  // Из примера прошлой главы.
2  #include "int_array.hpp"
3
4  // Класс, логическое содержимое которого включает
5  // последовательность целых чисел.
6  class my_class
7  {
8  public:
9      // Перенаправить запрос к члену класса.
10     std::size_t size() const
11     {
12         return data_.size();
13     }
14 private:
15     // Агрегация вектора символов классом my_class:
16     int_array data_;
17 }
```

> Существует множество классификаций отношений между объектами в объектно-ориентированной парадигме. Например, из агрегации часто выделяют ***композицию (composition)*** — подвид отношения has-a, при котором часть целого не является самодостаточной и не может без него осмысленно существовать. В данном разделе мы рассмотрим только основные отношения, имеющие прямые синтаксические проявления в программах.

Встречаются ситуации, когда связь между объектами является достаточно тесной концептуально (класс является более частным случаем другого, более общего) или технически (требуется, чтобы синтаксически операции над новым классом вели себя так же, как над другим). В таком случае можно применить механизм повторного использования кода языка C++, называемый ***inheritance (наследованием)***, соединяющий классы в иерархию. Устанавливаемое им отношение называют ***is-a*** — один объект ***является*** в том числе и другим объектом, более общего типа, но является более частным случаем, добавляя дополнительные возможности, не входящие в исходный класс. Отметим, что в C++ строгое соблюдение этого отношения в данном смысле не гарантируется и может нарушаться различными способами, поэтому наследование как одна из основных идей объектно-ориентированного программирования в её реализации языком C++ является в первую очередь средством повторного использования кода.

Про классы, состоящие в таком отношении говорят, что более частный класс, включающий в себя все элементы общего ***унаследован (inherits)*** от исходного или является ***производным (derived)*** от него или ***дочерним (child)***, а исходный класс для него — ***базовым (base)*** или ***родительским (parent)***.

C++ отличается поддержкой сложных иерархий наследования:

- Наследование может быть сколь угодно широким: один класс может быть базовым для произвольного количества других классов.
- Наследование может быть сколь угодно глубоким: класс, унаследованный от другого, может сам являться базовым для третьего и т.д. В таком случае класс, функциональность которого он непосредственно расширяет, называется ***прямым базовым (direct base)***, что выделяет его из всех «базовых» классов в иерархии наследования, функциональность которых включает данный дочерний транзитивно через них.

- Наследование может быть ***множественным (multiple inheritance)***: один класс может иметь несколько прямых базовых классов, являясь одновременно частным случаем каждого из них.

Иерархии наследования часто изображают в виде ациклических орграфов, где вершинами являются классы, а дуги ведут от производных классов к прямым базовым. Принято наращивать такую иерархию сверху вниз, так что подобное дерево «растёт» вниз при его изображении. Разумеется, допускается наличие в программе любого числа компонент связанности, соответствующих независимым деревьям наследования.

Начнём рассмотрение с простейшего случая наследования в виде отношения ровно двух классов. После имени класса в его определении можно указать двоеточие и имя другого полного классового типа который становится при этом его прямым базовым. При наследовании в производный класс вносятся два основных изменения.

Во-первых, объект типа прямого базового класса включается в производный в качестве безымянного базового подобъекта (base subobject). Это второй из двух типов подобъектов классов, включая нестатические члены данных. Указать инициализатор для базового подобъекта можно в списке инициализаторов конструктора производного класса. При этом в качестве имени объекта используется имя самого базового класса, и такой инициализатор должен идти перед инициализаторами именованных подобъектов:

```
1  struct base
2  {
3      using my_type = int;
4
5      my_type x;
6
7      // Конструктор класса base:
8      explicit base(my_type x)
9          : x(x)
10     {
11         std::cout << "Constructed base!\n";
12     }
13 };
14
15 struct derived : base
16 {
17     int y;
18
19     derived()
20         // Инициализатор базового подобъекта:
21         : base(42),
22         // Инициализаторы именованных подобъектов:
23         y(10)
24     {
25         std::cout << "Constructed a derived!\n";
26     }
27 };
```

В данном примере инициализация объекта типа `derived` конструктором начинается с инициализации базового подобъекта типа `base`, включая инициализацию подобъекта `x` и выполнение тела конструктора `base`, и только после этого инициализируются части `derived`, введённые непосредственно в нём — подобъект `y` и выполняются действия из тела его конструктора.

Во-вторых, поиск имён модифицируется таким образом, что после просмотра области видимости производного класса, просматривается область видимости прямого базового. Таким образом в производном классе могут быть видны имена вложенных типов и статических членов прямого базового, например, псевдоним типа в рассмотренном примере может быть поименован не только как `base::my_type`, но и как `derived::my_type`, то же возможно и для вложенных классов, статических членов данных и функций.

С такой модификацией поиск имён также может находить имена не статических членов данных базового класса в контекстах, где выборка осуществляется из объекта производного типа. В таком случае они идентифицируют соответствующие подобъекты безымянного подобъекта, соответствующего прямому базовому типу:

```
struct base
{
    int x = 10;
};

struct derived : base
{
    void f()
    {
        // Доступ к члену x безымянного подобъекта прямого
        // базового типа base в объекте типа derived.
        // После поиска имени x в derived до использования
        // (неудачно), просматривается область видимости класса base,
        // где оно находится как нестатический член данных.
        x = 20;
        this->x = 30;
    }
};

void f()
{
    base b;
    // То же самое.
    b.x = 15;
}
```

Нередко, говоря о наследовании, его пытаются представить как «копирование» членов базового класса в производный, или говорят о наследовании классом отдельных членов. И хотя, действительно, одной из основных целей использования наследования в языке C++ является повторное использование кода, следует понимать, что в реальности ***подобное поведение есть лишь следствие наличия в производном классе подобъекта типа базового и специфического механизма поиска имён***, позволяющего обычно использовать имена сущностей базового класса, словно они являются принадлежностью производного. Само понятие «наследование», формально, применимо только к классам.

Например, как и с любыми последовательностями просмотра областей видимости, при наследовании допустимо скрытие, для преодоления которого язык допускает квалифицированные имена в операциях выборки:

```
struct base
{
    int x;
};

struct derived : base
{
    int x;

    void f()
    {
        // В многоуровневом агрегате derived есть
        // два подобъекта с именем x: base::x и derived::x.
        // При неквалифицированном поиске из derived
        // находится derived::x, а base::x скрыт.
        // Для доступа к скрытому имени может использоваться
        // квалифицированное имя: ниже происходит
        // запись значения из base::x в derived::x.
        /* derived:: */ x = base::x;
    }
};

void g()
{
    derived d;
    // Для уточнения имён подобъектов при доступе к ним
    // через операции выборки, правый операнд прямой или
    // косвенной выборки может быть квалифицированным именем,
```

```
29      // именующим базовый класс объекта, из которого она осуществляется.
30      d.base::x = d.x;
31  }
```

Как отражение отношения is-a, указатель на объект производного типа может быть неявно приведён к указателю на прямой базовый тип (это приведение по классификации входит в приведения указателей). Результат — адрес базового подобъекта, это ещё один способ доступа к нему в обход отсутствия у него имени. Такое приведение типов называют ***приведением «вверх» (upcast)***, согласно принятой схеме представления графов наследования.

Кроме этого, ссылки на базовый класс могут напрямую привязываться к объектам производного класса. Фактически ссылка привязывается к подобъекту базового типа, но если полный объект является временным, расширяется время хранения всего полного объекта целиком.

Обратное приведение типа указателя или ссылки на базовый класс к производному возможно, но требует явного использования операции `static_cast`. Если исходный объект базового типа не являлся при этом базовым подобъектом производного, наступает неопределённое поведение. Такое приведение называют ***приведением «вниз» (downcast)***. Его можно использовать только, если косвенным образом известно, что имеющийся адрес или ссылка на объект базового типа обязательно является подобъектом требуемого производного.

```
1   struct base
2   {
3       int x;
4   };
5
6   struct derived : base
7   {
8       int y;
9   };
10
11  void f()
12  {
13      derived d;
14
15      // pb - адрес базового подобъекта типа base в d.
16      base* pb = &d;
17
18      // rb - ссылка на базовый подобъект временного объекта
19      //      типа derived. Весь этот временный объект будет существовать
20      //      до конца области видимости ссылки.
21      base&& rb = derived{};
22
23      // pd - адрес d
24      derived* pd = static_cast<derived*>(pb);
25
26      // rd привязано к полному временному объекту,
27      //    созданному в инициализаторе rb.
28      derived& rd = static_cast<derived&>(rb);
29
30      base b;
31
32      // ОШИБКА: b не является базовым подобъектом
33      //         объекта производного типа derived.
34      derived* pd2 = static_cast<derived*>(&b);
35
36      // ОШИБКА: аналогично.
37      derived& rd2 = static_cast<derived&>(b);
38  }
```

Благодаря такому свойству для ссылок, возможен вызов функций базового подобъекта на производном. При нахождении поиском имён не статической функции в базовом классе при поиске через производный, в процессе её вызова ссылка на неявный параметр-объект привязывается к соответствующему подобъекту:

```
1   struct base
2   {
```

```
3       int x = 0;
4
5       void f()
6       {
7           ++x;
8       }
9   };
10
11  struct derived : base
12  {
13      void g()
14      {
15          // Вызвать функцию f с неявным параметром-объектом,
16          // привязанным к базовому подобъекту типа base
17          // в объекте derived, привязанном к неявному параметру-объекту
18          // данной функции.
19          f();
20      }
21  };
22
23  void h()
24  {
25      derived d;
26
27      // Вызвать функцию f с неявным параметром-объектом,
28      // привязанным к базовому подобъекту типа base
29      // в объекте d.
30      d.f();
31
32      d.g();
33  }
```

Рассмотренное ранее на членах данных скрытие имён при наследовании наиболее интересно трактуется для случая скрытия функций-членов. В таком случае оно соответствует не добавлению новой алгоритмической функциональности в производный класс, как частный случай базового, а, потенциально, полной её замене:

```
1   struct base
2   {
3       void f() {}
4   };
5
6   struct derived : base
7   {
8       // Скрывает в derived base::f.
9       void f(int) {}
10
11      void g()
12      {
13          // Нахождение нужного множества перегрузок
14          // через квалифицированное имя.
15          base::f();
16      }
17  };
18
19  void h()
20  {
21      derived d;
22
23      // Вызывает derived::f(int).
24      d.f(2);
25
26      // ОШИБКА: поиск имён находит множество перегрузок
27      //         в первой по порядку просмотра области видимости
28      //         и останавливается. Это единственная функция
29      //         derived::f(int), которая не годна.
```

```
30      d.f();
31
32      // Явно квалифицировать требуемое пространство имён-класс.
33      d.base::f();
34  }
```

Если требуется именно дополнение функциональности новой перегрузкой функции, а не её замена, то описание using может привнести множество перегрузок имени функции из базового класса в множество перегрузок производного, и тем самым заменить скрытие расширением множества перегрузок:

```
1       struct base
2       {
3           void f() {}
4       };
5
6       struct derived
7       {
8           // Ещё одно описание всех описаний f в base.
9           using base::f;
10
11          // Перегрузка предыдущих описаний f в derived.
12          void f(int) {}
13      };
14
15      void g()
16      {
17          derived d;
18
19          // Множество перегрузок при поиске имени f
20          // в derived содержит base::f() и derived::f(int).
21          d.f();
22          d.f(42);
23      }
```

Это единственное допустимое применения описания `using` в классовой области видимости — оно допустимо только с именами из областей видимости базовых классов.

Если требуется промежуточный вариант, где функциональность базового класса уточняется, реализация в производном классе может выполнить дополнительные действия до и/или после вызова одноимённой функции базового через квалифицированное имя, возможно, модифицируя при этом аргументы и/или возвращаемое значение:

```
1   struct base
2   {
3       int f(int x)
4       {
5           std::cout << "Call to f() with " << x << '\n';
6           return 42;
7       }
8   };
9
10  struct logged_base : base
11  {
12      // Скрытие base::f
13      int f(int x)
14      {
15          // Сделать что-то до base::f.
16          std::cout << "Before call to base\n";
17          // Вызов базовой реализации с модифицированным аргументом:
18          int res = base::f(x+1);
19          // Сделать что-то после base::f.
20          std::cout << "After call to base\n";
21          // Вернуть результат base::f с модификацией.
22          return res+1;
23      }
24  };
```

Таким образом, уточнение поведения функций в производных классах может осуществляться без копирования функциональности из базового класса. Отметим, язык не требует как-либо использовать скрываемую функцию в реализации скрывающей, так что новая функция может иметь никак не связанное со старой поведение.

В теории объектно-ориентированно программирования имеет место одно из определений отношения `is-a` под названием ***Принцип подстановки Барбары Лисков (Liskov substitution principle)***: все свойства объектов базового типа должны быть также справедливы и для любых объектов типов, производных от данного. Это утверждение также известно как свойство замещения, при котором замена объектов базового типа на производные не влияет нежелательным образом на функционирование программы. Хотя подобное свойство может быть полезным, на уровне требований семантики языка C++ оно, как показано выше, может не соблюдаться. Именно поэтому его следует рассматривать как средство повторного использования кода в одном из первых смыслов.

Мы рассмотрели примеры с отношением только двух классов. В случае расширения иерархии в ширину, т.е. использования одного класса как базового для нескольких, никаких дополнительных эффектов пока наблюдаться не будет.

9.8.2. Специальные члены класса и наследование

Рассмотрим поведение специальных функций в производных классах.

Конструкторы не имеют имён, поэтому расширение алгоритма поиска имён для наследования класса само по себе на них не влияет. Например, имеется ли у класса неявное описание конструктора по умолчанию, и какова его форма, определяется по стандартным правилам с учётом того, что базовый подобъект рассматривается как один из подобъектов:

```
1  struct base1
2  {
3      base1() {}
4  };
5
6  struct derived1 : base1
7  {
8      // Имеется неявное определение конструктора по умолчанию,
9      // поскольку описанных пользователем конструкторов нет.
10     // Этот конструктор инициализирует базовый подобъект
11     // по умолчанию, тело его пусто.
12 };
13
14 struct base2
15 {
16     base2(int x) {}
17 };
18
19 struct derived2 : base2
20 {
21     // Неявное определение конструктора по умолчанию удалено,
22     // т.к. один из подобъектов данного (базовый) по умолчанию
23     // не инициализируем.
24 };
```

Специальная функция операции присваивания копирования по известным нам правилам, не относящимся к наследованию, либо описана в классе явно, либо получает неявное описание (пускай даже удалённое). Как следствие, функции с именем `operator=` базового класса всегда ей скрыты, хотя могут быть поименованы в данном с помощью квалифицированных имён. То же происходит и со специальными функциями перемещения.

Часто говорят, что «специальные функции не наследуются», но, как мы показали ранее, понятие наследование не применимо формально ни к каким членам класса, всё опять сводится к уже известными правилам поиска имён.

Реализации по умолчанию четырёх специальных функций, ответственных за копирование значений классовых типов, трактуют базовый подобъект так же, как и другие подобъекты, вызывая при почленной обработке соответствующие функции на них:

```
1  struct base
2  {
```

```
3       int x;
4   };
5
6   struct derived : base
7   {
8       int y;
9
10      /* Класс derived имеет неявное определение конструктора копирования
11      по умолчанию, эквивалентное
12      derived(const derived& other)
13          // Прямая инициализация через конструктор копирования базового
14          // подобъекта (его-параметр ссылка привязывается к базовому подобъекту
15          // типа derived объекта other):
16          : base(other),
17          // Прямая инициализация обычного не статического члена.
18          y(other.y)
19      {}
20      */
21  };
```

Деструкторы уничтожают все подобъекты в порядке, обратном порядку конструирования. Это также относится и к уничтожению базовых подобъектов: поскольку они создавались первыми, они уничтожаются последними, после всех именованных не статических членов класса.

Если в базовом классе имелся большой набор перегрузок конструкторов, терять возможность конструирования объектов производного класса теми же способами (в предположении, что этого достаточно) неудобно:

```
1   struct base
2   {
3       // Класс base имеет много полезных перегруженных конструкторов:
4       base(int x) {}
5       base(double x,char y) {}
6       template<typename T> base(T* p) {}
7   };
8
9   // Нам нужно унаследоваться от base, чтобы просто добавить одну функцию-член:
10
11  struct derived1 : base
12  {
13      void f() {}
14
15      // Мы хотим, чтобы у derived1 был такой же доступный набор конструкторов,
16      // как и у base, но они скрыты, поэтому приходится дублировать их определения,
17      // вызывающие их по цепочке в этом классе:
18
19      derived1(int x) : base(x) {}
20      derived1(double x,char y) : base(x,y) {}
21      template<typename T> derived1(T* p) : base(p) {}
22
23  };
```

Начиная с C++11, показанное выше дублирование можно устранить с помощью специальной формы описания `using`, в которой указано квалифицированное внедрённое имя прямого базового класса. Такое описание действует подобно уже рассмотренным формам описания `using` в классах для расширения множества перегрузок, но действует на формально безымянное множество конструкторов, за исключением специальных функций. В таком случае возникает ситуация, когда конструктор может быть вызван для инициализации объекта производного типа, и является для него ***унаследованным конструктором (inherited constructor)***. В таком случае производный объект инициализируется так же, как и стандартной формой конструктора по умолчанию, за исключением инициализации подобъекта, унаследованный от которого конструктор используется, который для него и применяется.

С его помощью пример выше может быть записан короче.

```
1   // Продолжение предыдущего примера.
2
```

```
3  struct derived2 : base
4  {
5      void f() {}
6
7      // Описание унаследованных конструкторов.
8      using base::base;
9  };
10
11 void f()
12 {
13     // Инициализация объекта типа derived2
14     // унаследованным конструктором base::base(int);
15     // Все подобъекты derived2, за исключением базового
16     // подобъекта типа base (никакие) инициализируются
17     // по умолчанию, а подобъект типа base - через
18     // вызов base::base(45) в стандартном порядке.
19     derived2 d2(45);
20 }
```

Для деструкторов, также не имеющих имён, подобной конструкции нет за ненадобностью.

9.8.3. Контроль доступа при наследовании классов

При наследовании перед именем прямого базового класса может указываться уровень доступа. Если уровень доступа не указан, для классовых типов, объявленных с ключом `class` подразумевается `private`, а для `struct` (как было во всех ранее приведённых примерах) — `public`, как и для прочих членов до первой явной секции уровня доступа.

Автор предлагает следующую формулировку правила, позволяющую легко запомнить все возможные ситуации с комбинациями уровней доступа: указываемый при наследовании уровень доступа следует трактовать как уровень доступа, который имело бы описание базового подобъекта в производном, если бы оно существовало в явном виде. В таком представлении для того, чтобы иметь доступ к члену базового подобъекта, необходимо иметь доступ как к самому подобъекту, так и его члену. Как обычно, проверка уровня доступа ортогональна поиску имён и осуществляется как его последний шаг. Рассмотрим возможные комбинации на примерах:

```
1  class base
2  {
3  public:
4      // Открытый член base.
5      static void f() {}
6  private:
7      // Закрытый член base.
8      static void g() {};
9  };
10
11 // Наследование с сохранением уровня защиты.
12 class derived1 : public base
13 {
14 public:
15     void h()
16     {
17         // f найдено как base::f
18         // после отсутствия в блоке и derived1.
19         // Доступ к базовому подобъекту есть у всех (public наследование),
20         // доступ к члену f тоже (public член base).
21         f();
22         // ОШИБКА: g найдено как закрытый член
23         // base::g, но доступ осуществляется
24         // не из класса base.
25         g();
26     }
27 };
28
29 // private подразумевается для class.
```

```
30  class derived2 : /* private */ base
31  {
32  public:
33      void h()
34      {
35          // Базовый подобъект типа base может трактоваться как закрытый член
36          // derived2, h имеет доступ как функция-член h.
37          // Сам член base::f открытый.
38          f();
39      }
40  };
41
42  void h()
43  {
44      // Всё открыто и доступно отовсюду.
45      derived1::f();
46      // ОШИБКА: g - закрытый член base, но доступ не из него.
47      derived1::g();
48
49      // Базовый подобъект типа base может трактоваться как закрытый член,
50      // поэтому ко всем его членам доступа не из derived2 нет.
51      // ОШИБКА - f не доступно.
52      derived2::f();
53      // ОШИБКА - g не доступно.
54      derived2::g();
55  }
```

При наследовании почти всегда используется открытое наследование, поэтому не забывайте указывать `public` перед именем прямого базового типа, если вы используете в определении нового класса ключ `class`. Причина этому такова: проверка уровня доступа к базовому подобъекту осуществляется также и в процессе операций приведения типов указателей или ссылок между базовыми родительскими и дочерними типами. Она осуществляется при приведении в любом направлении, но рассматривается в любом случае уровень доступа базового подобъекта в дочернем классе. При отсутствии доступа приведение типов невозможно, как неявное, так и явное. При этом говорят, что приведения возможны только к или из ***доступных базовых (accessible base)*** классов, причём как обычно для проверок доступа, их успешность зависит от места в программе, где они предпринимаются:

```
1   struct base {};
2
3   class derived : /* private */ base
4   {
5       void f()
6       {
7           // С точки зрения самого derived,
8           // базовый подобъект с закрытым уровнем
9           // доступа доступен.
10          base* bp = this;
11      }
12  };
13
14  void f()
15  {
16      derived d;
17      // ОШИБКА: базовый подобъект типа base в d
18      //         здесь не доступен.
19      base* pb = &d;
20      // ОШИБКА: аналогично.
21      base& rb = d;
22
23      // ОШИБКА: вниз по иерархии тоже нельзя, если из
24      //         производного типа базовый не доступен.
25      derived* pd = static_cast<derived*>(pb);
26  };
```

Когда используется не-`public` наследование, объект не может быть привязан к ссылке на соответствующий базовый тип вне его членов, что является очевидным нарушением отношения

`is-a` — с точки зрения окружающих, производный объект не может трактоваться как базовый. Это основная причина, по которой чаще всего применяется открытое наследование.

Для членов класса и наследования имеется ещё один уровень защиты: `protected` («защищённый»). Он разрешает доступ из того же класса, а также производных от него:

```
1  class base
2  {
3  protected:
4      // Защищённый член.
5      static void f() {}
6  };
7
8  class derived : public base
9  {
10     void g()
11     {
12         // Производный класс имеет доступ
13         // к защищённым членам базового:
14         f();
15     }
16 };
17
18 void h()
19 {
20     // Доступ к защищённым членам снаружи
21     // невозможен:
22     // ОШИБКА - f не доступно.
23     base::f();
24 }
```

Этот уровень доступа к членам класса применяется, чтобы дать доступ производным классам к деталям реализации базового без нарушения инкапсуляции. В том числе это используется, чтобы вынести общие для нескольких производных классов части реализации в общий базовый класс, чтобы избежать дублирования:

```
1  class base_checked
2  {
3  protected:
4      // Сложная функциональность, которая пригодится
5      // в реализации многих производных классов.
6      bool check(int x)
7      {
8          return x>=10&&x<99&&x!=42;
9      }
10 };
11
12 class derived_input_check : public base_checked
13 {
14 public:
15     bool do_something(int x)
16     {
17         if(!check(x)){
18             std::cout << "Bad x!\n";
19             return false;
20         }
21         // ...
22         return true;
23     }
24 };
25
26 class derived_output_check : public base_checked
27 {
28 public:
29     int do_something_else(int x,int y)
30     {
31         int res = x+y;
```

```
32          if(!check(res)){
33              std::cout << "Bad sum!\n";
34              return 0;
35          }
36          return res;
37      }
38  };
```

Такой подход используется только в иерархиях близких по функциональному использованию классов, потому что несмотря на нестрогость отношения ***is-a*** в C++, оно является одним из удобных соглашений, на основе которых строится модульная архитектура программ.

Друзья класса считаются с точки зрения контроля доступа эквивалентными в правах с его членами, и потому имеют доступ к защищённым членам базовых классов:

```
1  class base
2  {
3  protected:
4      int x;
5  };
6
7  class derived : public base
8  {
9      friend void f();
10 };
11
12 void f()
13 {
14     derived d;
15     // f имеет доступ, аналогичный членам класса derived,
16     // поэтому в ней доступен защищённый член base::x.
17     d.x = 5;
18 }
```

Описания, введённые описанием `using` в определении производного класса, получают уровень доступа секции, в которой находится это описание, поэтому такая конструкция может использоваться для изменения уровня доступа к имени в производном классе (если он сам имеет к нему доступ), причём в любую сторону:

```
1  class base
2  {
3  public:
4      // f - открытый член.
5      void f();
6  protected:
7      // g - защищённый член.
8      void g();
9  };
10
11 class derived : public base
12 {
13 public:
14     // Сделать base::g открытым в derived.
15     // Без этого описания он бы остался защищённым.
16     using base::g;
17 private:
18     // Запретить доступ к base::f из производныз классов,
19     // а также снаружи (хотя это можно будет обойти
20     // квалифицированным именем, потому применяется редко).
21     using base::f;
22 };
```

9.8.4. Цепочки наследования

Производные классы могут, в свою очередь, являться базовыми для других производных классов, таким образом иерархия наследования может быть сколь угодно глубокой не только в ширину, но и в высоту. В таком случае у объекта может быть несколько базовых классов,

но только тот, от которого он непосредственно унаследован — тот, который указан в его определении — называют ***прямым*** базовым:

```
struct A {};
struct B : A {};
struct C : B {};

// A и B - базовые классы C, B - его прямой базовый класс.
```

В таком случае объект глубокого производного типа содержит в своём представлении подобъекты всех базовых типов за счёт их вхождения друг в друга по цепочке. Поиск имён в таком классе просматривает области видимости всех базовых классов по порядку начиная с прямого базового, и кончая классом, который ни от чего не унаследован.

При контроле доступа в цепочках наследования для успеха необходимо наличие доступа к самому имени и всем подобъектам, находящимся в цепи наследования от класса, с которого начался поиск имён, до класса, содержащего описание этого имени. Покажем это на примере защищённого наследования, свойства которого видны только в цепочках из более, чем двух классов:

```
struct A
{
    void f();
};

class B : protected A {};

class C : public B
{
public:
    void g()
    {
        // Подобъект прямого базового типа
        // доступен из самого класса всегда.
        // Подобъект типа A имеет защищённый
        // уровень доступа в B, f имеет к нему
        // доступ как член производного класса.
        // Сам член A::f, найденный поиском имён,
        // открытый, то есть доступен всем.
        // Все имена в цепочке поиска доступны,
        // доступ разрешён.
        f();
    }
};
```

Большинство из приведённых ранее примеров работают в глубоких иерархиях аналогично. В первую очередь, приведения типов указателей на объекты классов и привязка ссылок могут осуществляться не только между прямым базовым классом и производным от него, но и на любую глубину, при соблюдении остальных указанных нами условий. С точки зрения механизма разрешения перегрузок из двух приведений к указателем или ссылкам на базовые типы лучше то, которое «ближе» по иерархии:

```
struct base {};
struct derived : base {};
struct derived2 : derived {};

void f(const base&);    // 1
void f(const derived&); // 2

void g()
{
    // Выбирается перегрузка 2, т.к.
    // между derived и derived2 меньше уровней
    // наследования, чем между base и derived2.
    f(derived2{});
}
```

Определение конструкторов наследования может содержать имя только прямого базового типа, не прямые не допускаются, но можно использовать наследование конструкторов в каждом из нескольких членов цепочки наследования подряд явно.

9.8.5. Виртуальные функции

Одним из понятий, которое обычно напрямую связывают с объектно-ориентированной парадигмой программирования, является полиморфизм. Это распространённое заблуждение, поскольку это понятие намного шире.

Под ***полиморфизмом (polymorphism)*** подразумевают предоставление одинакового интерфейса работы с сущностями разных типов. Речь идёт о том, что одинаковые языковые конструкции могут соответствовать различному поведению в зависимости от типов их параметров.

По тому, когда совершается выбор конкретного алгоритма в зависимости от типов параметров, полиморфизм делят на статический (во время трансляции) и динамический (во время выполнения программы).

Мы уже многократно встречались с примерами статического полиморфизма в языке C++: в первую очередь, это перегрузка функций и операций, включая встроенные в язык операции. Выбор соответствующих перегрузок осуществляется для них транслятором на основе информации о типе их аргументов или операндов с точки зрения семантики языка C++.

Попробуем вначале реализовать динамический полиморфизм собственными силами. Пускай требуется разработать и реализовать интерфейс драйверов блочных устройств в ядре операционной системы. Блочным устройством называется устройство, описываемые размером блока в байтах и их количеством, которое позволяет считывать содержимое блоков в память и записывать в них данные, хранящиеся в памяти, по номерам блоков.

С точки зрения построения иерархии классов разумно начать её с класса «блочное устройство», в котором заключить свойства, общие для всех блочных устройств, после чего унаследовать от него классы, соответствующие блочным устройствам разных типов, в которых реализовать по-своему для каждого операции чтения и записи блоков. Мы хотим, чтобы другие части операционной системы могли работать с интерфейсом произвольного блочного устройства, каким бы не была его реализация.

Работать для этого напрямую с базовым типом блочного устройства, очевидно, нельзя, т.к. он не содержит функциональности, специфичной для конкретного типа устройства. Чтобы работать с произвольными типами, мы могли бы воспользоваться шаблонами, но тогда вся часть ОС, использующая этот интерфейс должна быть переписана с их использованием. Поскольку выбор специализаций осуществляется на этапе трансляции, а используемый тот или иной тип блочного устройства будет известен только во время работы ОС, дальнейшее развитие этой идеи черезвычайно осложняется.

Вместо этого можно отложить выбор реализаций до этапа работы программы. Для этого будем в самой структуре базового типа блочного устройства хранить указатели на функции, реализующие необходимые алгоритмы работы с блочным устройством:

```
1  #include <cstddef>
2  #include <cstdint>
3
4  // Предположим, что все блочные устройства работают с блоками по 512 байт.
5  // В реальности это не так...
6  constexpr std::size_t block_size = 512;
7
8  class block_device;
9
10 // Таблица указателей на функции для конкретного типа устройства.
11 // Поскольку она одинакова для всех устройств одного типа, вынесем
12 // её в отдельный класс.
13 struct block_device_functions
14 {
15     // Возвращает число блоков на устройстве.
16     std::size_t (*block_count)(const block_device& device);
17     // Считывает данные указанного блока с устройства, возвращает успешность.
18     bool (*read)(const block_device& device,void* data,std::size_t block);
19     // Записывает указанные данные в блок на устройстве, возвращает успешность.
20     bool (*write)(block_device& device,const void* data,std::size_t block);
```

```
21  };
22
23  class block_device
24  {
25  public:
26      // Сохранить в объекте ссылку на таблицу функций-реализаций.
27      block_device(const block_device_functions& vfptr)
28      : vfptr(vfptr)
29      {
30      }
31
32      // Все методы вызывают функции реализации по указателям
33      // из таблицы, передавая дополнительно адрес этого объекта.
34
35      std::size_t block_count() const
36      {
37          return vfptr.block_count(*this);
38      }
39
40      bool read(void* data,std::size_t block) const
41      {
42          return vfptr.read(*this,data,block);
43      }
44
45      bool write(void* data,std::size_t block)
46      {
47          return vfptr.write(*this,data,block);
48      }
49  private:
50      const block_device_functions& vfptr;
51  };
52
53  // --- Реализация работы с жёстким диском ---
54
55  // Реализации работы с жёстким диском нужно хранить дополнительные
56  // данные, связанные с объектом-устройством, для своей работы.
57  // Включим их в производный класс.
58  class hdd : public block_device
59  {
60  public:
61      // Диски идентфицируются числами (для примера).
62      hdd(std::uint32_t id)
63      : block_device(hdd_functions),id(id)
64      {
65      }
66  private:
67      // Дополнительные данные для работы с жёстким диском.
68      std::uint32_t id;
69
70      static std::size_t block_count(const block_device& device)
71      {
72          // Поскольку мы можем быть вызваны только для объектов-блочных
73          // устройств, являющихся подобъектами объекта типа hdd,
74          // мы всегда можем провести приведение типов "вниз",
75          // чтобы добраться до полного объекта, содержащего
76          // специфичную для нашего типа устройства информацию.
77          const hdd& disk = static_cast<const hdd&>(device);
78          std::size_t res = 0;
79          // Узнать размер диска с идентификатором disk.id.
80          return res;
81      }
82
83      // Реализованы аналогично block_count где-то ещё.
84      static bool read(const block_device& device,void* data,std::size_t block);
85      static bool write(block_device& device,const void* data,std::size_t block);
86
```

```
87      // Таблица указателей на функции-реализации из этого класса.
88      static constexpr block_device_functions hdd_functions = {
89          block_count,
90          read,
91          write
92      };
93
94  };
95
96  // --- Пример использования показанной реализации ---
97
98  // Функция, работающая с произвольным блочным устройством.
99  void f(block_device& device)
100 {
101     auto count = device.block_count();
102     char buf[block_size];
103     for(std::size_t i=0;i<count;++i){
104         device.read(buf,i);
105         // Сделать что-то с данными в buf.
106     }
107 }
108
109 // Другая функция, использующая f.
110 void g()
111 {
112     // Создать устройство, соответствующее диску с идентификатором 0.
113     hdd disk(0);
114     // Передать его функции, которая работает с любыми устройствами.
115     f(disk);
116 }
```

В этом примере нам потребовалось самостоятельно написать функции, перенаправляющие вызовы с объекта блочного устройства на реализации, заняться ручным приведением типов и оставить функции реализации статическими. Всю эту работу можно возложить на компилятор.

Функции, для которых необходимо осуществить выбор необходимой реализации в зависимости от того, подобъектом какого полного объекта является тот, на котором она вызывается, называют ***виртуальными (virtual)***. Они описываются с помощью одноимённого спецификатора, применённого к описанию не статической функции, который действует на каждое описание функции отдельно, без учёта её перегрузок. Класс, содержащий хотя бы одну виртуальную функцию, является ***полиморфным (polymorphic)***. Все классы, унаследованные от полиморфного, тоже будут полиморфными. Для каждого полиморфного типа транслятором генерируется ***таблица виртуальных функций (virtual function table)***, аналогичная созданной нами вручную выше, которая содержит адреса реализаций всех виртуальных функций, описанных в классе. Её адрес сохраняется в специальном не статическом члене данных класса, который является деталью реализации и программисту напрямую не доступен.

Изначально определённая виртуальная функция задаёт первый вариант её реализации, и её адрес заносится в таблицу виртуальных функций в выделенную для неё позицию. Он может быть заменён любым производным классом, путём включения в него идентичного определения функции, заканчивающегося идентификатором со специальным значением `override`. Он используется только на описании в теле класса, если определение дано отдельно, спецификатор `virtual` на таких определениях не требуется, хотя и допустим. Такое определение является не скрывающим, а ***переопределяющим (overrider)***. Использовать `override` в описаниях, не являющихся переопределениями, нельзя, поэтому это ключевое слово служит для проверки компилятором намерений программиста, чтобы не допустить случайного перекрытия вместо переопределения из-за случайных ошибок в описании. `override` появился только в C++11, и, хотя он не требуется для создания переопределения, с этого момента компиляторы выдают предупреждения для случаев переопределения без его использования, чтобы отследить и обратный случай: нечаянное переопределение. Переопределение считается соответствующим всем производным классам в иерархии наследования от класса, в котором дано, и до ближайшего дочернего, в котором переопределено снова, т.е. класс, в котором виртуальная функция не переопределена получает такое же значение в запись таблицы виртуальных функций, ей соответствующую, что и прямой базовый. Таким образом, в языке C++ разрешено многократно заменять и/или расширять поведение виртуальных функций в иерархии наследования.

В отличие от ***статического типа (static type)*** объекта — типа с точки зрения системы

типов, используемой на этапе трансляции, ***динамическим типом (dynamic type)*** объекта полиморфного типа называют тип наиболее полного производного объекта, для которого данный является базовым. Понятие динамического типа применимо только во время выполнения программы, и для не полиморфных типов (включая не классовые) совпадает со статическим. При вызове виртуальной функции происходит обращение к таблице виртуальных функций, чтобы вызвать указанное в нём ***последнее переопределение (final overrider)*** — определение на объекте, соответствующему динамическому типу данного, или ближайшего от него подобъекта в иерархии наследования:

```
1  // Полиморфный класс с виртуальной функцией.
2  // Она является последним переопределем самой себя в нём.
3  struct A
4  {
5      virtual void f()
6      {
7          std::cout << "A::f\n";
8      }
9  };
10
11 struct B : A
12 {
13     // f() не переопределена в B,
14     // для него последним её переопределением
15     // является A::f.
16
17     // Ещё одна виртуальная функция.
18     virtual void g()
19     {
20         std::cout << "B::g\n";
21     }
22 };
23
24 struct C : B
25 {
26     // Переопределить в C и f(), и g().
27     // Эти определения - последние их переопределители в C.
28
29     void f() override
30     {
31         std::cout << "C::f\n";
32     }
33
34     void g() override
35     {
36         std::cout << "C::g\n";
37     }
38 };
39
40 void h()
41 {
42     C c;
43     A& ra = c;
44     // Вызывает C::f(), поскольку C - полиморфный класс,
45     // и динамический тип объекта, привязанного к ra - C.
46     ra.f();
47 }
```

Механизм виртуальных функций работает независимо от видимости или доступа к изначальному описанию виртуальной функции:

```
1  class A
2  {
3  private:
4      virtual void f() {}
5  };
6
```

```
class B : public A
{
public:
    // Скрытие A::f.
    void f(int) {};
};

class C : public B
{
private:
    // Переопределение A::f(), хотя это описание
    // не видно, а если бы и было видно, то недоступно.
    void f() override {}
}
```

Таким образом, если виртуальная функция предназначена только для переопределения в дочерних классах, но не для вызовов из них, её можной оставить закрытой.

При необходимости вызов виртуальной функции для динамического типа можно подавить путём использования квалифицированного имени виртуальной функции. Это часто делают, когда переопределение опирается на реализацию в базовом классе:

```
struct base
{
    virtual void f()
    {
        std::cout << "base::f";
    }
}

struct derived
{
    void f() override
    {
        // Бесконечный цикл, если только это не вызвано
        // для объекта производного от derived типа,
        // у которого есть более низкое в иерархии
        // переопределение.
        f();

        // Вызов base::f в явном виде без использования
        // динамического типа объекта.
        base::f();
    }
};
```

При хранении объекта полиморфного типа в динамической памяти операции `delete` необходимо вызвать деструктор динамического типа объекта по данному ей указателю, который часто является указателем на базовый подобъект полного производного объекта. Для этого в иерархии полиморфных типов сам деструктор должен быть описан как виртуальный, иначе может быть вызван деструктор статического типа объекта, что приведёт к его неполному уничтожению, являющемся неопределённым поведением:

```
struct base1
{
};

struct derived1 : base1
{
};

struct base2
{
    // Нам достаточно деструктора по умолчанию,
    // главное - что он виртуальный.
    virtual ~base2() = default;
};
```

```
15
16  struct derived2 : base2 {};
17
18  void f()
19  {
20      base1* bp1 = new derived1;
21      // ОШИБКА: вызов base1::~base1 по статическому типу bp1,
22      //         хотя полный объект более производного типа.
23      delete bp1;
24
25      base2* bp2 = new derived2;
26      // Вызов derived2::~derived2 по таблице виртуальных функций
27      // исходя из динамического типа объекта.
28      delete bp2;
29  }
```

Некоторые источники рекомендуют всегда определять деструктор корня иерархии полиморфных классов как виртуальный. Автор считает, что занимать лишний слот в таблице виртуальных функций имеет смысл только, если такая иерархия предназначена для использования с передачей владения, т.к. именно в этом случае на практике возникает ситуация с необходимостью удаления объекта по динамическому типу. В примере с драйверами устройств, рассмотренном выше, например, подобное не понадобилось.

Запись нужного значения в указатель на таблицу виртуальных функций осуществляется конструкторами полиморфного класса автоматически сразу после создания всех базовых подобъектов и перед инициализацией не статических членов класса. Таким образом, для объекта в глубине цепочки наследований значение этого указателя обновляется всеми конструкторами базовых подобъектов, начиная с самого базового, по ходу конструирования объекта. По этой причине вызов виртуальных функций из конструкторов не рекомендуется, поскольку вызовет не последнее переопределение для конструируемого полного объекта, а только последнее переопределение для класса, из конструктора которого осуществляется вызов. Почти всегда это ошибочно, к счастью, хорошие трансляторы выдают в этом случае предупреждения — будьте бдительны, это может произойти не напрямую в конструкторе, а в одной из вызванных им функций. Цепочка деструкторов таких объектов производит обратные поэтапные изменения этого указателя, так что эта рекомендация справедлива и для них.

9.8.6. Чисто виртуальные функции и абстрактные классы

Часто встречается ситуация, когда в базовом классе не может быть осмысленного определения виртуальной функции, но соответствующее описание быть должно, чтобы его можно было переопределить в производных. Наш пример с блочными устройствами является именно такой ситуацией.

В таком случае можно воспользоваться определением ***чисто виртуальной функции (pure virtual function)***, которое имеет тело `= 0`. Полиморфный класс, в котором хотя бы одно последнее переопределение является чисто виртуальной функцией, называется ***абстрактным (abstract)***. Создание полных объектов абстрактных типов невозможно, но они могут существовать как базовые подобъекты других. Эта возможность позволяет пометить функции базового полиморфного класса как требующие переопределения.

Теперь мы можем переписать пример с блочными устройствами с использованием механизма виртуальных функций:

```
1   #include <cstddef>
2   #include <cstdint>
3
4   // Абстрактный базовый тип блочного устройства.
5   struct block_device
6   {
7       // Возвращает число блоков на устройстве.
8       virtual std::size_t block_count() = 0;
9       // Считывает данные указанного блока с устройства, возвращает успешность.
10      virtual bool read(void* data,std::size_t block) = 0;
11      // Записывает указанные данные в блок на устройстве, возвращает успешность.
12      virtual bool write(const void* data,std::size_t block) = 0;
13  };
14
```

```
15  class hdd : public block_device
16  {
17  public:
18      hdd(std::uint32_t id)
19      : id(id)
20      {
21      }
22
23      std::size_t block_count() override
24      {
25          std::size_t res = 0;
26          // Узнать размер диска с идентификатором id.
27          return res;
28      }
29
30      // Реализованы аналогично block_count.
31      bool read(void* data,std::size_t block) override;
32      bool write(const void* data,std::size_t block) override;
33  private:
34      // Дополнительные данные для работы с жёстким диском.
35      std::uint32_t id;
36  };
37
38  // Функции f и g идентичны предыдущему примеру.
```

В этом примере вся черновая работа по реализации динамического полиморфизма возложена на транслятор. Имеющиеся в языке C++ механизмы реализации динамического полиморфизма влияют только на выбор вызываемых функций, что также называется ***поздним связыванием (late binding)***.

В данном случае абстрактный базовый класс состоит исключительно из чисто виртуальных функций, т.е. является описанием интерфейса некоторого полиморфного класса без каких-либо деталей его реализации. В некоторых языках понятие «интерфейса» закреплено на уровне синтаксиса языка, а в C++ обычно используют подобную конструкцию.

9.8.7. Идентификатор со специальным значением final

После имени класса (и до двоеточия, если оно есть) может быть указан идентификатор со специальным значением `final`. Такой класс запрещено использовать в качестве базового.

```
1   class my_class final
2   {
3       // ...
4   };
5
6   // ОШИБКА: наследование от final класса.
7   class derived : public my_class
8   {
9       // ...
10  };
```

Этот идентификатор со специальным значением также может применяться на отдельных переопределениях виртуальных функций, включая исходные. Такое использование записывается в конце описания, как и `override` в любом порядке с ним, и запрещает последующие переопределения в производных классах:

```
1   struct base
2   {
3       virtual void f() {}
4   };
5
6   struct derived : base
7   {
8       // Последний (совсем) переопределитель виртуальной
9       // функции base::f().
10      void f() final override {}
```

```
};

struct derived2: derived
{
    // ОШИБКА: переопределения больше не возможны.
    void f() override {}
};
```

Подобные аннотации позволяют сделать компилятору дополнительные оптимизации: например, при вызове в показанном выше примере функции `(f()` на объекте статического типа `derived`, можно не обращаться к таблице виртуальных функций, а напрямую вызвать `derived::f`, как не виртуальную функцию, поскольку ни в каком из потенциально бесконечного количества производных от `derived` классов другого последнего переопределителя быть не может. Это, размеется, эффективнее.

Подобная замена вызова виртуальной функции на обычный вызов называется ***девиртуализацией (devirtualization)*** и является важной оптимизацией в программах, широко использующих позднее связывание. Компилятор во многих случаях может выполнить её и без дополнительных подсказок, например, при вызове виртуальной функции непосредственно на объекте, идентифицированным своим именем без использования указателей и ссылок, но в более сложных случаях даже полный анализ программы на этапе компоновки не сможет выявить некоторые случаи, которые могут быть явно указаны программистом с помощью этой возможности.

9.8.8. Множественное наследование

Класс может иметь несколько различных прямых базовых классов, перечисляемых через запятую. Наследование от двух или более классов называют ***множественным (multiple inheritance)***. В таком случае несколько прямых базовых классов, указываются в определении дочернего через запятую, указанный доступа задаётся для каждого из них независимо от остальных:

```
struct B1 {};
struct B2 {};

// D1 унаследован от B1 и B2,
// соответствующие подобъекты имеют уровни
// доступа открытый (указано явно) и закрытый (умолчание для class)
// соответственно.
class D1 : public B1,B2 {};

// ОШИБКА: несколько прямых базовых подобъектов одного типа не допускается.
struct D2 : B1,B1 {};
```

Такой класс имеет несколько членов-базовых подобъектов, которые инициализируются по порядку их указания в описании (и уничтожаются в обратном), так же как и для именованных подобъектов класса порядок инициализаторов значения не имеет:

```
// Продолжение предыдущего примера.

struct D3 : B1,B2
{
    int x,y;

    // Корректно, но порядок инициализации подобъектов
    // всё равно базовый типа B1, базовый типа B2, x, y.
    // Многие компиляторы дадут предупреждение.
    D3() : B2(),B1(),x(2),y(3) {}
};
```

При поиске имён в классе, унаследованном от нескольких, после просмотра его собственной области, искомое имя может быть найдено в любом из базовых подобъектов, входящих в дерево наследования. Говоря о поиске имён в иерархиях классов говорят о множествах описаний каждого класса, чтобы учесть возможность перегрузки функций — выбор конкретных перегрузок осуществляется только после выбора конкретного множества описаний. ***Основное***

правило скрытие при наследовании — множество описаний-членов класса, связанное с каждым подобъектом иерархии наследования, скрывает все описания, являющиеся членами других подобъектов, являющихся для данного базовыми. Помимо отсутствия описания требуемого имени, наличие более одного такого множества, не скрытого никаким другим является неоднозначностью выбора, т.е. ошибкой:

```
1  struct B1
2  {
3      void f();
4      void g();
5      void h();
6  };
7
8  struct B2
9  {
10     void f(int);
11     void g();
12     void h();
13 };
14
15 struct D1 : B1,B2
16 {
17     void g();
18
19     using B1::h;
20     void h(int);
21 };
22
23 struct D2
24 {
25     void test()
26     {
27         // ОШИБКА: множества описаний f в подобъектах типов B1 и B2
28         //         не скрыты ничем, и их два, а не одно.
29         //         (Хотя описаний, пригодных для вызова с одним аргументом,
30         //          в иерархии наследования ровно одно, до выбора конкретных
31         //          перегрузок дело не доходит.)
32         f();
33
34         // Вызов D1::g(). Множества описаний в B1 и B2 скрыты таковым в D1.
35         g();
36
37         // Вызов B1::h(). Множества описаний в B1 и B2 скрыты таковым в D1,
38         // состоящим из B1::h(), внесённого описанием using, и D1::h(int),
39         // разрешение перегрузок выбирает первое.
40         h();
41
42         // ОШИБКА: unknown не описано нигде (включая все классы в дереве
43         //         наследования: B1,B2,D1,D2).
44         unknown();
45     }
46 };
```

Концептуальное отношение, выражаемое множественным наследованием — объекты производного класса считаются частными случаями всех базовых, и включают в себя всех их характеристики.

Подобное отношение, когда все базовые классы относительно равноправны, встречается не часто, настолько, что не все языки, поддерживающие объектно-ориентированную парадигму, поддерживают полноценное множественное наследование. Обычно лишь один из базовых классов рассматривается как полноценное отношение is-a, а остальные вносят в класс дополнительный функционал, например:

```
1  // Класс, считающий число своих экземпляров во всей программе,
2  // может применяться для учёта общего числа "важных" объектов
3  // произвольных классовых типов.
4
```

```
5  class important_object
6  {
7  public:
8      static std::size_t count()
9      {
10         return count_;
11     }
12 protected:
13     // Объект не предназначен для создания,
14     // а только для наследования, поэтому
15     // защищённые конструктор/деструктор.
16
17     important_object()
18     {
19         ++count_;
20     }
21
22     ~important_object()
23     {
24         --count_;
25     }
26 private:
27     std::size_t count_;
28 };
29
30 std::size_t important_object::count_ = 0;
31
32 // Один из важных классов. Самому по себе ему не нужно
33 // ни от чего наследоваться, но для учёта количества объектов
34 // это типа одиночное наследование от important_object.
35 // Не будем считать это отношением is-a, потому наследование
36 // с закрытым уровнем доступа, это позволит сделать функцию
37 // count недоступной для пользователя это объекта, что
38 // могло бы сбить с толку - она считает не только объекты
39 // типа important_connection.
40
41 class important_connection : important_object
42 {
43     // ...
44 };
45
46 class base
47 {
48     // ...
49 };
50
51 // Другой важный класс. Ему требуется наследование от base,
52 // а также учёт его как важного, потому у него два базовых класса.
53 class derived : public base,important_object
54 {
55     // ...
56 };
```

Классы, не являющиеся самостоятельными, а лишь содержащими дополнительные части функциональности для добавления в другие классы наследованием, называют ***примесями (mixin)***.

Хотя несколько одинаковых прямых базовых классов не допускается, при множественном наследовании возможно несколько одинаковых базовых подобъектов, некоторые из которых не прямые:

```
1 struct B
2 {
3     int x;
4 };
5
6 struct D1 : B {};
```

```
7  struct D2 : B {};
8
9  struct D3 : D1,D2
10 {
11     // Поскольку D3 включает в себя подобъекты типа D1 и D2,
12     // а каждый их них - B, то в данном классе два базовых
13     // подобъекта типа B.
14
15     // ОШИБКА: неоднозначное использование имени.
16     x = 3;
17
18     // Ошибки нет, находится B::x из подобъекта D1.
19     D1::x = 3;
20 };
```

Как видно из данного примера, основное правило поиска имён при наследовании сформулировано в форме скрытия для подобъектов, а не самих классов именно для учёта неоднозначностей, когда подобъектов одного и того же класса может встретится несколько.

Показанная ситуация редко является желаемой — часто требуется, чтобы в подобной ситуации вместо нескольких копий подобъекта B имелся один, «общий» для всех частей класса D3. Для этого в языке C++ также имеется решение.

9.8.9. Виртуальное наследование

Для решения показанной выше проблемы с множественным наследованием, в языке C++ есть альтернативаня форма наследования, называемая ***виртуальным (virtual)*** наследованием. Оно имеет место, если в списке базовых подобъектов класса перед именем указано ключевое слово `virtual`.

В условиях наличия двух способов наследования, представление производного класса определяется следующим образом: ***производный класс содержит под одному подобъекту соответствующего типа на каждый случай его упоминания в качестве прямого базового без ключевого слова `virtual` в его полном дереве наследования, и один отдельный на все упоминания с ним.***

В случаях с виртуальным наследованием графическое его изображение имеет не ориентированные циклы, поэтому в более общем случае такую форму называют ***решёткой (lattice)*** наследования.

Считая, что в примере из предыдущей главы требовалось наличие всего одного базового подобъекта типа `B`, модифицируем его для этого:

```
1  struct B
2  {
3      int x;
4  };
5
6  // Виртуальное наследование от B.
7  struct D1 : virtual B {};
8  // Уровень доступа и virtual могут быть указаны в любом порядке.
9  struct D2 : public virtual B {};
10
11 struct D3 : D1,D2
12 {
13     // Все упоминания B в решётке наследования содержат virtual,
14     // поэтому им всем соответствует один и тот же подобъект.
15     // Поиск имён однозначен, т.к. выбора нет.
16
17     x = 3;
18 };
```

В случае с виртуальным наследованием правила поиска имён остаются теми же, хотя возникает один специфический случай:

```
1  struct B
2  {
3      void f();
4  };
```

```
5
6  struct D1 : B
7  {
8      void f();
9  };
10
11 struct D2 : B {};
12
13 struct D3 : D1,D2
14 {
15     void g()
16     {
17         // Вызов D1::f, которая скрывает B::f,
18         // хотя может казаться что по пути наследования
19         // D3->D2->B B::f не скрыта и конфликтует с ней.
20         // Понятия "путей" в правиле нет - скрытие
21         // осуществляется глобально, в рамках всей решётки наследования.
22         f();
23     }
24 }
```

Данное выше через понятие скрытия правило и предыдущий пример, который может казаться неожиданным в его рамках, были даны в стандарте C++03. Это правило иногда называют правилом ***доминантности (dominance)***. Начиная с C++11 поиск имён в классах формально выражается следующим эквивалентным алгоритмом поиска имени `f` в классе `C`.

Состояние поиска (lookup set) имени `f` в классе `C` состоит из множества описаний и множества подобъектов, в классах которых они были найдены, и обозначается $S(\texttt{f}, \texttt{C})$. Описания `using` в этом процессе при обработке заменяются описаниями, которым соответствуют.

1. Если сам класс `C` содержит описания `f`, то они и составляют множество описаний, множество подобъектов есть `C`, и алгоритм закончен.
2. Иначе оба множества изначально пусты и модифицируются следующими шагами алгоритма. Для каждого из прямых базовых классов `C` $\texttt{B}_i$ строится состояние поиска $S(\texttt{f}, \texttt{B}_i)$ по этому же алгоритму и объединяется с текущим состоянием поиска следующим образом:
   (a) Если $S(\texttt{f}, \texttt{B}_i)$ пусто, или каждый элемент его множества подобектов есть подобъект какого-либо из подобъектов текущего состояния, то текущее состояние не меняется.
   (b) Если верно обратное: каждый элемент множества подобъектов текущего состояния является подобъектом какого-либо элемента из множества подобъектов $S(\texttt{f}, \texttt{B}_i)$, то текущее состояние заменяется на $S(\texttt{f}, \texttt{B}_i)$.
   (c) Иначе в множество подобъектов $S(\texttt{f}, \texttt{B}_i)$ объединяется с множеством подобъектов текущего состояния. Если множества описаний текущего состояния и $S(\texttt{f}, \texttt{B}_i)$ не равны, множество описаний текущего состояния принимает специальное значение «ошибочное». Поиск имён может оказаться успешным и в этой ситуации, если один из следующих шагов заменит такое состояние корректным.

В конце алгоритма его результатом является множество описаний текущего состояния — если оно ошибочное, то это и есть ошибка.

Виртуальное наследование входит в разрез с известным нам правилом о том, что конструктор каждого класса инициализирует все свои прямые базовые подобъекты — в случае, рассмотренном выше по нему требуется инициализировать `B` два раза: конструктором `D1` и конструктором `D2`, чего не может быть по смыслу понятия «инициализация». C++ решает этот вопрос переносом обязанности инициализации виртуальных подобъектов на объекты самого производного типа, при этом инициализация того же подобъекта в остальных конструкторах подавляется. Общий порядок инициализации — для конструктора самого производного класса сначала все виртуальные базовые подобъекты в порядке обхода решётки наследования сначала в глубину, а затем по порядку указания прямых базовых подобъектов. Затем, для всех конструкторов, все прямые базовые подобъекты также в порядке указания — это также вызывает подобный обход для невиртуальных подобъектов, т.к. этот пункт рекурсивно выполняется вызванными конструкторами и для своих прямых не виртуальных базовых классов.

```
1  // Базовый класс объекта, имеющего имя.
2
3  class named_object
4  {
5  protected:
6      named_object(const char* name)
7          : name_(name)
8      {}
9  public:
10     const char* name() const
11     {
12         return name_;
13     }
14 private:
15     const char* name_;
16 };
17
18 // Именованный объект A.
19 // Не важно, сколько раз базовый класс именованного
20 // объекта встречается в решётке наследования, у объекта
21 // по смыслу нескольких имён быть не может, поэтому
22 // наследуемся виртуально.
23
24 class object_a : virtual public named_object
25 {
26 public:
27     object_a(const char* name,int a)
28         : named_object(name),a(a)
29     {}
30
31     // Полезная функциональность A.
32
33 private:
34     int a;
35 };
36
37 // Ещё один именованный подкласс.
38
39 class object_b : virtual public named_object
40 {
41 public:
42     object_b(const char* name,int b)
43         : named_object(name),b(b)
44     {}
45
46     // Полезная функциональность B.
47
48 private:
49     int b;
50 };
51
52 // Подкласс, объединяющий функциональность object_a и object_b.
53
54 class object_c : public object_a,public object_b
55 {
56 public:
57     object_c(const char* name,int a,int b,int c)
58         // 1. Все виртуальные базовые классы сначала вглубь, затем по порядку:
59         //    1.1 named_object
60         // 2. Не-виртуальные прямые базы:
61         //    2.1 object_a: виртуальный базовый named_object пропущен,
62         //        т.к. object_a не самый производный в object_c,
63         //        не виртуальных нет.
64         //    2.2 object_b: виртуальный базовый named_object пропущен,
65         //        т.к. object_b не самый производный в object_c.
66         //        не виртуальных нет.
```

```
67          : named_object(name),object_a(name,a),object_b(name,b),c(c)
68      {}
69
70      // Полезная функциональность C.
71
72  private:
73      int c;
74  };
```

9.8.10. Представления классов при наследовании

9.8.11. Операция приведения полиморфных типов dynamic_cast

Операция `static_cast` для указателей и ссылок на объекты классовых типов может осуществлять их приведение как вверх, так и вниз по иерархии, если в самой иерархии наследования имеется соответствующий путь от дочернего класса к родительскому или наоборот. Для случая приведения вниз она, правда, корректна только если имеющийся объект действительно является подобъектом требуемого производного типа, иначе наступает неопределённое поведение. Как следует из её названия, операция `static_cast` работает на основе статических типов объектов и не может предугадать наличие или отсутствие более полного объекта вокруг рассматриваемого.

Как было показано в прошлой главе, информация о динамическом типе объекта, тем не менее, в процессе работы программы содержится в её памяти в виде RTTI. Существует операция приведения типов, которая может ей воспользоваться — это `dynamic_cast`. Исходный и требуемый типы в ней должны оба быть ссылками или указателями на классовые типы. Для случаев приведения от производных классов к базовым, приведения к тому же типу (возможно, с добавлением квалификаторов) или приведения нулевого указателя, её поведение аналогично `static_cast`. В остальных случаях её операнд должен быть указателем на или обобщённым леводопустимым значением полиморфного классового типа, и операция осуществляет приведение с учётом RTTI во время работы программы. Приведение к нетипизированному указателю даёт адрес полного объекта, подобъектом которого является операнд (или он сам, если он и есть полный объект). Иначе проверяется, есть ли в полной иерархии классов, соответствующей полному объекту, в который входит операнд, однозначный доступный подобъект базового типа ссылки или указателя, к которому осуществляется приведение, и в случае успеха возвращается соответствующее значение.

Чаще всего `dynamic_cast` используется в форме приведения к указателю, в таком случае невозможность приведения сигнализируется возвратом нулевого указателя:

```
1   struct base
2   {
3       // dynamic_cast показывает свои отличия от static_cast
4       // только для полиморфных типов.
5       virtual ~base() = default;
6   };
7
8   struct derived : base {};
9
10  void f(const base& b)
11  {
12      // Стандартная форма проверки, является
13      // ли данный объект подобъектом требуемого:
14      if(auto pd = dynamic_cast<const derived*>(&b)){
15          // Да, является, pd - указатель на объект типа base,
16          // подобъектом которого является b.
17      }else
18          ; // иначе не является.
19  }
20
21  struct other_base
22  {
23      virtual ~other_base() = default;
24  };
25
26  // Множественное наследование.
```

```
27  struct other_derived : base,other_base {};
28
29  void g()
30  {
31      const base& b = other_derived();
32
33      if(auto pob = dynamic_cast<const other_base&>(&b)){
34          // Успешно!
35          // dynamic_cast рассматривает значение с точки зрения
36          // его полного динамического типа, поэтому может
37          // совершать приведения "поперёк" иерархии для
38          // случаев множественного наследования, которые
39          // через static_cast возможны только за счёт двойного
40          // применения, сначала вниз по иерархии, а затем
41          // вверх по другой ветке множественного наследования.
42          auto pob2 = static_cast<const other_base&>(
43                      static_cast<const other_derived&>(b));
44      }
45  }
```

«Нулевых» ссылок не бывает, и для случая приведения к ним используется другое средство индикации ошибки приведения типов во время выполнения программы, о котором будет рассказано в разделе «Исключения».

Использование `dynamic_cast` транслируется в вызов функции библиотеки поддержки языка C++, входящей в состав правильно установленного компилятора C++, и, поскольку включает разбор RTTI, занимает значительно больше времени, чем фиксированное смещение указателей, выполняемое `static_cast` в аналогичных случаях. Это следует учитывать при его использовании в местах кода, критичных к производительности. По этой причине некоторые библиотеки используют собственные средства для подобных целей, упрощённые для их случая и потому более быстрые. Например, можно ввести в иерархию классов виртуальную функцию, переопределения которой в каждом классе возвращают уникальные константы — подобный приём нам встретится далее при рассмотрении библиотеки Qt.

9.8.12. Задачи на наследование классов

Глава 10

Обобщённое программирование на языке C++

Чтобы устранить наиболее крупный из недостатков разрабатываемого в наших примерах класса `int_array`, потребуется познакомиться с ещё одной парадигмой языка C++. Этот класс работает с элементами типа `int`, однако в программах часто требуются динамические массивы из элементов различных типов, и дублировать полное определение такого класса для каждого из требуемых типов элементов кажется чрезвычайно неудобным.

На помощь приходят средства ***обобщённого (generic)*** программирования языка C++, которые в нём представлены в виде ***шаблонов (tempalte)***. Напомним, что основной идеей этой парадигмы программирования является обобщение алгоритмов и структур данных с отделением их от конкретных типов значений, которыми они оперируют. В нашем случае мы стремимся отделить алгоритмы обработки динамического массива от типа его элемента.

10.1. Шаблоны

Шаблон — специальная форма описания, задающая общий вид семейства описаний функции, классового типа, псевдонима типа или переменной. Существуют как определения, так и не являющиеся ими описания шаблонов, как и для соответствующих сущностей. Само описание, например, шаблона функции описанием функции не является, поскольку содержит в себе некоторое количество неизвестных параметров — ***параметров шаблона (template parameter)***. Шаблоны могут быть описаны только в областях видимости пространств имён и классов.

Описание шаблона содержит в себе описания всех параметров, в нём используемых. В качестве неизвестных сущностей могут выступать типы, значения заданного типа и другие шаблоны — такие параметры называют ***типовыми (type)***, ***не типовыми (non-type)*** и ***шаблонными (template)*** параметрами шаблона соответственно. Мы будем работать в первую очередь с типовыми параметрами шаблонов функций и классов, а остальные случаи рассмотрим по мере надобности.

Имея описание шаблона, можно задать значения всех его параметров, т.е. указать ***аргументы шаблона (template argument)***, и после их подстановки на место параметров шаблона из него получается уже настоящее описание функции, класса и т.д. Такую подстановку можно делать сколько угодно раз с разными аргументами шаблона, получая в результате различные ***специализации (specialization)***. Процесс формирования из описания шаблона конкретной сущности, которой он соответствует, называется ***инстанциацией (instantiation)*** (дословно, «созданием экземпляра»).

Рассмотрим эти понятия на примере. Напишем определение шаблона функции, возвращающей больший из двух её аргументов. В качестве типа её параметров могут выступать многие фундамантальные типы, а также, например, классовые типы, для которых имеется перегруженная операция `<`. Описание шаблона синтаксически является описанием соответствующей сущности, перед которым указывается ключевое слово `template`, а за ним в угловых скобках описания параметров шаблона (через запятую, если их несколько). Нам потребуется один типовой параметр, чтобы описать его как таковой, перед его идентификатором указывается ключевое слово `typename` или `class` (разницы нет, автор будет придерживаться первого варианта).

```
template<typename T>
T min(T x,T y)
{
    return x<y?x:y;
}
```

Это определение следует читать следующим образом: «семейство функций `min` имеет указанное определение, где под `T` понимается некоторый тип». `min` — это имя шаблона функции, но не какой-либо конкретной функции.

Автор будет использовать для имён параметров шаблонов идентификаторы без знаков подчёркивания, записывая каждое слово с большой буквы, например, `ParameterName`. Часто, когда параметров мало и их роль очевидна, применяют имена `T` и `U` (аналогично именам переменных-счётчиков `i` и `j`), как в приведённом выше примере. Этот стиль используется и в описаниях стандартной библиотеки языка C++.

Имена параметров шаблона видны в его описании для неквалифицированного поиска и не могут быть перекрыты в нём другими описаниями (включая имя самого шаблона). Имена параметров шаблонов, как и имена параметров функции, опциональны и могут не указываться. Параметры шаблонов могут иметь значения по умолчанию, синтаксис и поведение которых при наличии нескольких совместимых описаний практически совпадает с таковым для значений аргументов функции по умолчанию. Важным отличием является то, что в значениях параметров шаблона по умолчанию могут использоваться имена ранее описанных параметров.

Имя типового параметра шаблона (в нашем случае `T`) в его описании может выступать на месте спецификатора типа:

```
// Описание шаблона функции f с двумя типовыми параметрами шаблона:
// - T со значением по умолчанию int;
// - U со значением по умолчанию, совпадающим с T.
// Как спецификатор типа, T может быть элементом построения
// более сложных конструкций, например указателя на T в
// возвращаемом значении функции.
template<typename T = int,typename U = T>
T* f();

template<typename T,typename U>
// ОШИБКА: имя шаблона скрывает имя его параметра.
void U()
{
    // ОШИБКА: описание в шаблоне скрывает имя его параметра.
    int T;

    // T может использоваться везде, где требуется спецификатор типа.
    std::size_t x = sizeof(T);
}

// Определение шаблона, описание которого было дано ранее.
// Значения параметров шаблонов по умолчанию повторно не
// должны быть указаны, требование аналогично правилам
// задания значений аргументов функции по умолчанию.
template<typename T,typename U>
T* f()
{
    return nullptr;
}

// Описание шаблона функции с одним типовым параметром без имени.
template<typename>
void g();

// Описание шаблона функции с одним типовым параметром без имени,
// имеющим значение по умолчанию.
template<typename = double>
void h();
```

10.1.1. Концепции

Вернёмся к рассмотренному ранее примеру:

```
template<typename T>
T min(T x,T y)
{
    return x<y?x:y;
}
```

Хотя явно это нигде не указано, на тип `T` данный шаблон накладывает определённые ограничения, чтобы быть семантически и прагматически корректным:

- Значения типа `T` принимаются по значению в качестве аргументов и возвращаются из функции, поэтому `T` должен быть перемещаемым типом.
- Должна существовать встроенная или заданная программистом единственная лучшая перегрузка операции «меньше», которая может быть вызвана для двух леводопустимых операндов типа `T`, результат которой контекстно преобразуем к `bool`.
- Эта операция должна нести смысл сравнения, устанавливая некоторый порядок на множестве элементов типа `T` и удовлетворяя требованиям для таких операций, иначе название функции не будет отражать смысл возвращаемого ей результата.

Таким образом, за счёт использования параметров шаблона в тех или иных конструкциях внутри своего определения, шаблоны неявно накладывают на них ограничения по синтаксису конструкций, от них зависящих, и их семантике. Не проявилась в этом примере последняя третья, но не менее важная, часть требований — по максимальной алгоритмической сложности. Про типы, которые удовлетворяют формально заданным требованиям этих трёх видов, говорят, что они ***удовлетворяют (satisfy)*** соответствующей ***концепции (concept)***.

Таким образом, концепции являются переложением понятия абстрактных типов данных на утилитарную точку зрения конкретного языка C++. Это понятие в языке в настоящее время не имеет синтаксического проявления — подобную возможность планировалось ввести, но этой непростой задаче в современном стандарте языка пока не нашли приемлемого представления. Тем не менее, стандарт содержит формальные описания множества концепций, в терминах которых описывает требования многих шаблонов.

Концепции могут образовывать иерархии. В таком случае про концепцию, которая состоит из требований некоторой другой и добавляет свои дополнительные, говорят, что она ***уточняет (refine)*** другую.

С конкретными концепциями мы познакомимся, когда начнём рассмотрение шаблонов стандартной библиотеки языка, но предлагаем читателю держать эту идею в голове и в дальнейшем рассмотрении технических деталей.

10.1.2. Шаблоны функций и разрешение перегрузок

Чтобы воспользоваться шаблоном, необходимо выяснить значения всех его параметров. Начнём с описания алгоритма определения значений параметров шаблонов функций.

В одной и той же области видимости, как нам известно, может быть несколько перегруженных описаний функций с одним именем. Кроме этого, может присутствовать любое количество одноимённых шаблонов функций, которые тоже участвуют в процессе разрешения перегрузок. Чтобы иметь несколько перегруженных шаблонов, они должны различаться хотя бы по типу параметров функции, как для обычных функций, либо по списку параметров шаблона.

```
// Две перегруженные функции
void f();
void f(int);

// Шаблон, являющийся третьей перегрузкой имени f.
template<typename T>
int f(int,T);

// Четвёртая перегрузка отличается от третьей по
// списку параметров функции, хотя совпадает
// по списку параметров шаблона.
template<typename T>
double f(double,T);
```

```
14
15  // Пятая перегрузка отличается от третьей по списку
16  // параметров шаблона, хотя совпадает по списку параметров
17  // функции.
18  template<typename T,typename U>
19  int f(int,T);
20
21  // ОШИБКА: отличается от третьей перегрузки только типом
22  //         возвращаемого значения.
23  template<typename T>
24  char f(int,T);
```

При рассмотрении списка перегрузок функции, содержащего шаблоны, производится попытка выяснить однозначно значения параметров каждого шаблона. Для каждого шаблона, для которого этот процесс успешен, подстановка выясненных значений даёт конкретную специализацию функции, которая далее участвует в механизме перегрузок наравне с обычными функциями.

При вызове функции значения некоторых параметров могут быть указаны явно с помощью ***списка аргументов шаблона (template argument list)*** — списка значений параметров шаблона в угловых скобках через запятую, указываемого после имени самой вызываемой функции. Такой список становится частью имени вызываемой функции, поскольку позволяет идентифицировать конкретную перегрузку. Он может отсутствовать вообще или быть пустым (<>). Этот синтаксис распознаётся только если в точке его использования виден хотя бы один шаблон функции с указанным именем, иначе знак < трактуется как операция «меньше». Этот синтаксис является опциональной частью не квалифицированных имён.

После нахождения множества перегруженных функций и шаблонов функций, алгоритм разрешения перегрузок таков:

1. Если был указан список аргументов шаблона (даже пустой), все обычные функции (не шаблоны) признаются негодными. Производится попытка подставить указанные аргументы в параметры шаблона. Каждый шаблон, для которого это подстановка невозможна, признаётся негодным. Подстановка невозможна, если аргументов больше, чем параметров по числу, или если указанное значение не подходит для данного параметра: для типовых параметров должны быть указаны типы, для не типовых — значения требуемого типа, для шаблонных — шаблоны.

2. Для шаблонов функций, у которых не все значения параметров были указаны явно списком аргументов шаблона, начинается процесс ***дедукции (deduction)*** — автоматического выяснения значений параметров шаблона исходя из требуемых типов аргументов функции. Требуемые типы соответствуют фактическим типам аргументов функции в операции вызова, а также в других контекстах, где они известны (явное приведение к типу функции, присваивание адреса функции объекту, использование имени перегруженной функции в качестве аргумента другой функции и т.д.). Дедукция значений параметров шаблона является одним из основных проявлений механизма вывода типов в языке C++.

   Для каждого параметра функции, содержащего параметры шаблона, происходит попытка найти такие их значения, что при подстановке будет получен требуемый тип аргумента:

```
1   template<typename T,typename U>
2   void f(T (*)(U));
3
4   int g(double);
5
6   void h()
7   {
8       // Дедукция T (*)(U) из int (double) даёт (после разложения в указатель)
9       // T = int и U = double.
10      f(g);
11  }
```

   Из этого правила есть следующие исключения:

   - Если тип параметра функции — в точности один из типовых параметров шаблона, то дедукция выводит значение типа аргумента после разложения в нём массивов и функций в указатели и отбрасывания квалификаторов.

```
template<typename T,typename U>
void f(T,U);

void g()
{
    const int x = 1;
    int y[5] = {};
    // Выводит T = int, Y = int*.
    f(x,y);
}
```

- Если тип параметра функции является ссылкой на зависящий от параметров шаблона тип, разложения не происходит.

```
template<typename T,typename U>
void f(T&,const U&);

void g()
{
    int x[5] = {};
    volatile int y = 6;
    // Выводит T = int [5] и U = volatile int.
    // После подстановки тип первого параметра становится
    // int (&)[5] - ссылка на массив из 5 значений типа int,
    // а второго - const volatile int&.
    // Дедукция выводит тип, требуемый для корректной
    // передачи значения, добавляя квалификаторы по надобности
    // (но отнимать, как обычно, не может).
    f(x,y);
}
```

- Когда тип параметра функции является в точности праводопустимой ссылкой на типовой параметр шаблона без квалификаторов, такую ссылку неформально называют ***универсальной (universal)***. Механизм дедукции типов для универсальных ссылок проходит успешно для значений аргументов любой категории: для леводопустимых категорий выводится леводопустимая ссылка, для праводопустимых — сам тип выражения:

```
template<typename T,typename U>
void f(T&&,U&&);

void g()
{
    int x = 5;
    // Выводит T = int& и U = int.
    // При подстановке в результате коллапса ссылок
    // типы параметров становятся int& и int&&,
    // что позволяет успешно привязать аргументы
    // соответствующих категорий значений.
    // За эту способность эти ссылки и называют универсальными.
    f(x,6);
}
```

Если для нескольких параметров функции дедукция одного и того же параметра шаблона даёт разные значения, дедукция в целом заканчивается неуспешно и шаблон признаётся негодным.

Значения аргументов функции по умолчанию не участвуют в дедукции:

```
template<typename T>
void f(T x = 0);

void g()
{
    // ОШИБКА: T не может быть выведено.
    f();

    // T указано явно, и значение аргумента функции
    // по умолчанию используется.
    f<int>();
```

```
12
13      // Из типа аргумента выводится T = double,
14      // Значение аргумента по умолчанию не используется.
15      f(3.45);
16  }
```

3. Если после успешной дедукции остались параметры шаблона без выясненных значений, они берутся из их значений по умолчанию. Если хотя бы для одного из оставшихся параметров нет значения по умолчанию, этот шаблон не является годным.
4. После успешного выяснения значений всех параметров шаблона, они подставляются в описание функции для получения описания специализации, которая будет участвовать на месте шаблона в основном процессе разрешения перегрузок. Если в результате такой подстановки образуется некорректная конструкция, шаблон удаляется из списка годных.

```
1   template<typename T>
2   T f(T&);
3
4   void g()
5   {
6       int x[5];
7       // ОШИБКА: единственная перегрузка-шаблон не годна -
8       //         дедукция даёт T = int [5], что после
9       //         подстановки в описание приводит к тому,
10      //         что возвращаемым значением функции
11      //         становится массив, что недопустимо.
12      f(x);
13  }
```

5. После формирования специализаций из всех годных шаблонов, запускается обычный механизм разрешения перегрузок.
6. Если обычный механизм выбора перегрузок не может выбрать лучшую из нескольких годных функций, среди которых присутствуют специализации шаблонов, применяются дополнительные правила, которые могут устранить неопределённость:

   - Специализация шаблона считается хуже обычной функции с тем же описанием:

```
1   // Обычная функция
2   void f(int);
3
4   // Шаблон
5   template<typename T>
6   void f(T);
7
8   void g()
9   {
10      // Годны обе перегрузки. Дедукция для шаблона
11      // выводит T = int, что при подстановке даёт
12      // void f(int), что идентично описанию обычной
13      // функции. Несмотря на идентичность описаний,
14      // обычная функция и специализация шаблона -
15      // различные функции, и лучшей считается первая.
16      f(5);
17  }
```

   - Если имеется неопределённость между несколькими специализациями шаблонов, одна из них может быть лучше всех других, если является ***наиболее специализированной (most specialized)***, чем все остальные. Одна из двух специализаций считается более специализированной, если по всем параметрам функции, зависящим от параметров шаблона, она не менее специализирована, а хотя бы по одному — более специализирована, чем другая. Чтобы выяснить, какой тип в каждой паре соответствующих параметров является более специализированным, применяется следующая процедура:

     (a) Каждый из параметров, являющийся ссылкой, заменяется её базовым типом. После этого квалификаторы верхнего уровня отбрасываются.

     (b) Для всех параметров шаблона, использованных в обоих типах, придумываются фиктивные значения: для типовых — типы, для не типовых — значения требуемого типа, для шаблонных — шаблоны.

(c) Производится попытка вывести с помощью дедукции значения параметров первого типа при подстановке в него второго типа с подставленными фиктивными параметрами, а затем наоборот. Если только одна из этих дедукций успешна, более специализированным является тип, дедукция параметров которого оказалась неудачной:

```
template<typename T>
void f(T);

template<typename T>
void f(T*);

void g()
{
    int x = 5;
    // Годны оба шаблона, дедукция в первом выводит T = int*,
    // во втором - T = int. После подстановки обе специализации
    // имеют одинаковый вид void f(int*), не различимый обычным
    // механизмом разрешения перегрузок. Глубина специализации
    // двух шаблонов будет определяться по их единственному параметру,
    // T против T*.
    // Направление 1: в первой функции используется один типовой параметр,
    // который заменяем фиктивным типом SomeType, после подстановки в описание
    // получим SomeType. Попытка вывести значение параметра шаблона T из
    // параметра вида T* второй функции из SomeType неудачна - неизвестный тип
    // в общем случае не указатель.
    // Направление 2: во второй функции используется один типовой параметр,
    // который заменяем фиктивным типом SomeType, после подстановки в описание
    // получим SomeType*. Попытка вывести значение параметра шаблона T из
    // параметра вида T первой функции из SomeType* даёт T = SomeType*.
    // Поскольку только одно направление удачно, T* во втором шаблоне
    // является более специализированным типом, чем T в первом, а значит и весь
    // второй шаблон более специализирован чем первый и выбирается в качестве
    // результата разрешения перегрузок.
    f(&x);
}
```

(d) Если дедукция успешна в обоих случаях, но один из параметров изначально был леводопустимой ссылкой, а другой - нет, то более специализирован первый из них.

```
template<typename T>
void f(T);

template<typename T>
void f(T&);

void g()
{
    int x;
    // После отбрасывания ссылок получаются идентичные типы T,
    // так что дедукция успешна в оба направления.
    // До отбрасывания T& был леводопустимой ссылкой, а T - нет,
    // так что вторая перегрузка более специализирована.
    f(x);
}
```

(e) Иначе более специализированным является более квалифицированный тип (до отбрасывания):

```
template<typename T>
void f(T&);

template<typename T>
void f(const T&);

void g()
{
    const int x = 5;
    // Дедукция параметров шаблона даёт в первом шаблоне T = const int,
    // во втором - T = int, специализации void f(const int&) идентичны.
```

```
12      // Дедукция успешна в обоих случаях, оба типа не отличаются по леводопустимости
13      // Второй более квалифицирован: const против пустого множества квалификаторов,
14      // так что вторая перегрузка более специализирована.
15      f(x);
16  }
```

(f) Иначе ни один из типов не является более специализированным, чем другой.

Этот набор правил называют ***частичным порядком шаблонов функций (partial ordering of function templates)***, частичным в том смысле, что он может различить по глубине специализации не любую пару шаблонов. Часто вместо формальных правил достаточно определения на глаз: более специализирован тот тип, который имеет «более сложное устройство».

Отметим, что если при подстановке в процессе выявление параметров шаблонов функций в её описании образуется недопустимая конструкция, это не является само по себе ошибкой в программе, а всего лишь удаляет данный шаблон из множества годных перегрузок. Этот принцип носит название ***неудача подстановки не является ошибкой (Substitution Failure Is Not An Error, SFINAE)*** и умышленно эксплуатируется в метапрограммировании с использованием шаблонов языка C++. Однако после того, как механизм выбора перегрузок выбрал специализацию шаблона в качестве требуемой функции, проблемы, возникающие при подстановке вычисленных значений параметров шаблона в её определение уже являются настоящими ошибками:

```
1  tempalte<typename T>
2  void f(T& x)
3  {
4      ++x;
5  }
6
7  void g()
8  {
9      struct S { int x; };
10     S s;
11     // Выбирается единственная перегрузка f с T = S,
12     // но в её теле возникает ошибка - для S нет
13     // перегруженной операции инкремента.
14     f(s);
15 }
```

При диагностировании подобных ошибок компилятором сообщение об ошибке может быть достаточно громоздким, поскольку в нём будет указана вся цепочка выбранных специализаций и значений их параметров, которая привела к созданию по шаблону ошибочного кода.

Правила дедукции не ограничиваются перечисленными выше, где автор постарался разобрать большинство наиболее часто встречаемых случаев. Например, в некоторых случаях могут возникать ***не дедуцируемые контексты (non-deduced context)*** — комбинации параметров функции, содержащие параметры шаблона, и аргументов, которые не влияют на результат дедукции: не приводят к её ошибке, если значения параметров определяются без их учёта однозначно, но и сами не предоставляют возможности для дедукции. Одним из примеров такого контекста является сопоставление аргумента типа функция параметру, являющемуся адресом или ссылкой на функцию, когда ни одна или несколько перегрузок, связанных с именем аргумента подходят, или среди них есть шаблоны:

```
1  template<typename T>
2  void f(T);
3
4  template<typename T>
5  void g(void (*)(T),T);
6
7  void h()
8  {
9      // Среди перегрузок f есть шаблоны,
10     // поэтому тип первого параметра шаблона g
11     // становится не дедуцируемым контекстом.
```

```
12      // Это не является ошибкой. Из второго параметра
13      // выводится T = int, после подстановки которого
14      // в первый параметр выбор требуемой перегрузки
15      // (специализации f с T = int) становится однозначным.
16      g(f,5);
17  }
```

Более сложные случаи читателю предлагается разобрать с использованием дополнительной литературы.

10.1.3. Шаблоны классов

Шаблоны классов задают семейство классов, т.е. семейство общих по внутреннему устройству структур данных и связанных с ними алгоритмов их обработки.

К шаблонам не применим механизм дедукции типов параметров шаблона. Значения всех их параметров должны быть заданы либо явно в списке аргументов шаблона, либо значениями по умолчанию, поэтому после параметров со значениями по умолчанию не могут идти параметры без них (что также справедливо для параметров функций, но не для параметров шаблонов функций, где следующие параметры без значений по умолчанию могут получить значения в результате дедукции). При именовании специализаций классов использование списка аргументов шаблона обязательно, даже если он пустой.

```
1   // Шаблон класса с двумя типовыми параметрами,
2   // у обоих есть значения по умолчанию.
3   template<typename T = double,typename U = int>
4   struct S
5   {
6       // Поскольку типовые параметры используются
7       // в нестатических членах данных, разные
8       // специализации этой структуры устроены
9       // в памяти по разному.
10      T x;
11      U* y;
12
13      // Поскольку это функция-член класса-шаблона,
14      // каждая специализация класса имеет свою
15      // функцию-член f.
16      void f()
17      {
18          x += *y;
19      }
20  };
21
22  void g()
23  {
24      int x = 5;
25      // При именовании специализации класса
26      // список аргументов шаблона обязателен,
27      // даже если все его параметры имеют
28      // значения по умолчанию.
29      S<> s = {42.0,&x};
30      s.f();
31  }
```

Когда функция-член шаблона класса определяется вне него в окружающей области видимости, её определение должно быть квалифицировано именем содержащего её класса. Поскольку каждая специализация класса содержит свою «версию» этой функции, в её квалифицированном имени потребуется указать все параметры, с которыми инстанциируется содержащий её класс:

```
1   // Шаблон класса
2   template<typename T>
3   class C
4   {
5       // Описание функции-члена шаблона класса
6       void f();
```

```
}

// Это определение функции-члена f шаблона класса
// C, имеющего один типовой параметр T, вне его тела.
template<typename T>
void C<T>::f()
{
    // ...
}
```

Поскольку в классовой области видимости можно описывать другие шаблоны, могут существовать шаблоны-члены шаблонов классов. При их определении вне класса, на каждый уровень шаблона, содержащего вложенный, указывается своя «шапка» из ключевого слова `template` и списка параметров:

```
template<typename T>
class C1
{
    // Определение шаблона функции-члена
    // в теле содержащего её шаблона-класса.
    template<typename U>
    void f()
    {

    }

    // Описание вложенного шаблона класса.
    template<typename U>
    class C2;

    // Описание вложенного шаблона функции.
    template<typename U>
    void g(U x);
};

// Первый список параметров шаблона нужен,
// чтобы указать, что описываемый класс -
// член шаблона класса C1 с одним типовым
// параметром T. Второй список является списком
// параметров самого определяемого класса C2.
template<typename T> template<typename U>
class C1<T>::C2
{

};

// Определение шаблона-члена функции вне
// содержащего её шаблона класса.
template<typename T> template<typename U>
void C1<T>::g(U x)
{
    // ...
}
```

Внедрённое имя класса в шаблоне класса играет двойную роль: при его использовании со списком аргументов шаблона оно является именем шаблона, а при использовании без списка — либо именем шаблона, либо именем его специализации со значениями всех его параметров, какими они были определены в данной специализации, в качестве аргументов, т.е. той специализации, о которой в данный момент идёт речь, так называемой ***текущей специализации (current specialization)***.

```
template<typename T>
struct S
{
    // Возвращаемое значение - произвольная специализация того же шаблона,
    // по построению никогда не совпадающая с текущей.
```

```
6       S<T*> f1()
7       {
8           return {};
9       }
10
11      // Специализация текущего шаблона со значениями его параметров,
12      // соответствующим данной специализации - явная форма текущей специализации.
13      S<T> f2()
14      {
15          return {};
16      }
17
18      // В возвращаемом значении функции ожидается имя типа, в данном
19      // случае это текущая специализация, сокращённая форма записи предыдущего примера.
20      S f3()
21      {
22          return {};
23      }
24  };
```

Рассмотрим взаимодействие шаблонов со специальными и другими необычными функциями-членами класса. Конструкторы могут быть шаблонами, однако поскольку они вызываются без использования обычного синтаксиса вызова функции, указать явный список аргументов шаблона для них невозможно — аргументы в угловых скобках после имени шаблона класса являются аргументами самого шаблона класса, а не его конструктора. Таким образом, все параметры шаблонов конструкторов должны быть дедуцируемы или иметь значения по умолчанию. Деструкторы не могут быть шаблонами, т.к. для них это бессмысленно — при уничтожении экземпляра класса нет никаких параметров, позволивших бы выбрать один деструктор из нескольких. Поскольку те или иные, включая удалённые, описания специальных функций перемещения и копирования даются всегда, явно или неявно, и они не являются шаблонами, они будут выигрывать разрешение перегрузок у любых шаблонов, даже если разрешение перегрузок даёт от них специализацию, по описанию соответствующую описанию такой специальной функции. Операции преобразования также могут быть шаблонами, для них вместо типа параметров функции производится дедукция самого типа, приведение к которому они осуществляют.

Друзья классов и шаблоны могут комбинироваться во всех сочетаниях, правда, соответствующий синтаксис несколько запутан. Основной проблемой является то, что при указании одной функции в качестве друга класса, когда её описание содержит параметры шаблона класса, наличие или отсутствие списка параметров функции определяет, идёт ли речь о специализации некоторого шаблона или об обычной функции. Рассмотрим возможные ситуации:

```
1   // Требуется описать шаблон функции до его использования в
2   // описании друга в классе, чтобы список аргументов шаблона
3   // был распознан как таковой.
4   template<typename T>
5   void f3(T);
6
7   // Требуется описать шаблон класса до его использования
8   // в описании друга в классе, чтобы список аргументов шаблона
9   // был распознан как таковой.
10  template<typename U>
11  class D;
12
13  template<typename T>
14  class C
15  {
16      // Функция f1 - друг всех специализаций шаблона класса C.
17      friend void f1();
18
19      // У каждой специализации шаблона класса C
20      // есть друг-функция f2, описание которой зависит
21      // от T, и это не специализация другого шаблона,
```

```
    // а отдельная функция.
    friend void f2(T);

    // У каждой специализации шаблона класса C
    // есть друг-функция f3, описание которой зависит
    // от T, и это специализация другого шаблона,
    // а не отдельная функция. Мы могли не указывать
    // значение параметра в списке аргументов шаблона, если
    // он дедуцируем из списка параметров функции, но
    // хотя бы пустой список необходимо оставить, чтобы
    // показать, что друг - специализация шаблона, а
    // не обычная функция. Если таких шаблонов несколько,
    // может потребоваться разрешение перегрузок.
    friend void f3<T>(T);

    // Все специализации шаблона функции f - друзья
    // каждой специализации шаблона класса C.
    template<typename U>
    friend void f(U);

    // Все специализации C имеют друзьями две специализации
    // класса D.
    friend class D<int>;
    friend class D<T*>;

    // Все специализации D - друзья каждой специализации C.
    template<typename U>
    friend class D;
};
```

Объявление функций-членов друзьями, а также друзей-шаблонов и их специализаций обычных классов производится аналогично.

Чаще всего взаимодействие друзей и шаблонов наблюдается в перегрузках операций над шаблонами классов. Покажем требуемую последовательность описаний для объявления требуемой специализации шаблона перегруженной операции сложения другом шаблона класса:

```
// Описание шаблона класса, чтобы его можно было
// использовать в следующем описании.
template<typename T>
class C;

// Описание шаблона перегруженной для всех
// специализаций шаблона класса C операции сложения,
// чтобы в описании друга в классе распознавался
// список аргументов шаблона, чтобы показать, что
// друг - специализация шаблона.
template<typename T>
C<T> operator+(const C<T>& x,const C<T>& y);

template<typename T>
class C
{
    // ...

    // Описание конкретной специализации (показано
    // наличием списка аргументов шаблона) в качестве
    // друга класса. Поскольку внедрённое имя класса
    // может именовать текущую специализацию, указание
    // списка аргументов шаблона после C не требуется.
    friend C operator+<>(const C&,const C&);
};

// Определение перегруженной операции сложения для
// каждой специализации C в виде свободной функции.
```

```
template<typename T>
C<T> operator+(const C<T>& x,const C<T>& y)
{
    // ...
}
```

Чтобы упростить эту последовательность описаний, можно воспользоваться тем фактом, что описания функций-друзей в не локальных классах могут быть определениями. Такие функции, как и другие друзья, сами не являются членами класса, однако, поскольку определены в нём, имеют особый статус: единственный способ найти их имя — аргументо-зависимый поиск имён, они невидимы для обычного квалифицированного или не квалифицированного поиска имён в окружающем класс пространстве имён. Поскольку они — друзья класса, они имеют доступ к закрытым членам класса. Как и другие определения функций в теле класса, они неявно встраиваемы. Таким образом, пример выше можно сократить до:

```
template<typename T>
class C
{
    // У каждой специализации класса C есть друг -
    // обычная функция-перегруженная операция для
    // конкретного значения параметра T.
    // Она может быть найдена только аргументо-зависимым
    // поиском имён (что обычно и нужно для перегруженный операций).
    friend C operator+(const C&,const C&)
    {
        // ...
    }
};
```

В классах также допустимы определения шаблонов функций-друзей.

Теперь у читателя достаточно знаний, чтобы превратить определение класса `int_array` в шаблон `array`, работающий с любыми тривиальными типами данных — по сути требуется только добавить описание параметра шаблона, да заменить использование имени типа `int` в определении класса на имя типового параметра. Могут потребоваться и другие мелкие изменения, например замена значения 0, как значения инициализированных по значению объектов, явным видом такой формы инициализации `T{}`. Для проверки тривиальности типа-параметра шаблона можно использовать утверждение времени компиляции и метафункцию `std::is_trivial_v`.

10.1.4. Неявная и явная инстанциация шаблонов. Шаблоны и заголовочные файлы.

Как нам известно, инстанциация — процесс создания специализации шаблона по заданным его параметрам с получением сущности того типа, шаблон которой использовался. Этот процесс происходит только один раз для данного шаблона и значений параметров, в последующие разы, когда эта специализация нужна, используется уже существующая.

Все рассмотренные нами примеры пользовались ***неявной инстанциацией (implicit instantiation)*** — инстанциацией специализации в момент, когда она требуется. Для шаблонов функций инстанциация их описания происходит в процессе разрешения перегрузок, а определений — при вызове функции или взятии её адреса. Для шаблона класса неявная инстанциация происходит тогда, когда требуется полнота определённой инстанциации. При неявной инстанциации класса инстанциируются описания, но не определения всех его членов, не являющихся шаблонами. Определения остальных членов класса могут быть инстанциированы позднее обычным образом. Это позволяет иметь в классах, например, члены-функции, определения которых не корректны для всех значений параметров, с которыми корректны все неявно инстанциируемые части класса. Такую функцию можно вызывать только у специализаций класса, в которых её определение корректно, однако само наличие её в шаблоне класса не мешает его инстанциации с параметрами, для которых его тело некорректно:

```
template<typename T>
struct C
{
```

```
4       T x;
5
6       void f()
7       {
8           ++x;
9       }
10  };
11
12  void g()
13  {
14      // Неявная инстанциация C<int>,
15      // инстанциируются описания x и f.
16      C<int> c1;
17      c1.x = 42;
18      // Инстанциируется определение C<int>::f.
19      c1.f();
20      // Используется уже существующая инстанциация C<int>.
21      C<int> c2;
22      // Ничего не инстанциируется, для описания указателя
23      // полнота базового типа не требуется.
24      C<char>* c3 = nullptr;
25      // Неявная инстанциация C<int [5]> возможна.
26      // Хотя тело f некорректно для T = int [5],
27      // при этом инстанциируется только её описание
28      // void f(), которое корректно.
29      C<int [5]> c4;
30      c4.x[2] = 53;
31      // ОШИБКА: инстанциируется определение C<int [5]>::f,
32      //         которое ошибочно (операции инкремента не
33      //         применима к значению типу категории массив или не
34      //         леводопустимому значению, в которое оно разлагается).
35      c4.f();
36  }
```

Поскольку для инстанциации определения сущности всем распространённым трансляторам языка C++ требуется вначале обработать определение соответствующего шаблона, использование шаблонов в интерфейсах единиц трансляции следует тем же принципам, что и встраиваемых функций: в заголовочном файле размещают не просто описания, а определения шаблонов, необходимых другим единицам трансляции. При этом часто определяют даже не встраиваемые функции-члены в телах классов, чтобы избежать громоздкого синтаксиса их определений в окружающем пространстве имён — современные компиляторы не будут встраивать функции там, где это не выгодно.

В крайнем случае, когда вся полезная составляющая интерфейса единицы трансляции состоит из шаблонов, самой единицы трансляции, парной к заголовочному файлу, может и не быть вообще. Эта ситуация часто наблюдается в самой стандартной библиотеке языка C++, выросшей из стандартной библиотеки шаблонов.

Шаблоны и их члены, как и встраиваемые функции в C++, при использовании внешней связанности обрабатываются особым образом на уровне символов объектных файлов, т.е. факт наличия их определений в нескольких единицах трансляции не является ошибкой компоновки, компоновщик выбирает из всех идентичных определений одно любое, а остальные выбрасывает. Затраченное впустую на компиляцию выброшенных инстанциаций время является неприятным побочным эффектом использования шаблонов в языке C++ в современных реализациях.

> Помимо неявной инстанциации при первом использовании, существует возможность затребовать инстанциацию определения любой указанной программистом специализации с помощью специальной формы описания — ***определения явной инстанциации (explicit instantiation definition)***. Оно предписывает инстанциировать определение именованной в нём специализации, если это ещё не было сделано. Для классов явная инстанциация, в отличие от неявной, приводит к инстанциации определений всех его членов, не являющихся шаблонами.
>
> Обратной по значению к этому описанию является ***описание явной инстанциации (explicit instantiation declaration)***. Это предписание транслятору не инстанциировать указанную специализацию, а генерировать на неё в объектном файле ссылку как на символ, для которого есть только описание, но не определение (даже если в этом месте

видно определение шаблона). Использовать описание явной инстанциации в единице трансляции после того, как в ней уже была явно или неявно инстанциирована указанная специализация, нельзя.

Синтаксически определение явной инстанциации состоит из ключевого слова `template`, за которым следует полное имя специализации: для классов оно должно быть детальным именем, для функций — описанием функции. Параметры шаблона должны быть указаны явным списком аргументов шаблона, для функций возможно применение дедукции, чтобы опустить или сократить его. Описание явной инстанциации также начинается с ключевого слова `extern`. Проверки на доступ к членам класса в таких описаниях не производится, поскольку речь не идёт о доступе к ним.

Используя эти конструкции, можно избежать множественной компиляции наиболее часто используемых инстанциаций шаблона, воспользовавшись следующей схемой:

```
// util.hpp - заголовочный файл

#ifndef UUID_AA919A53_5008_4A60_BB96_485C9A95EEFF
#define UUID_AA919A53_5008_4A60_BB96_485C9A95EEFF

// Определение шаблона класса для использования
// другими единицами трансляции.
template<typename T>
class C
{
    // ...
};

// Наиболее часто используемые специализации - это
// C<int> и C<double>, дадим для них описания явной
// инстанциации, чтобы они не компилировались в
// каждой единице трансляции, где используются, заново.

extern template class C<int>;
extern template class C<double>;

#endif
```

```
// util.cpp - единица трансляции

// Включим определение шаблона
#include "util.hpp"

// В этой единице трансляции будут содержаться
// все символы наиболее часто используемых инстанциаций
// C<int> и C<double>, которые будут использованы
// другими единицами трансляции за счёт внешней связанности.
// Затребуем их инстанциацию здесь явно.

template class C<int>;
template class C<double>;
```

Этот приём удаётся применить только, если число часто используемых инстанциаций не велико, и имеет смысл только в очень крупных проектах, где выигрыш в скорости компиляции будет заметен.

Формально, стандарт языка не предписывает такой порядок инстанциации, при котором по умолчанию многие специализации генерируются многократно в нескольких единицах трансляции. По сравнению с фазами трансляции программы на языке C, между получением единиц трансляции из файлов исходного текста и компоновкой выполняется ещё одна фаза. После создания единицы трансляции, требуемые в ней специализации (явно или нет) лишь отмечены, как требуемые, но не инстанциированы. На следующем этапе определяется местанахождение всех определений требуемых инстанциаций, происходит сама инстанциация, и единицы трансляции превращаются в ***единицы инстанциации (instantiation units)*** — единицы трансляции, не содержащие ссылок на не инстанци-

ированные специализации и не содержащие никакой информации о самих шаблонах. Единицы трансляции затем участвуют в процессе компоновки.

Таким образом, требование к видимости текста определения шаблона в месте, где его требуется инстанциировать, является лишь деталью современных реализаций языка C++. В прошлом была попытка к изменению этой модели, завершившаяся неудачей, артефактом которой является наличие в языке не используемого, но зарезервированного ключевого слова `export`. Автор предполагает, что благодаря развитию технологий ***оптимизации программ, совмещённых с компоновкой (Link-Time Optimization, LTO)***, основанных на отказе от раздельной обработки компилятором единиц трансляции, а также введению в язык средств поддержки модульного программирования, которые в настоящее время находятся в разработке, эта проблема, отпугивающая многих программистов от использования шаблонов и иногда даже языка в целом, рано или поздно получит эффективное решение.

10.1.5. Явная специализация шаблонов

Пока нам известен только один способ получения специализации шаблона — подстановка значений параметров в сам шаблон. В языке C++ также можно задать явный вид любой конкретной специализации без обращения к механизму инстанциации. Такие описания и определения называют ***явной специализацией (explicit specialization)***. Синтаксически это описания (определения), в которых «шапка» параметров шаблона содержит пустой список параметров, показывая, что это описание относится к шаблону, но все его параметры уже зафиксированы. Их значения указываются в списке аргументов шаблона после имени описываемой сущности, а для функций также могут быть дедуцированы. Явные специализации не могут быть описаны после того, как соответствующая специализация уже была инстанциирована обычным образом или указана в описании явной инстанциации. Иначе они используются напрямую, когда требуется соответствующая специализация, не вызывая процесса инстанциации:

```
1  // Шаблон класса C с одним типовым параметром T.
2  template<typename T>
3  struct C
4  {
5      T x;
6  };
7
8  // Явная специализация шаблона класса C для T = int.
9  // в ней x - не объект типа T, а массив из двух int.
10 template<>
11 struct C<int>
12 {
13     int x[2];
14 }
15
16 void f()
17 {
18     // Явной специализации C<double> нет,
19     // происходит инстанциация из шаблона.
20     C<double> c1;
21     c1.x = 12.3;
22     // Инстанциации из шаблона не происходит, используется
23     // определённая выше явная специализация C<int>.
24     C<int> c2;
25     c2.x[0] = c2.x[1] = 42;
26 }
```

Явные специализации позволяют, например, создавать для конкретных значений параметров шаблона отдельные оптимизированные реализации, оставляя для пользователей интерфейс в виде универсального шаблона. Для классов они могут быть даны в пространстве имён, окружающем то, в котором дано определение самого шаблона, с соответствующим образом квалифицированным именем.

Выше явная специализация, которую иногда называют полной (заданы явно значения всех параметров) показана на примере класса, поскольку, хотя шаблоны функций можно явно

специализировать, это ведёт к путанице, поскольку они, как любые специализации шаблонов функций, отличны от обычных функций с тем же описанием:

```
1  // Описание шаблона функции.
2  template<typename T>
3  void f(T);
4
5  // Явная специализация шаблона выше для T = int,
6  // значение этого параметра в этом описании
7  // выводится с помощью дедукции. У него мог также
8  // быть указан явный список аргументов шаблона в виде <> или <int>.
9  template<>
10 void f(int)
11 {
12     std::cout << "Specialization!\n";
13 }
14
15 // Определение обычной функции, перегружающей шаблон,
16 // и имеющей одинаковое с его специализацией для T = int,
17 // явно заданной, определение, но тем не менее, это
18 // отдельная от неё функция.
19 void f(int)
20 {
21     std::cout << "Ordinary function!\n";
22 }
23
24 void g()
25 {
26     // Вызывает обычную функцию, т.к.
27     // годны обе, но обычная функция
28     // лучше специализации при перегрузке.
29     f(5);
30
31     // Вызывает явную специализацию, поскольку
32     // наличие даже пустого списка аргументов шаблона
33     // сразу устраняет все обычные функции из списка перегрузок.
34     f<>(5);
35 }
```

Разница в том, какая функция вызывается в примере выше не всегда очевидна. По этой причине рекомендуется вместо явных специализаций функций использовать дополнительные перегрузки в виде обычных функций. Исключением из этого совета являются случаи, в которых планируется указывать список аргументов шаблона явно, что не позволит найти такую перегрузку.

10.1.6. Частичная специализация шаблонов классов

Шаблоны позволяют задать параметризованное семейство сущностей, а явные специализации — уточнить поведение для конкретных наборов параметров. Как поступить, если особый вид необходимо задать для случаев, когда только некоторые из нескольких параметров шаблона зафиксированы?

Для функций этого можно добиться введением дополнительных перегрузок шаблонов:

```
1  // 1. Перегрузка для типов параметров int и int.
2  void f(int x,int y)
3  {
4      std::cout << "int,int\n";
5  }
6
7  // 2. Перегрузка для типов параметров любой не int и int.
8  template<typename T>
9  void f(T x,int y)
10 {
11     std::cout << "T,int\n";
12 }
13
```

```
14  // 3. Перегрузка для двух любых параметров, когда второй не int.
15  template<typename T,typename U>
16  void f(T x,U y)
17  {
18      std::cout << "T,U\n";
19  }
20
21  int g()
22  {
23      // Годны и идентичны все,
24      // но 1 - не специализация и потому лучше
25      // двух других специализаций.
26      f(3,4);
27      // Годны все, но для 1 требуется преобразование
28      // для второго параметра, а для двух других - нет.
29      // На этом обычное разрешение перегрузок заканчивается -
30      // невозможно различить 2 и 3. Частичный порядок шаблонов
31      // даёт для второго аргумента: T выводится из int, но int
32      // нельзя вывести из T, поэтому 2 более специализированный
33      // шаблон, чем 3, и вызывается.
34      f(3.5,4);
35      // Обычный механизм разрешения перегрузок выбирает 3,
36      // т.к. первым двум требуется конверсия второго аргумента к int.
37      f(3.5,4.5);
38  }
```

Для классов это решение не подходит, поскольку для них понятия перегрузки не существует. Вместо этого для классов допускается задание ***частичных специализаций (partial specialization)***. Это описания, в которых после имени шаблона указан список аргументов шаблона более специализированного вида, чем просто перечисление по порядку всех параметров исходного шаблона, который в таком случае называют ***первичным (primary)*** и отличают от специализаций частичных и явных по отсутствию списка аргументов шаблона после имени. Список параметров частичной специализации шаблона может отличаться от списка параметров первичного шаблона.

Частичные специализации не являются отдельными сущностями, видимыми для поиска имён. Вместо этого, когда поиск имён находит первичный шаблон, все видимые в точке инстанциации частичные специализации рассматриваются на предмет использования их вместо первичного шаблона (если нет явной специализации с необходимыми параметрами, которые всегда имеют приоритет над первичным шаблоном и частичными специализациями). Частичные специализации, как и явные, могут быть описаны в окружающем пространстве имён.

Частичные специализации используются вместо первичного, если их список параметров может быть успешно дедуцирован из указанных аргументов шаблона. Если таких частичных специализаций оказывается несколько, происходит попытка выбрать из них более специализированную. Для этого синтезируются шаблоны вымышленных функций, по одному для каждой специализации. Их список параметров шаблона совпадает с таковым для частичной специализации, каждая из них имеет один параметр типа специализация рассматриваемого шаблона класса со списком аргументов, соответствующим списку аргументов частичной специализации (их возвращаемое значение не важно). Результат определения частичного порядка среди данных функций, успешный или нет, и определяет используемую частичную специализацию:

```
1   // 1. Первичный шаблон с тремя типовыми параметрами.
2   template<typename T,typename U,typename V>
3   struct S
4   {
5   };
6
7   // 2. Частичная специализация, используемая когда
8   // первый параметр - int, а два других - разные.
9   template<typename U,typename V>
10  struct S<int,U,V>
11  {
12  };
```

```
13
14  // 3. Частичная специализация, используемая когда
15  // первый параметр - int, а два других - одинаковые.
16  template<typename W>
17  struct S<int,W,W>
18  {
19  };
20
21  void f()
22  {
23      // Инстанциация из первичного шаблона 1, т.к.
24      // для частичных специализаций дедукция int из double
25      // в списке аргументов неудачна.
26      S<double,int,char> s1;
27
28      // Дедукция для 2 успешна и даёт U = double, V = char.
29      // Дедукция для 3 выводит из второго аргумента W = double,
30      // а из третьего - W = char, и заканчивается неудачей.
31      // Поскольку есть годные частичные специализации,
32      // первичный шаблон не рассматривается. Годная частичная
33      // специализация одна - это 2, инстанциация проводится из неё.
34      S<int,double,char> s2;
35
36      // Дедукция для 2 успешна: U = int, V = int.
37      // Дедукция для 3 тоже успешна: W = int.
38      // Для установки частичного порядка синтезируются описания функций:
39      // template<typename U,typename V> void f1(S<int,U,V>);
40      // template<typename W> void f2(S<int,W,W>);
41      // Дедукция параметров шаблона f2 по f1:
42      //   U = SomeType1, V = Sometype2, S<int,U,V> -> S<int,SomeType1,SomeType2>,
43      //   дедукция по S<int,W,W> даёт W = SomeType1, W = SomeType2 и неудачна.
44      // Дедукция параметров шаблона f1 по f2:
45      //   W = SomeType, S<int,W,W> -> S<int,SomeType,SomeType>
46      //   дедукция по S<int,U,V> даёт U = SomeType, V = SomeType и успешна.
47      // Успешна только дедукция параметров шаблона f1 по f2, поэтому тип
48      // параметра f2, сам шаблон f2 и, следовательно, специализация 3
49      // более специализированная, инстанциация производится из неё.
50      S<int,int,int> s3;
51  };
```

10.1.7. Поиск имён в шаблонах

Рассмотрим поиск имён в шаблонах. Все имена, используемые в шаблонах, можно поделить на три группы:

1. Имя самого шаблона и описанные в его определении имена;
2. Имена из областей видимости, просматриваемых по обычным правилам поиска имён в теле сущности, шаблон которой рассматривается;
3. Имена, содержащие параметры шаблона — ***зависимые имена (dependent names)***.

Поиск первых двух категорий имён осуществляется по обычным правилам.

Приведём пример, демонстрирующий необходимость дополнительных уточнений при поиске зависимых имён:

```
1  int y = 3;
2
3  template<typename T>
4  void f()
5  {
6      T::x * y;
7  }
8
9  struct S1
10 {
11     using x = int;
12 };
```

```
13
14  struct S2
15  {
16      static int x;
17  };
18
19  int S2::x = 2;
20
21  void g()
22  {
23      // T::x в теле специализации - якобы тип int,
24      // поэтому T::x * y - определение указателя на int
25      // по имени y, скрывающего имя из глобального пространства имён.
26      // На самом деле, ОШИБКА в C++.
27      f<S1>();
28
29      // T::x - имя статического члена данных класса типа int,
30      // поэтому T::x * y - умножение его значения на значение
31      // объекта y, определённого в глобальном пространстве имён
32      // с отбрасыванием результата.
33      f<S2>();
34  }
```

Неформально, обе специализации из приведённого примера кажутся корректными, хотя и имеют абсолютно разную семантику. На самом деле, такая многозначность выходит за пределы разрешённой правилами языка C++ даже для шаблонов, поскольку не позволяет транслятору выявить многие ошибки в определении шаблона ещё до того, как понадобятся какие-либо его специализации.

Указанную неоднозначность в рамках C++ разрешают, объявляя только вторую специализацию корректной: квалифицированные зависимые имена, если специально не указано обратное, не являются именами типов в описании шаблона. Если требуется обратное, следует явно затребовать это указанием ключевого слова `typename` перед зависимым именем:

```
36  // Продолжение предыдущего примера.
37
38  template<typename T>
39  void f2()
40  {
41      typename T::x * y;
42  }
43
44  void g2()
45  {
46      // Теперь верно, перед использованием T::x указано,
47      // что это имя типа явно.
48      f2<S1>();
49
50      // Теперь ОШИБКА, т.к. S2::x - не имя типа.
51      f2<S2>();
52  }
```

Таким образом, необходимо выбрать один из двух способов интерпретации зависимого имени — как имя типа или нет.

В конструкциях, где поиск имён в любом случае ищет только имена типов, например, в детальном имени классового типа, использовать ключевое слово `typename` не нужно. Когда имя в определении класса зависимо только от типа текущей специализации, указания `typename` тоже не нужно:

```
1   template<typename T>
2   void f()
3   {
4       // Имя с ключом классового типа -
5       // детальное имя, typename не требуется.
6       struct T::S2 x;
7   }
8
```

```
template<typename T>
struct S1
{
    using U = T;

    // S1<T> - текущая специализация,
    // U распознаётся как имя типа без typename.
    S1<T>::U x;

    struct S2 {};
};

void f()
{
    f<S1<int>>();
}
```

В этом примере также показана одна из хитрых связей этапа трансляции языка C++ с этапом предварительной обработки: два слитных знака `>` в предпоследней строке примера формально являются одним токеном операции сдвига вправо, но, когда имеются открытые списки аргументов шаблонов, этот токен может автоматически расщепляться на две закрывающие угловые скобки, чтобы разрешить более удобную форму записи часто встречающейся конструкции без добавления разделяющих пробелов — это было очень популярным расширением множества компиляторов, пока не попало в стандарт C++11. По той же причине, использовать знак операции `>` в выражении в списке аргументов шаблона не просто так нельзя — он воспринимается как закрывающая угловая скобка. В таком случае выражение необходимо заключить в пару «лишних» круглых скобок.

По тем же причинам, когда зависимое имя является шаблоном, необходимо явное указание этого, чтобы список аргументов шаблона распознавался как таковой. Для этого перед именем шаблона указывают ключевое слово `template`:

```
template<typename T>
void f(T x)
{
    // Явное указание, что g - член-шаблон x (т.е. T),
    // чтобы распознавался список аргументов шаблона.
    x.template g<int>();

    // Ключевое слово template не требуется, т.к. список
    // аргументов шаблона отсутствует. Хотя h и шаблон,
    // его параметры выявляются дедукцией в процессе
    // разрешения перегрузок.
    x.h(42);
}

struct S
{
    template<typename T>
    void g()
    {
        // ...
    }

    template<typename T>
    void h(T x)
    {
       // ...
    }
};

void h()
{
    S s;
    f(s);
}
```

Такое указание может происходить после операций прямой и косвенной выборки и разрешения области видимости.

Несмотря на желание сделать максимально полную и раннюю проверку корректности конструкций, входящих в определение шаблона, те из них, которые содержат зависимые имена, невозможно проверить при чтении определения шаблона, и эта проверка откладывается до инстанциации. Поиск имён и сопоставление с ними описаний при этом всё равно производится в контексте определения шаблона, и результат такого поиска может отличаться от результата поиска имён в точке инстанции или использования специализации шаблона:

```
template<typename T>
void f(T x)
{
    g(x);
}

void g(int x)
{
    std::cout << "x = " << x << '\n';
}

void h()
{
    // ОШИБКА. При инстанциации f с T = int
    //         поиск имени g осуществляется
    //         с точки зрения определения шаблона,
    //         т.е. до его определения.
    //         Несмотря на то, что аргументы g
    //         имеют зависимые типы и поиск имён
    //         был отложен до инстанциации, g
    //         не видно из определения шаблона,
    //         а аргументо-зависимый поиск не может
    //         помочь, т.к. тип int - базовый,
    //         и не имеет ассоциированных пространств имён.
    f(5);

    // В точке инстанциации g видно.
    g(5);
}
```

В таком случае для функций, которые не могут быть найдены аргументо-зависимым поиском необходимо дать их описание до определения шаблона. Для функций, которые могут быть найдены ADL, также можно переместить их описание в одно из ассоциированных пространств имён.

Приведённые в этом разделе ошибочные примеры некоторые компиляторы могут принимать за счёт отсутствия реализации довольно сложного механизма двухэтапной верификации шаблонов, регламентируемой стандартом, которая не требуется с точки зрения их внутреннего устройства — будьте внимательны!

> Часть типа параметра шаблона функции, стоящая слева от операции разрешения области видимости, является не-дедуцируемым контекстом — невозможно однозначно угадать, что подставить в качестве параметра шаблона, чтобы получилось выражение требуемой формы.

```
template<typename T>
void f(T x,typename T::U y)
{
    // ...
}

struct S
{
    using U = int;
};

void g()
```

```
13  {
14      S s;
15      // Второй параметр - не дедуцируемый контекст и в дедукции
16      // не участвует. Дедукция только по первому параметру,
17      // тем не менее, определяет все параметры шаблона: T = S.
18      // После подстановки получаем корректное описание специализации
19      // void f(S x,int y), которое может быть вызвано с указанными
20      // в операции вызова аргументами.
21      f(s,42);
22  }
```

10.2. Простые применения шаблонов

10.2.1. Списки инициализации в роли аргументов

Разрабатываемый нами шаблон класса `vector` пока имеет один недостаток по сравнению с обычным массивом: его нельзя инициализировать списком инициализации — только агрегаты среди классовых типов обладают таким свойством автоматически. Однако, эту способность можно реализовать вручную.

В заголовке `<initializer_list` в пространстве имён `std` определён шаблон одноимённого класса, с одним типовым параметром, определяющим тип элементов в представляемом им списке инициализации. Функция-член `size()` позволяет узнать число элементов в нём, а функции-члены `begin()` и `end()` возвращают указатели на неизменяемые элементы списка, первый и после-конечный. Список инициализации в аргументах функции (сам по себе не имеющий типа и даже не являющийся выражением) может быть преобразован к `std::initializer_list<T>`, при условии, что все элементы списка преобразуемы к `T`, ранг такого преобразования равен наихудшему среди преобразований отдельных элементов. Объект этого типа легко копировать, это не приводит к копированию описываемых им элементов.

Конструктор, который может быть вызван с одним параметром типа специализации `std::initializer_list` имеет особую роль: при инициализации объекта класса в форме со списком инициализации, такой конструктор может быть вызван со значением аргумента, соответствующим всему списку инициализации объекта, причём, если такая перегрузка вообще годна, то она лучше любых других, не использующих `std::initializer_list`.

Добавим к нашему шаблону класса возможность быть сконструированным из списка инициализации:

```
1  #include <initliazer_list>
2
3  template<typename T>
4  class vector
5  {
6  public:
7      vector(std::initializer_list<T> il)
8      // Делегировать конструирование перегрузке конструктора,
9      // копирующей элементы из указанного массива с указанием
10     // его длины.
11         : vector(il.begin(),il.size())
12     {
13     }
14     // Остальные члены шаблона класса
15 };
```

При использовании такой перегрузки конструктора может возникать странная ситуация, с которой мы не могли столкнуться раньше: прямая инициализация и инициализация списком с одинаковыми аргументами вызывает разные перегрузки. Покажем её на примере нашего шаблона:

```
1  template<typename T>
2  class vector
3  {
4  public:
5      // 1. Конструирование из списка инициализации
6      // с элементами хранимого типа.
7      vector(std::initializer_list<T> il);
```

```
8
9       // 2. Инициализация заданным числом элементов,
10      // инициализированных по значению.
11      vector(std::size_t initial_size);
12
13      // Остальные члены шаблона класса
14  };
15
16  void f()
17  {
18      // Вызывает конструктор 1, т.к. он годен,
19      // он автоматически лучше чем 2, не использующий
20      // std::initializer_list.
21      vector<std::size_t> v1{5};
22
23      // Вызывает 2. 1 не годен т.к. инициализация
24      // производится не в форме списка инициализации.
25      vector<std::size_t> v2(5);
26  }
```

> При дедукции параметров шаблона, когда аргумент является списком инициализации, а параметр функции имеет тип вида `std::initializer_list<(T)>`, где `(T)` — некоторый зависимый тип, то производится попытка вывести `(T)` из каждого типа элемента списка, для успеха результат должен быть одинаков для всех элементов, иначе дедукция завершается неудачей. Если аргумент — список инициализации, но параметр не имеет указанного вида, эта пара является не дедуцируемым контекстом. Если разрешение перегрузок завершится успешно с использованием этой перегрузки, список инициализации станет инициализатором объекта типа данного параметра, когда его тип уже будет известен.

10.2.2. Прозрачная передача аргументов. std::forward. Вариадические шаблоны.

Представление списка инициализации в виде шаблона класса `std::initializer_list` может использоваться для передачи функции произвольного числа параметров, но в этом случае мы ограничены элементам одного типа. Рассмотрим другое языковое средство, позволяющее решить эту задачу для произвольного числа аргументов произвольных типов на следующем примере. Нередко бывает необходимо принять некоторые аргументы для вызова другой, заведомо неизвестной функции. Их число, типы и категория значений заранее не известны, но должны быть сохранены в точности при вызове другой функции из нашей.

Попробуем начать решения этой задачи известными нам средствами. В качестве примера, где требуется такое решение, рассмотрим шаблон функции, создающей объект типа, заданного её типовым параметром, с передачей ему неизвестных по характеристикам аргументов. Такая функция, может пригодиться, например, если перед или после создания объекта необходимо каждый раз выполнять дополнительные действия, не заложенные в сам конструктор этого объекта.

Мы пока не знаем, как принять произвольное число аргументов (вариадические функции из языка C использовать нельзя, т.к. они не передают информации о типе аргументов, которая нам нужна), так что начнём с решений задачи в частных случаях. Для конструкции без параметров решение тривиально:

```
1  template<typename T>
2  T create()
3  {
4      return T();
5  }
```

Перейдём к конструированию по одному аргументу. Поскольку аргументы потенциально могут быть не копируемыми, их необходимо принять по ссылке. Рассмотрим возможные варианты в зависимости от вида ссылки и её квалификаторов (передачу `volatile` параметров пока не будем рассматривать):

- По леводопустимой ссылке на неизменяемое значение. К такой ссылке привяжуться значения любой категории и изменяемости, но полученная ссылка не может быть передана дальше, если вызываемая функция принимает параметр без квалификатора `const`. Избавится от него

в общем случае через `const_cast` некорректно, т.к. потенциально вызывает неопределённое поведение и выбор не той перегрузки вызываемой функции из-за смены типа аргумента.

- Леводопустимая ссылка на изменяемое значение. Этот вариант не добавляет своих квалификаторов, дедуцирует имеющиеся, но к нему привязываются только леводопустимые аргументы.

Попытка использовать комбинацию из этих двух вариантов в виде перегрузок потребует 2^n перегрузок для n аргументов, что неприемлемо, и всё равно не будет точно передавать квалификаторы и категорию не леводопустимых аргументов.

Решить эту проблему можно, вспомнив специальное правило дедукции для параметра вида в точности праводопустимая ссылка на типовой параметр шаблона: параметр шаблона дедуцируется в леводопустимую ссылку для леводопустимых значений, и просто тип аргумента для не леводопустимых значений. За счёт коллапса ссылочных типов результирующий тип параметра функции становится лево- и праводопустимой ссылкой на тип аргумента соответственно, таким образом такой параметр шаблона функции позволяет привязать к нему значения любых категорий.

В теле функции этот параметр имеет имя, и потому всегда леводопустим. Чтобы восстановить его исходную категорию значения, его можно привести к праводопустимой ссылке на дедуцированный тип параметра шаблона, что в результате аналогичного коллапса даст вновь ссылку правильного типа:

```
1  template<typename T,typename Arg>
2  T create(Arg&& arg)
3  {
4      return T(static_cast<Arg&&>(arg));
5  }
```

Эта конструкция работает следующим образом:

- Передаётся леводопустимый аргумент типа `type`. Дедукция даёт `Arg = type&`, тип параметра функции становится `type& &&`, за счёт коллапса ссылок — `type&`. В теле функции леводопустимый аргумент приводится к этому же типу, и остаётся леводопустимым.
- Передаётся не леводопустимый аргумент типа `type`. Дедукция даёт `Arg = type`, тип параметра функции становится `type &&`. В теле функции леводопустимый аргумент приводится к `type&&` и становится не леводопустимым, каким он был изначально передан этой функции.

Эти варианты работают для значений с любыми квалификаторами, дедуцируя их как часть `type`. Мы уже встречались с подобными приведениями к праводопустимой ссылке, чтобы чётко выразить его смысл в данном случае, обычно пользуются библиотечной функцией, которая совершает это приведение типов. Это шаблон функции `std::forward` из заголовка `<utility>`, который требует явного указания его единственного типового параметра как результата дедукции в вызываемой функции для обеспечения корректности передачи:

```
1  template<typename T,typename Arg>
2  T create(Arg&& arg)
3  {
4      return T(std::forward<Arg>(arg));
5  }
```

Хотя имя этого шаблона выражает смысл действия — *передачу* аргумента, следует помнить, что никакой «волшебной передачи» он не совершает, а только корректирует категорию значения её аргумента. Такая передача по универсальной ссылке плюс коррекция категории для дальнейшее передачи, которая сохраняет все характеристики первоначального аргумента, называется ***прозрачной передачей (perfect forwarding)***.

Таким образом, для каждого числа аргументов достаточно одной перегрузки, но, оказывается, все они могут быть сведены к одной единственной.

Шаблоны C++ содержат средство представления последовательностей типов или параметров функции в виде ***пачек параметров (parameter pack)***. ***Пачка параметров шаблона (template parameter pack)*** — специальный вид параметра шаблона (типового, не типового или шаблонного), когда один параметр шаблона соответствует произвольной упорядоченной последовательности аргументов шаблона того же вида (включая пустой). Шаблоны, имеющие пачки параметров, называют ***вариадическими (variadic)***. В описании такого параметра после указания его типа ставится троеточие. Единственная операция, которая может быть применена к пачке в целом — операция `sizeof...`, возвращающая константное значение, соответствующее числу аргументов в пачке:

```
template<typename... Ts>
class C
{
    static void f()
    {
        std::cout << sizeof...(Ts) << " template argument(s)!\n";
    }
};

void g()
{
    // Выводит 3:
    C<int,bool,double>::f();
}
```

Единственным способом доступа к отдельным элементам пачки является операция ***распаковки пачки (pack expansion)*** `...` (троеточие). Её аргумент называют ***образцом (pattern)***, он должен быть некоторой конструкцией языка, содержащей ещё не распакованные пачки с одинаковым количеством аргументов в каждой. Результат распаковки эквивалентен записи стольких копий образца, сколько аргументов в каждой пачке, разделённых запятыми. В каждой копии каждое вхождение имени пачки заменяется очередным входящим в неё аргументом. Если пачка пуста или содержит только один элемент, беспокоиться о синтаксически лишней запятой до или после распаковки пачки не нужно — язык обрабатывает такие неточности автоматически.

Операция распаковки может применяться только в некоторых контекстах:

- В списке аргументов операции вызова функции пачка распаковывается в последовательность аргументов функции.
- В списке аргументов шаблона пачка распаковывается в последовательность аргументов шаблона.
- В списке инициализации в фигурных скобках пачка распаковывается в последовательность инициализаторов.
- В качестве типа параметра функции — об этом случае поговорим отдельно.

Последний рассмотренный вариант позволяет создавать специальный вид параметров функции — ***пачку параметров функции (function parameter pack)***. Такой аргумент соответствует (возможно, пустой) последовательности аргументов функции, типы которых определяются входящими в образец её типа пачками параметрамов шаблонов данной функции.

К пачке параметров функции также применима операция `sizeof...`, результат которой есть целочисленное константное выражение со значением числа элементов в пачке.

Если пачка параметров функции является её последним параметром, она позволяет принять произвольное число аргументов и дедуцировать пачки параметров шаблона, входящие в описание её типа по типу этих аргументов.

> Если такая пачка — не последний параметр функции, она не дедуцируема, а не дедуцируемые пачки параметров шаблона считаются по умолчанию пустыми последовательностями. Это можно изменить явным указанием значений соответствующих пачек параметров шаблона. При явном указании аргументов для пачек параметров шаблонов по достижении первой такой пачки все последующие аргументы в списке аргументов шаблона захватываются ей, так что все остальные должны иметь значения по умолчанию или быть дедуцируемы. По этой причине шаблон класса может иметь только одну пачку параметров, которая должна быть последним параметром шаблона.

```
template<typename... Ts>
void f(Ts... xs);

template<typename... Ts>
void g(Ts... xs,int y);

int main()
{
    // Дедукция: Ts = [int,char,bool]
    f(1,'c',true);

```

```
    // Дедукция: Ts = [], Us = [int]
    g(42);

    // ОШИБКА: нет годной перегрузки, xs - пачка параметров,
    //         не являющаяся последним параметром, дедуцируется
    //         в пустую последовательность, итого у шаблона функции
    //         g только один параметр y, а передаются два аргумента.
    g(12,34);

    // Явно задано Ts = [int].
    g<int>(12,34);
}
```

Поскольку в явном виде нельзя получить доступ к отдельным элементам пачки, алгоритмы, их использующие, для перебора всех значений вместо циклов часто используют рекурсию:

```
// Конец рекурсии: параметров нет
void print_sizes()
{
    // Ничего не делаем.
}

// Шаг рекурсии: есть один аргумент head и
// (потенциально пустая) последовательность остальных tail.
template<typename T,typename... Ts>
void print_sizes(const T& head,const Ts&... tail)
{
    // Первый аргумент отщеплён в head, остальные попали в пачку tail.
    // Обработать первый параметр:
    std::cout << sizeof head << '\n';
    // Рекурсивно вызвать себя с остальными параметрами.
    print_sizes(tail...);
}

void f()
{
    // Может вывести 4 8 1.
    // Вызывает print_sizes<int,double,char>(2,3.5,'x'), которая
    // вызывает print_sizes<double,char>(3.5,'x'), которая
    // вызывает print_sizes<char>('x'), которая
    // вызывает print_sizes().
    print_sizes(2,3.5,'x');
}
```

Теперь мы можем записать одну единственную функцию, выполняющую прозрачную передачу произвольного числа аргументов конструктору указанного типа:

```
template<typename T,typename... Args>
T create(Args&&... args)
{
    // Образец с двумя пачками. При распаковке копии образца
    // содержат последовательно все первые элементы всех пачек,
    // затем все вторые и т.д. Поскольку это пачка параметров
    // шаблона функции и использующая его в своём образце пачка
    // параметров функции, в них всегда будет одинаковое количество
    // элементов.
    return T{std::forward<Args>(args)...};
}
```

10.2.3. std::move. Метафункции. Шаблоны псевдонимов типов

Другой ситуацией, в которой мы уже сталкивались с необходимостью приведения к правопустимой ссылке — указание на то, что содержимое объекта более не нужно и может быть перемещено:

```
class big_class
{
```

```
3       big_class(const big_class&);
4       big_class(big_class&&);
5       big_class& operator=(const big_class&);
6       big_class& operator=(big_class&&);
7       // Другие члены
8   };
9
10  void consume_big_class(const big_class&);
11  void consume_big_class(big_class&&);
12
13  void f()
14  {
15      big_class bc;
16
17      // ...
18
19      // bc более не нужен и может быть перемещён:
20      consume_big_class(static_cast<big_class&&>(bc))
21  }
```

Хотелось бы заменить это приведение типов более лаконичной и выражающей намерение конструкцией. Попробуем написать для этого шаблон функции, хотя с первого раза два фрагмента написать не удастся:

```
1   // Шаблон функции для придания значениям
2   // категории значения rvalue.
3   // T дедуцируется из типа параметра, который
4   // является универсальной ссылкой, чтобы
5   // принимать значения любых категорий (полезно для
6   // использования в других шаблонах в общем случае).
7   template<typename T>
8   ??? move(T&& x)
9   {
10      return static_cast<???>(x);
11  }
```

В возвращаемом значении функции и операции приведения типов должна стоять правопустимая ссылка на тип исходного выражения, но записать просто `T&&` неверно: для левопустимых аргументов функции `move T` дедуцируется в левопустимую ссылку, которая при коллапсе с правопустимой в этом имени типа останется левопустимой. Как показывает этот пример, создать более сложный тип обычно можно на лету, просто записав вокруг исходного необходимую конструкцию создания производного типа, но для обратной операции — получения базовых и других частей имеющихся производных типов потребуются дополнительные усилия.

Нам необходимо, имея один тип, получить другой из него по некоторому алгоритму. Имеющееся понятие «функция» не подходит, поскольку функции оперируют значениями, а не типами. На этом примере автор предлагает совершить небольшое знакомство со средствами программирования языка C++, которые в языке возникли совершенно непредвиденно благодаря находчивому применению шаблонов.

Конструкция, в некотором смысле принимающая в качестве параметров типы, и возвращающая другие типы, имеет не официальное, но устоявшееся название — ***метафункция (metafunction)***. В стандартной библиотеке языка C++ принято реализовывать её как шаблон структуры, параметра шаблона которого и есть параметры метафункции. Результат своего «вычисления» метафункция должна сделать доступным в качестве своего вложенного типа `type`, чаще всего это псевдоним типа. Рассмотрим устройство метафункции `remove_reference`, возвращающей данный ей тип с отнятой конструкцией создания ссылки любого вида на верхнем уровне, если она есть:

```
1   template<typename T>
2   struct remove_reference
3   {
4       using type = T;
5   };
6
7   template<typename T>
```

```
8  struct remove_reference<T&>
9  {
10     using type = T;
11 };
12
13 template<typename T>
14 struct remove_reference<T&&>
15 {
16     using type = T;
17 };
```

Первичный шаблон в данном случае обрабатывает общий случай: для типов, не являющимися ссылками, результат отнимания ссылки есть сам исходный тип. Если исходный тип является лево- или праводопустимой ссылкой, при инстанциации этого шаблона используется первая или вторая частичная специализация соответственно. За счёт дедукции в них в качестве `T` выводится базовый тип ссылки, который определяется в качестве «возвращаемого значения» метафункции. Используется метафункция с помощью её инстанциации и обращения к её вложенному типу `type`:

```
1  // T1 = double, отнятие ссылки от не ссылочного типа даёт сам тип.
2  using T1 = remove_reference<double>::type;
3
4  // T2 = int
5  using T2 = remove_reference<int&&>::type;
```

Когда в качестве параметров метафункции, как в примере выше, используются наперёд заданные типы, то и вызов метафункции, вообще говоря, не нужен — требуемое преобразование можно выполнить самостоятельно и записать сразу результат. Их область применения — это другие шаблоны, где их аргументы являются зависимыми и заранее не известными типами. В таком случае придётся перед всей конструкцией записать ещё и требуемое в таком случае ключевое слово `typename`. Чтобы избежать громоздкости этой записи, можно воспользоваться ***шаблоном псевдонима типа (alias template)***:

```
1  template<typename T>
2  using remove_reference_t = typename remove_reference<T>::type;
```

Шаблон псевдонима типа синтаксически — список параметров шаблона в стандартном виде, за котором следует определение псевдонима типа в форме с ключевым словом `using` (старый вариант с `typedef` не допускается). После определения имя этого шаблона с указанным списком аргументов шаблонов может использоваться в качестве имени типа, который определяется подстановкой аргументов в конструкцию имени типа в его правой части. Как и для шаблонов классов, дедукция для шаблонов псевдонимов типов не поддерживается, а список аргументов шаблона при использовании обязателен, даже если пуст: значения параметров берутся только из списка аргументов и значений по умолчанию. Аналогичны таковым для классов и требования к пачкам параметров шаблона.

Теперь мы можем записать полное определение шаблона `move`, вычислив требуемый нам тип следующим образом: отнимем с помощью метафункции от него любую имеющуюся ссылку и добавим праводопустимую назад:

```
1  template<typename T>
2  remove_reference_t<T>&& move(T&& x)
3  {
4      return static_cast<remove_reference_t<T>&&>(x);
5  }
```

Данная метафункция и соответствующий шаблон псевдонима типов описаны в стандартной библиотеке в заголовочном файле `<type_traits>`. В этом заголовке также содержатся много других метафункций по манипулированию и определению свойств типов, с которыми читатель может ознакомиться по справочнику. Написанный нами шаблон функции `move` также имеется в стандартной библиотеке в заголовке `<utility>`.

Теперь исходный пример можно записать понятнее:

```
1  #include <utility>
2
3  void f()
```

```
4  {
5      big_class bc;
6
7      // ...
8
9      // bc более не нужен и может быть перемещён:
10     consume_big_class(std::move(bc));
11 }
```

Остаётся напомнить: несмотря на своё имя, шаблон функции `move` ничего сам не «перемещает»: он только потенциально меняет категорию значения таким образом, что при наличии соответствующих перегрузок появляется возможность вызвать специальные функции классов и другие функции, работающие с праводопустимыми ссылками.

10.2.4. std::exchange

`std::exchange` — шаблон функции из заголовка `<utility>`, которому требуется комбинировать многие рассмотренные нами конструкции для чёткого решения в общем случае, на первый взгляд, тривиальной задачи: заменить старое значение некоторого объекта новым, вернув старое значение (потребуется сохранить его во временном объекте). Его поведение задано в стандарте языка явным фрагментов кода:

```
1  // T - тип изменяемого объекта
2  // U - тип нового значения.
3  // Это отдельный типовой параметр, потому что
4  // у T могут быть произвольные перегрузки operator=.
5  // U по умолчанию приравнен к T для удобства:
6  // например, для задания соответствующего аргумента в форме {}.
7  // В большинстве случаев, когда функция явно вызывается,
8  // эти параметры будут дедуцированы автоматически.
9  template<typename T,typename U = T>
10 T exchange(T& obj,U&& new_val)
11 {
12     // Старое значение перемещаем во
13     // временный объект, поскольку всё
14     // равно его тут же заменим.
15     T old_val(std::move(obj));
16
17     // new_val принято по универсальной ссылке,
18     // прозрачно передадим его операции присваивания
19     // на типе T, чтобы она, например, могла тоже
20     // использовать перегрузку перемещения,
21     // если был передан перемещаемый аргумент.
22     obj = std::forward<U>(new_val);
23
24     // Вернуть старое значение.
25     return old_val;
26 }
```

10.2.5. Автоматический вывод типов в языке C++. Заполнители. Хвостовой возвращаемый тип функции

Механизм дедукции является проявлением свойства автоматического вывода типов в языке C++. Это полезное свойство может применяться и независимо от шаблонов.

Попробуем написать функцию над двумя аргументами, действие которой аналогично действию операции сложения над двумя аргументами произвольных типов. Для начала предположим, что такая операция не модифицирует аргументы, и возвращает новый объект, и не будем применять прозрачную передачу аргументов:

```
1  template<typename T,typename U>
2  ??? add(const T& x,const U& y)
3  {
4      return x+y;
5  }
```

Возникает проблема с записью возвращаемого значения функции — оно зависит от типов `T` и `U` и в общем случае может быть любым за счёт перегруженных операций классовых типов. Подобно операциям `sizeof` и `alignof`, возвращающим размер и выравнивание для произвольных типов и выражений, в языке имеется операция `decltype`, вычисляющая тип своего аргумента. Она может быть использована везде, где требуется имя типа и возвращает тип своего аргумента по следующему алгоритму:

1. Если её аргумент — в точности имя объекта или не перегруженной функции, то возвращается их тип;
2. Иначе если аргумент — выражение категории xvalue, возвращается праводопустимая ссылка на тип выражения;
3. Иначе если аргумент — выражение категории lvalue, возвращается леводопустимая ссылка на тип выражения;
4. Иначе возвращается тип аргумента-выражения.

Будьте аккуратны с первым из возможных вариантов:

```
1  int x;
2
3  // decltype(x) есть тип x, т.е. int.
4  decltype(x) y1 = x;
5
6  // Уже пара скобок вокруг - не имя объекта, а выражение,
7  // decltype((x)) есть int&, т.к. (x) - леводопустимое выражение.
8  decltype((x)) y2 = x;
```

Таким образом, необходимый нам тип возвращаемого значения функции — `decltype(x+y)`. Однако записать его в таком виде по известным нам правилам нельзя — он будет указан перед именем функции, где имена `x` и `y` ещё не описаны, т.к. список параметров функции идёт дальше. Для решения этой проблемы в языке имеется альтернативной способ записи описаний функций из одного описателя: вместо типа возвращаемого значения функции в обычном месте пишется ключевое слово `auto`, а в конце описания функции пишется пунктуатор `->`, за которым записывается настоящий тип возвращаемого функцией значения. Таким образом, он синтаксически оказывается после списка параметров функции, где их имена уже видны:

```
1  template<typename T,typename U>
2  auto add(const T& x,const U& y) -> decltype(x+y)
3  {
4      return x+y;
5  }
```

Такой вид описаний применим всегда, когда описание функции содержит только один описатель и называется формой описания функции с ***хвостовым возвращаемым типом (trialing return type)***. Для функций-членов класса хвостовой возвращаемый тип и квалификаторы параметра-объекта могут идти в любом порядке. Некоторые программисты предлагают использовать стиль, в котором этот синтаксис используется для всех функций вообще — утверждается, что тогда из-за одинаковой длины слова `auto` в каждом описании, последовательность многих описаний функций подряд удобнее читать, так как имена всех функций начинаются в одном столбце. Автор этого текста не будет призывать к столь радикальным переменам.

Сделаем ещё один шаг, применив, наконец, автоматический вывод типов. Можно не указывать хвостовой возвращаемый тип вообще, оставив на привычном месте типа возвращаемого функцией значения `auto`:

```
1  template<typename T,typename U>
2  auto add(const T& x,const U& y)
3  {
4      return x+y;
5  }
```

В таком случае тип возвращаемого функцией значения будет дедуцирован из выражений во всех операторах `return` в её теле. Дедукция происходит словно для некоторого фиктивного типового параметра шаблона, представленного ключевом словом `auto` в возвращаемом значении функции как параметре, дедукция по выражениям в операторах `return` в качестве аргументов

(и может быть неудачна как обычно, если несколько выражений в операторах `return` имеют разные типы). Такая функция может быть использована только после её определения, т.к. без наличия тела функции нельзя дедуцировать тип её возвращаемого значения.

Поскольку в нашем случае возвращаемое значение состоит только из ключевого слова `auto`, это эквивалентно дедукции по параметру типа `T`, где `T` — некоторый типовой параметр шаблона. Таким образом, дедукция отбросит ссылки и квалификаторы от выражения операторе `return`, как обычно для дедукции параметров такого вида, но нас это устраивает, пока мы считаем, что операция сложения возвращает результат по значению.

Таким образом, `auto` является ***заполнителем (placeholder)*** некоторого типа и может участвовать в более сложных выражениях. Кроме этого, его можно применять не только в возвращаемом значении функций, но и при определении объектов с блочной видимостью или областью видимости пространства имён. В таком случае дедукция осуществляется по типу инициализатора, который в таком определении обязателен:

```
1  // Заполнители в области видимости пространства имён (глобального).
2
3  // Дедукция T по 5, T = int, подставляя назад, x имеет тип int.
4  auto x = 5;
5
6  // Дедукция const T& по x, T = int, подставляя назад y имеет тип const int&.
7  const auto& y = x;
8
9  int x[5];
10
11 auto* f(size_t i)
12 {
13     // Дедукция возвращаемого типа даёт после подстановки int*.
14     return x+i;
15 }
16
17 // ОШИБКА: дедукция заполнителя даёт два разных варианта int и double.
18 auto g(bool v)
19 {
20    if(v)
21        return 3;
22    else
23        return 4.5;
24 }
```

Вернёмся к рассмотрению примера с эмуляцией операции сложения в общем виде. Будем теперь считать, что, как это и есть на самом деле, операция сложения, как любая функция с потенциально неизвестным множеством перегрузок, может принимать и возвращать значения любых категорий, по значению или ссылкам. Для начала добавим прозрачную передачу параметров:

```
1  template<typename T,typename U>
2  auto add(T&& x,U&& y)
3  {
4      return std::forward<T>(x)+std::forward<U>(y);
5  }
```

Эта версия прозрачно передаёт аргументы нашей функции операции сложения, но не делает того же с возвращаемым значением: поскольку используется форма из одного имени типового параметра, в качестве которого выступает `auto`, дедукция отбрасывает ссылки и квалификаторы, что может вызывать, например, лишнее копирование вовзращаемого операцией сложения значения, если оно является ссылкой. Мы можем вернуть хвостовой возвращаемый тип, поскольку операция `decltype` корректно возвращает тип всех трёх категорий значений: lvalue, xvalue (как ссылки) и rvalue.

```
1  template<typename T,typename U>
2  auto add(T&& x,U&& y) -> decltype(std::forward<T>(x)+std::forward<U>(y))
3  {
4      return std::forward<T>(x)+std::forward<U>(y);
5  }
```

Этот вариант полностью прозрачен, но громоздок. Последнее упрощение можно получить, воспользовавшись вместо заполнителя `auto` во варианте с дедукцией специальным заполнителем `decltype(auto)`. Этот заполнитель может использоваться только сам по себе, не являясь частью более сложного типа, и предписывает дедукцию по правилам операции `decltype`:

```
1 template<typename T,typename U>
2 decltype(auto) add(T&& x,U&& y)
3 {
4     return std::forward<T>(x)+std::forward<U>(y);
5 }
```

В таком виде, если операция сложения возвращает по значению, наша функция тоже возвращает по значению (и допустимо RVO), а если по ссылке, то и наша функция возвращает по ссылке и лишних копий не создаётся.

Хотя этот заполнитель может использоваться, как и `auto`, не только в возвращаемом значении функции, использование его в этом качестве является наиболее частым случаем его использования для шаблонов функций, желающих прозрачно вернуть значение другой функции.

10.3. Стандартная библиотека шаблонов

Стандартная библиотека шаблонов — часть стандартной библиотеки языка C++, содержащая реализации основных структур данных и алгоритмов. Исторически она была отдельной частью, разработанной Александром Степановым и Менг Ли, но по стечению обстоятельств вошла в стандарт языка и стала одной из основных частей стандартной библиотеки C++.

Реализованные с точки зрения обобщённого программирования, все эти средства являются шаблонами. Для обеспечения общности между способами хранения данных и их обработки потребуется ввести промежуточное звено — итераторы. Начать же рассмотрение придётся с введения формальных требований к параметрам шаблонов, в терминах которых даны ограничения использования всех интересных нам библиотечных средств.

10.3.1. Простые концепции. Не типовые параметры шаблонов

Начнём с формального описания самых простых концепций.

В языке описана группа концепций, характеризующих объекты по свойствам их специальных функций:

- `DefaultConstructible` — типы, которые могут быть сконструированы по умолчанию или по значению;
- `MoveConstructible` — типы, которые могут быть сконструированы из праводопустимого значения собственного типа, которое может иметь не уточняемое значение после конструирования;
- `CopyConstructible` — типы, которые могут быть сконструированы из леводопустимого или неизменяемого праводопустимого значения собственного типа, которое сохраняет значение после конструирования.
- `MoveAssignable` — типы, которым может быть присвоено праводопустимое значение собственного типа, которое может иметь не уточняемое значение после присваивания;
- `CopyAssignable` — типы, которым может быть присвоено леводопустимое или неизменяемое праводопустимое значение собственного типа, которое сохраняет значение после конструирования.
- `Destructible` — тип, имеющий доступный деструктор (типы, не удовлетворяющие этой концепции встречаются редко).

Легко заметить, что концепции `MoveConstructible` и `MoveAssignable` допускают семантику переноса, но не требуют её, поэтому типы, удовлетворяющие `CopyConstructible`, удовлетворяют и `MoveConstructible`, то же для концепций присваивания, но не наоборот. Стандарт рассматривает концепции конструирования и присваивания по отдельности, хотя когда мы давали характеристику типов по реализации ими специальных функций, мы договорились о параллельной их реализации. Стандарт же рассматривает общий случай, включая все странные исключения, и старается накладывать минимально возможные требования на типы, и потому его набор концепций по поведению специальных функций более детален.

В качестве более сложной концепции, требующей соблюдения некоторого протокола использования объектов соответствующего типа, приведём концепцию `Swappable` — пару объектов типа, удовлетворяющего ей можно обменять значениями. Стандартная библиотека содержит следующие описания в заголовочном файле `<utility>` в пространстве имён `std` (в упрощённом виде):

```
template<typename T>
void swap(T& x,T& y)
{
    // Обменять значения через третье временное,
    // по возможности перемещая вместо копирования.
    T t(std::move(x));
    x = std::move(y);
    y = std::move(t);
}

template<typename T,size_t N>
void swap(T (&x)[N],T (&y)[N]);
```

Последнее описание является перегрузкой, обменивающей местами попарно элементы двух массив одного типа элементов и длины. Первая перегрузка в общем случае не подходит для массивов, поскольку для них из-за разложения не применима операция присваивания.

Это описания также является примером использования не типового параметра шаблона, а также примером, где он может быть дедуцирован. Тип параметров этого шаблона — ссылка на массив из n элементов типа `T`. Поскольку дедукция осуществляется для ссылочного типа, массив не разлагается, тип элемента используется для дедукции типового параметра шаблона, а размерность — для дедукции не типового. Как видно из примера, описание не типового параметра шаблона содержит вместо ключевого слова `typename` обычное имя типа. В качестве типов не типовых параметров шаблонов могут использоваться: целочисленные типы, типы-перечисления, указатели и леводопустимые ссылки на объекты и функции и `std::nullptr_t`. Следует учитывать, что аргументы шаблонов должны быть константными выражениями, что особенно важно для указателей и ссылок.

> Кроме того, для таких параметров имеются дополнительные ограничения по их форме и неявному приведению типов в случае, если они не совпадают в точности с типом параметра шаблона. Эти ограничения в данном тексте рассматриваться не будут, поскольку их описание объёмно, но с ними редко сталкиваются на практике.

Обмен через третий объект — стандартная реализация, работающая для любых типов, удовлетворяющих `MoveConstructible` и `MoveAssignable`, но для некоторых типов может существовать более эффективная реализация. Эту реализацию необходимо выполнить в виде перегрузки функции с именем `swap` и соответствующими типами параметров, расположенной в том же пространстве имён, что и сам тип, чтобы быть доступной для аргументо-зависимого поиска имён.

Концепция `Swappable` предписывает следующий протокол для обмена местами двух объектов удовлетворяющего ей типа: сделать видимыми два общих описания из пространства имён `std`, после чего вызывать функцию `swap` без квалификации её имени, так что помимо этих стандартных будут также рассмотрены оптимизированные перегрузки для конкретных классов, найденные через ADL:

```
namespace my_ns
{
    // Шаблон класса простейшего односвязного списка.
    template<typename T>
    class my_linked_list
    {
    public:
        // ...

        // См. ниже, почему сам обмен реализован
        // функцией-членом
        void swap(my_linked_list& other)
        {
            // Просто обменять два указателя.
            // Вызывается напрямую стандартная реализация
```

```
16              // swap, т.к. для обычных указателей не может
17              // быть более эффективной.
18              std::swap(head,other.head);
19          }
20      private:
21          // Единственный член данных:
22          // указатель на голову.
23          T* head;
24      };
25
26      // Эффективный обмен значений двух списков.
27      template<typename T>
28      void swap(my_linked_list<T>& x,my_linked_list<T>& y)
29      {
30          // Делегировать обмен функции-члену,
31          // чтобы не мучиться с описаниями друзей шаблонов.
32          x.swap(y);
33      }
34  }
35
36  // Шаблон функции, требующей чтобы T удовлетворял Swappable
37  template<typename T>
38  void swap_and_do_something_else(T& x,T& y)
39  {
40      // ...
41
42      // Здесь требуется обменять x и y местами, следуем
43      // протоколу Swappable:
44      // Делаем видимыми стандартные перегрузки из std:
45      using std::swap;
46      // Делаем не квалифицированный вызов swap:
47      swap(x,y);
48
49      // ...
50  }
```

10.3.2. Концепции итераторов

Рассмотрим пример шаблона функции линейного поиска:

```
1  template<typename T,typename U>
2  T linear_search(T from,const T& to,const U& value)
3  {
4      while(from!=to&&*from!=value)
5          ++from;
6      return from;
7  }
```

Этот шаблон функции осуществляет поиск значения, равного `value` в диапазоне $[from; to)$. Хотя мы могли использовать `T*` вместо просто `T` в описании функции, чтобы ограничить использование функции только с указателями для задания диапазона поиска, это оказалось не нужным. Вместо этого требования к типу `T` можно изложить следующим образом: к нему должна быть применима операция разыменования в смысле чтения значения по указываемому им месту, операция инкремента в смысле перехода к следующему значению и операция сравнения на равенство с другим объектом того же типа для проверки, что они указывают на одно и то же место. Для сигнализации неудачи поиска мы возвращаем правую границу интервала поиска, не входящую в него, поэтому дополнительных требований на тип `T` нет. Легко убедиться, что, например, обычные указатели удовлетворяют этим требованиям.

Перечисленные нами требования, на самом деле, являются основой концепции, более широкой, чем только указатели — концепции ***итераторов (iterator)***. Её смысл — понятие объекта, «указывающего» на последовательность других объектов одного типа, и позволяющего по ней перемещаться. (В данном случае под последовательностью понимается более широкий смысл «упорядоченное множество», а не рассмотренный нами ранее класс структур данных.) Одно из основных требований всех концепций итераторов — все операции, требуемые

концепциями итераторов должны занимать константное время. Это требование определяет в том числе и иерархию этих концепций.

Формально концепция `Iterator` задаётся следующим образом:

- Она уточняет `CopyConstructible`, `CopyAssignable`, `Destructible`;
- Леводопустимые изменяемые выражения этого типа должны удовлетворять `Swappable`;
- Применение унарной операции * к объекту этого типа должно возвращать значение, на которое он указывает. Так же, как и для указателей, не каждый итератор ***разыменуем (dereferencable)*** в конкретный момент времени: не разыменуемы итераторы, указывающие на послеконечный элемент последовательности, ***сингулярные (singular)*** итераторы, не связанные ни с какой последовательностью (например, нулевой указатель) и ***недействительные (invalidated)*** итераторы — указывающие на объекты, которые более не существуют или были перемещены в другое место в памяти (например, висячие указатели). Формально, тип возвращаемого значения этой операции носит название `reference`, хотя может и не быть просто ссылкой.
- Применение унарной префиксной операции `++` сдвигает его на следующий элемент последовательности. Как и с указателями, эта операция не применима ко всем итератором, например, к итератору на послеконечный элемент последовательности.

Это наиболее общая концепция, которая ничего не говорит даже о категории значения, получаемого его разыменованием. На практике применяют одну из пяти уточняющих её концепций.

Концепция `InputIterator` (входной итератор) соответствует итератору, с помощью которого можно осуществить ***однократный проход (single-pass)*** по последовательности объектов с чтением их значений. Она используется как требования к итераторам алгоритмов, которым достаточно одного прохода по входным данным. Её требования:

- Уточняет `Iterator` и `EqualityComparable`. `EqualityComparable` — концепция, требующая наличие операций сравнения на равенство между парой объектов удовлетворяющего ей типа с возвратом значения, приводимого к `bool` с соответствующим смыслом. Сравнение допустимо только если оба итератора указывают на элементы одной последовательности. Наличие этого требования позволяет сравнивать итераторы, чтобы отследить конец последовательности, заданной парой итераторов на начальный и послеконечный элементы.
- Наличие операции сравнения на неравенство, с противоположным равенству смыслом (обобщение, минимальной концепцией `EqualityComparable` не требуемое).
- Наличие неявного преобразования из типа `reference` в тип значения, на которое указывает итератор, именуемого `value_type`. Разыменования равных разыменуемых итераторов должны давать одинаковый результат. Многократное разыменование одного итератора допустимо и должно давать одинаковый результат.
- Операция косвенной выборки должна быть эквивалентна разыменованию с последующей прямой выборкой, как и для обычных указателей, если итератор разыменуем.
- Операция преинкремента возвращает ссылку на сам итератор. После такой операции все итераторы, полученные копированием исходного могут перестать быть разыменованными. Кроме того, из `a == b` ***не*** следует `++a == ++b`. Таким образом, для итераторов этой концепции возможность их копирования полезна только временно для передачи, поскольку из всех временных копий для дальнейшего перемещения по последовательности можно использовать только одну. Это, странное на первый взгляд, свойство позволяет, например, маскировать под входные итераторы последовательности, которые не существуют постоянно в памяти, а вычисляются поэлементно на каждом шаге, поэтому хранится только текущий элемент и старый перестаёт существовать при переходе к следующему.
- Наличие операции постинкремента со стандартной семантикой.

Требования концепции `InputIterator` позволяют перемещаться по последовательности элементов и осуществлять их чтение. Поскольку алгоритмы не только используют последовательности как входные данные, но и генерируют их в качестве результата своей работы, существует параллельная концепция `OutputIterator` (выходной итератор). Она требует:

- Уточняет `Iterator`. `EqualityComprabale` не требуется, т.к. число выходных значений алгоритма обычно уже определено по входным параметрам, так что в отличие от пары итераторов для входной последовательности элементов, для записи результатов часто указывается только один начальный выходной итератор.

- Применимость результата разыменования итератора в качестве левого операнда операции простого присваивания. В качестве правого операнда должно выступать значение, записываемое в очередной элемент последовательности. Конкретный его тип не указывается, обычно это все типы, из которых может быть сконструирован или присвоен элемент выходной последовательности.
- Те же требования по операциям инкремента, что и у входного итератора.
- Разыменование может сделать итератор не разыменуемым, так что его необходимо чередовать с инкрементом.

Требования этой концепции также минимальны, чтобы разрешить реализацию выходных итераторов, не указывающих на реально существующую в памяти последовательность объектов, а обрабатывающих результаты работы алгоритмов произвольным образом.

Концепция `InputIterator` имеет дальнейшие уточнения. Концепция `ForwardIterator` (однонаправленный итератор) требует:

- Уточняет `InputIterator` и `DefaultConstructible`. Результат инициализации по значению — послеконечный итератор некоторой пустой последовательности.
- Областью определения операций сравнения на (не)равенство являются итераторы над одной последовательностью (одной для всех итераторов, инициализированных по значению).
- Тип `reference` является настоящей ссылкой на соответствующий объект.
- Копии итераторов гарантировано не становятся недействительными при инкременте одного из них. Кроме этого, из `a == b` должно следовать `++a == ++b`, где `a` и `b` — два итератора.

Указанные выше свойства характерны для итераторов над действительной последовательностью объектов в памяти, которая за счёт своего физического существования может быть пройдена многократно — трюки, допустимые концепцией `InputIterator` в этой концепции, скорее всего, выражены быть не могут. Итераторы, удовлетворяющие этой концепции необходимы для работы ***многопроходных (multi-pass)*** алгоритмов, которым требуется несколько проходов по одной последовательности — гарантия сохранности копий итераторов при инкременте оригиналов позволяет это сделать.

Итераторы над последовательностями, по которым проход может осуществляться в оба направления за константное время на один шаг, соответствуют концепции `BidirectionalIterator` (двунаправленный итератор). Она уточняет `ForwardIterator`, а также требует аналогичных по семантике операций декремента, осуществляющих проход в обратном направлении. В том числе, требуется `--(++a) == a` и из `--a == --b` должно следовать `a == b`. Эти требования аналогичны таковым для `ForwardIterator`.

Когда итераторы, как, например, обычные указатели за счёт операций арифметики над ними, позволяют перемещаться по последовательности на любое количество шагов за константное время, они могут удовлетворять концепции `RandomAccessIterator` (итератор произвольного доступа). Она требует:

- Уточнение `BidirectionalIterator`.
- Операции `+=` и `-=` над левым операндом-итератором и правым операндом — целым числом сдвигают его на указанное число шагов в прямом или обратном направлении.
- Операции `+` и `-` делают то же, возвращая результат сдвига вместо модификации исходного итератора.
- Операция индексации соответствует стандартной семантике: `a[n] == *(a+n)`.
- Разность итераторов есть знаковое число, означающее число шагов для прохода от одного к другому, знак означает направление.
- Операции сравнения `<`, `>`, `<=` и `>=` для итераторов над одной последовательностью проверяют их относительный порядок.

Нетрудно убедиться, что обычные указатели на последовательность расположенных в памяти подряд объектов удовлетворяет этой концепции. Ключевым в ней является общее требование константной сложности всех операций — любой итератор можно сдвинуть вперёд на допустимое число шагов, но для итераторов, не удовлетворяющих `RandomAccessIterator`, сделать это можно только вызывая операцию инкремента в цикле, что повышает сложность до линейной.

> В новом стандарте языка планируется введение ещё более точной концепции `ContiguousIterator` (смежный итератор), соответствующей итераторам на объекты, расположенные в памяти последовательно. Её требования — адрес объекта, на который указывает итератор после сдвига, может также быть получен из адреса объекта, на который указывал изначальный итератор, аналогичным сдвигом за счёт арифметики указателей.

Итератор, удовлетворяющий `InputIterator` или более строгой концепции, может также удовлетворять и `OutputIterator`. Такой итератор позволяет осуществлять как чтение, так и запись объектов, на которые указывает, и называется ***модифицируемым (mutable)***, в отличие от ***неизменяемого (constant)***.

Все итераторы также должны задавать следующие типы в виде членов класса своей реализации (в специальных случаях, когда они бессмысленны, допустимо указывать `void`):

- `value_type` — тип объектов, на которые указывает итератор.
- `reference` — тип, возвращаемый операцией разыменования.
- `pointer` — тип указателя на объект, на который указывает итератор, обычно `value_type*`.
- `difference_type` — тип, в котором выражаются расстояния между итераторами, почти всегда `std::ptrdiff_t`, как и для обычных указателей.
- `iterator_category` — тип, обозначающий наиболее строгую концепцию, которой удовлетворяет итератор.

Наличие этих типов предоставляет информацию, необходимую для реализации обобщённых алгоритмов над различными итераторами.

С этой концепцией пока есть две проблемы: во-первых, не ясно как можно разместить «вложенные типы» внутри обычных указателей, которые по остальным характеристикам прекрасно удовлетворяют концепциям итераторов, а во вторых пока не было сказано какие типы можно указывать в качестве категории итераторов. Решим эти две проблемы в следующих разделах.

10.3.3. Черты

Требования наличия вложенных типов не могут технически быть выполнены категорией типов «указатель», которая, тем не менее, по остальным характеристикам удовлетворяет самой мощной концепции итератора для элементов, расположенных в памяти последовательно. Возможны и другие ситуации, когда необходимо приписать тому или иному типу дополнительные характеристики, но возможности это сделать нет, технически или организационно, например, когда речь идёт о классе из внешней библиотеки. В таких случаях можно применить механизм ***черт (traits)*** — шаблонов классов, параметризуемых исследуемыми типами и содержащими необходимую дополнительную информацию. Такие классы часто содержат в первичном шаблоне набор стандартных характеристик, которые можно уточнить для отдельных случаев с помощью механизма частичной специализации.

В стандартной библиотеке C++ свойства итераторов описываются шаблоном черт `std::iterator_traits`:

```
1  namespace std
2  {
3      template<typename Iterator>
4      struct iterator_traits
5      {
6          using value_type = typename Iterator::value_type;
7          using reference = typename Iterator::reference;
8          using pointer = typename Iterator::pointer;
9          using difference_type = typename Iterator::difference_type;
10         using iterator_category = typename Iterator::iterator_category;
11     };
12
13     template<typename T>
14     struct iterator_traits<T*>
15     {
16         using value_type = T;
17         using reference = T&;
18         using pointer = T*;
19         using difference_type = ptrdiff_t;
```

```
20          using iterator_category = random_access_iterator_tag;
21      };
22  }
```

Таким образом, частичная специализация для типов-указателей корректно указывает их характеристики, а первичный шаблон ссылается на соответствующие вложенные типы самого класса итератора. Теперь шаблоны алгоритмов, которым необходимо знать характеристики итераторов, могут получать их в общем виде. В современном C++ некоторые характеристики итераторов можно было бы выяснить в общем случае и без механизма черт, например, используя `decltype`, но в общем случае механизм черт может нести произвольную дополнительную информацию о типах, не доступную иначе.

10.3.4. Диспетчеризация по тегам. if constexpr

Диспетчеризация по тегам (tag dispatching) — приём, использующий вспомогательные классы вместо реальных обрабатываемых вместе с механизмом разрешения перегрузок. Под тегом понимают тип, являющийся «маркером» требуемого качества типа, к которому он относится. Сам тег обычно является пустым классовым типом.

Указанный выше в рассмотрении концепции итераторов вложенный тип `iterator_category` должен являться производным от одного из типов тегов категорий итераторов, имеющихся в стандартной библиотеке:

```
1  struct input_iterator_tag {};
2  struct output_iterator_tag {};
3  struct forward_iterator_tag : input_iterator_tag {};
4  struct bidirectional_iterator_tag : public forward_iterator_tag {};
5  struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Эта иерархия наследования пустых классов повторяет по структуре иерархию концепций итераторов. При реализации собственных итераторов следует в качестве вложенного типа `iterator_category` указать один из них.

В дальнейшем это может быть использовано алгоритмами для выбора наиболее эффективных реализаций из применимых к конкретному итератору. Рассмотрим в качестве примера алгоритм `std::distance`: он находит расстояние между двумя итераторами произвольной категории, уточняющей входную. Для итераторов произвольного доступа он может быть выполнен за константное время, т.к. для них он эквивалентен взятию их разности, входящей в концепцию. Для остальных итераторов потребуется цикл линейной сложности, осуществляющий пошаговый проход, и дополнительное требование, чтобы второй итератор был достижим из первого инкрементами. Выбор первой, более эффективной реализации для частного случая осуществляется диспетчеризацией по тегам:

```
1  // Для скрытия деталей реализации в классах есть уровень доступа private,
2  // вне них принято использовать вложенное пространство имён detail.
3  namespace detail
4  {
5      // Если тип тега произведён от тега итератора произвольного доступа, эта функция
6      // годна и лучше, чем следующая.
7      typename std::iterator_traits<Iterator>::difference_type distance_impl(
8              Iterator b,Iterator e,std::random_access_iterator_tag)
9      {
10         return e-b;
11     }
12
13     // Для более слабых итераторов эта функция единственная годная.
14     typename std::iterator_traits<Iterator>::difference_type distance_impl(
15             Iterator b,Iterator e,std::input_iterator_tag)
16     {
17         std::iterator_traits::difference_type d{};
18         while(b!=e){
19             ++b;
20             ++d;
21         }
22         return d;
23     }
```

```
}

typename std::iterator_traits<Iterator>::difference_type distance(Iterator b,Iterator e)
{
    // Вызов функции реализации с дополнительным параметром - объектом типа тега
    // для применения механизма перегрузок к иерархии концепций.
    return detail::distance_impl(b,e,
            typename std::iterator_traits<Iterator>::iterator_category());
}
```

В показанной реализации тип тега, действительно, используется только для создания временного объекта, тип которого используется для выбора одной из перегрузок-реализаций алгоритма. Подобное использование пустых классов легко будет оптимизировано компилятором без потери производительности на передачу лишних пустых аргументов. За счёт наследования тегов и механизма черт можно строить довольно сложные конструкции, модифицирующие механизм разрешения перегрузок под нужды программиста.

К сожалению, такой подход требует разбиение реализаций алгоритма на несколько функций и введение дополнительных вспомогательных, что не всегда удобно.

В C++17 появилась дополнительная форма оператора `if` с ключевым словом `constexpr` перед скобками вокруг контролирующего выражения. Такая форма требует константности контролирующего выражения, и выбор ветви определяется на этапе компиляции. Эту форму отличает то, что ветвь, не выбранная для компиляции, может содержать конструкции, успешно разбираемые, но не обязательно полностью синтаксически корректные, подобно не инстанциированным членам класса. Это позволяет записывать в одну функцию варианты реализации алгоритма, имеющие разные требования:

```
typename std::iterator_traits<Iterator>::difference_type distance(Iterator b,Iterator e)
{
    // Выясним, унаследован ли тег итератора от интересующего типа, метафункцией,
    // результат - константное выражение.
    if constexpr(std::is_base_of_v<std::random_access_iterator_tag,
            typename std::iterator_traits<Iterator>::iterator_category>)
        // Если условие выше ложно, выражение ниже не корректно,
        // (нет перегрузки operator-), но достаточно его синтаксической корректности.
        return e-b;
    else{
        std::iterator_traits::difference_type d{};
        while(b!=e){
            ++b;
            ++d;
        }
        return d;
    }
}
```

Единственная конструкция, сохраняющая силу в не выбранных ветвях `if constexpr` — утверждения времени компиляции.

Помимо рассмотренного алгоритма `std::distance`, в стандартной библиотеке есть аналогичные по действиям и реализации, полезные в обобщённых реализациях более сложных алгоритмов. Алгоритм `std::advance` продвигает итератор на заданное число шагов (знак определяет направление). Алгоритмы `std::next` / std::prev возвращают итератор, следующий или предшествующий данному, что удобно для применения к итераторам, возвращённым из других функций без сохранения их в промежуточный объект — операции инкремента концепции итераторов гарантируют применимыми только к леводопустимым значениям, как и стандартные операции.

10.3.5. Примеры итераторов

Приведём в качестве примера реализации бидерекционный итератор двусвязного списка с ограничителем.

```
// Узел двусвязного списка из элементов типа T.
template<typename T>
struct list_node
```

```
4  {
5      T data;
6      list_node *prev,*next;
7  };
8
9  // Модифицируемый итератор над двусвязным списком элементов типа T.
10 template<typename T>
11 class list_node_iterator
12 {
13 public:
14     // Связанные типы.
15     using value_type = T;
16     // Для удовлетворения ForwardIterator, тип reference - обычная ссылка.
17     using reference = T&;
18     using pointer = T*;
19     using difference_type = std::ptrdiff_t;
20     using iterator_category = /* ... */;
21
22     // Конструирование итератора из адреса узла, на который
23     // он должен указывать. Аргумент по умолчанию позволяет
24     // удовлетворить DefaultConstructible.
25     list_node_iterator(list_node<T>* node = nullptr)
26         : node(node)
27     {
28     }
29
30     // Все специальные функции используют реализации по умолчанию,
31     // удовлетворяя CopyConstructible, CopyAssignable и Destructible.
32
33     // Инкремент сдвигает внутренний укзаатель на следующий узел.
34     list_node_iterator& operator++()
35     {
36         node = node->next;
37         return *this;
38     }
39
40     // Декрмент, так как итератор двунаправленный.
41     list_node_iterator& operator--()
42     {
43         node = node->prev;
44         return *this;
45     }
46
47     // Стандартные реализации префиксных форм.
48     list_node_iterator operator++(int)
49     {
50         list_node_iterator old(*this);
51         ++*this;
52         return old;
53     }
54
55     list_node_iterator operator--(int)
56     {
57         list_node_iterator old(*this);
58         --*this;
59         return old;
60     }
61
62     // Операции сравнения сравнивают адреса узлов.
63     // Для итератора, сконструированного по умолчанию/значению
64     // сравниваются два пустых указателя, что даёт верный результат.
65     bool operator==(const list_node_iterator& other) const
66     {
67         return node==other.node;
68     }
69
```

```
70      bool operator!=(const list_node_iterator& other) const
71      {
72          return !(*this==other);
73      }
74
75      // Разыменование возвращает модифицируемую ссылку на данные,
76      // хранящиеся в узле, для удовлетворения ForwardIterator и
77      // OutputIterator.
78      // Как и с указателями, есть два уровня изменяемости: у итератора
79      // и самого значения. Изменяемость итератора не влияет на изменяемость
80      // значения, поэтому функция квалифицирована const.
81      reference operator*() const
82      {
83          return node->data;
84      }
85
86      pointer operator->() const
87      {
88          // Стандартная реализация для многих итераторов.
89          // *this - объект, на котором вызвана данная функция.
90          // **this - применение к нему operator* выше -
91          //          идентифицирует элемент на который указывает этот итератор.
92          // &**this - адрес этого элемента.
93          return &**this;
94      }
95  private:
96      list_node<T>* node;
97  };
98
99  // Swappable удовлетворяется стандартной реализацией std::swap.
```

В качестве более хитрого примера, показывающего возможности слабых концепций итераторов, приведём итератор, запись последовательности элементов в который приводит к их выводу по одному на строку.

```
1   #include <iostream>
2
3   // Итератор вывода в стандартный поток.
4   class cout_iterator
5   {
6   public:
7       // Этот итератор может вывести значения любого типа,
8       // которые можно вывести в std::cout, так что конкретного
9       // типа элемента нет, указателей нет по той же
10      // причине, понятия расстояния тоже нет!
11      using value_type = void;
12      using pointer = void;
13      using difference_type = void;
14      using iterator_category = std::output_iterator_tag;
15
16      // Инкремент должен быть реализован для удовлетворения
17      // OutputIterator, но делать в нём нечего.
18      cout_iterator& operator++()
19      {
20          return *this;
21      }
22
23      cout_iterator operator++(int)
24      {
25          return *this;
26      }
27
28      // В качестве резудьтата разыменования возвращается
29      // специальный объект, перехватывающий последующую
30      // операцию присваивания.
31
32      struct reference
```

```
33      {
34          // Шаблон операции присваивания, выводящий
35          // правый операнд в стандартный поток вывода.
36          template<typename T>
37          void operator=(const T& x)
38          {
39              std::cout << x << '\n';
40          }
41      };
42
43      reference operator*() const
44      {
45          return {};
46      }
47  };
48
49  template<typename OutputIterator>
50  void gen(OutputIterator it)
51  {
52      for(int i=1;i<=10;++i)
53          *it++ = i;
54  }
55
56  int main()
57  {
58      gen(cout_iterator());
59  }
```

10.3.6. Контейнеры

Итераторы предназначены для указания на элементы, однако они не решают вопроса владения множеством объектов. Для владения множеством объектов и организации их в ту или иную структуру данных предназначены ***Контейнеры (container)***. Идейно схожие классы, осуществляющие владение множеством объектов и предоставляющие доступ к ним, мы уже рассматривали, например, `int_vector`. Контейнеры стандартной библиотеки отличает то, что в качестве основного способа предоставления доступа к хранящимся в них элементах они используют итераторы. На элементы контейнера могут быть наложены требования (в первую очередь, по возможности их конструирования, копирования, перемещения и уничтожения), следующие из внутреннего устройства контейнера. В современном C++ накладывается минимум требований, исходя из тех операций, которые реально осуществляются над контейнером.

«Контейнер» также является именем концепции, в которой закреплены общие для них свойства. Здесь и далее автор будет указывать только имена сущностей без полного задания их описаний, поскольку это займёт очень много места и не даст сконцентрироваться на идеях — читателю рекомендуется проверять своё понимание по справочнику. Концепция «контейнер» требует:

- Классы контейнеров имеют следующие вложенные типы:
  - `value_type` — тип хранимых элементов;
  - `reference` и `const_reference` — типы ссылок на элементы, модифицируемые и не модифицируемые соответственно;
  - `iterator` и `const_iterator` — типы итераторов на модифицируемые и не модифицируемые элементы контейнера, итераторы контейнера как минимум однонаправленные, поскольку соответствуют реальным элементам в памяти;
  - `size_type` — тип, выражающий число элементов в контейнере и `difference_type` — тип расстояния между итераторами контейнера.

  Аналогично тому, что для типа указателей свойства модифицируемости самого объекта-указателя и объекта, на который он указывает независимы, наличие двух разных типов итераторов на модифицируемые и не модифицируемые элементы контейнера позволяет отличить модифицируемость элемента от модифицируемости самого итератора. Всегда существует неявное преобразование из типа итератора на модифицируемые элементы в тип итератора на не модифицируемые элементы, но не наоборот. С учётом этого, когда

итераторы используются только для указания положения в контейнере, а не для доступа к самим элементам, достаточно итераторов на не модифицируемые элементы.

- Контейнер, инициализированный по умолчанию, пуст.
- Контейнеры перемещаемы, а также копируемы, если тип их элемента копируем, конструированием и присваиванием. При этом может происходить как передача владения, так и поэлементное присваивание, в любом случае новое значение контейнера назначения становится эквивалентно значению контейнера-источника до операции в смысле числа элементов и их значений.
- Контейнеры владеют своими элементами и уничтожают их при уничтожении себя.
- Функции-члены `begin` и `end` возвращают начальный и послеконечный итераторы на все элементы контейнера, в рамках данной концепции порядок перечисления не указывается. Эти функции перегружены по модифицируемости неявного параметра объекта для возврата `iterator` или `const_iterator`, так что через не модифицируемое значение контейнера модификация его элементов не возможна. Для получения итераторов на неизменяемые объекты на изменяемом контейнере можно использовать вспомогательные функции-члены `cbegin` и `cend`.
- Если тип элементов контейнера удовлетворяет `EqualityComparable`, то контейнеры могут быть сравнены операциями `==` и `!=` поэлементно.
- Контейнеры удовлетворяют `Swappable` и в большинстве случаев выполняют обмен значений за константное время.
- Функция-член `size` возвращает число элементов в контейнере, а `empty` проверяет контейнер на отсутствие элементов (она может быть более эффективна, чем сравнение возвращаемого значения `size` с нулём).

Контейнеры, итераторы которых являются двунаправленными или более сильными, называются ***обратимыми (reversible)***. Они имеют вложенные типы `reverse_iterator` и `const_reverse_iterator`, которые задают типы обратных итераторов, осуществляющих перечисление элементов контейнера в порядке, обратном порядку для обычных итераторов. Получить такие итераторы на «последний» и «пред-первый» элементы контейнера можно функциями `rbegin` и `rend` соответственно. Функции `crbegin` и `crend` по аналогии возвращают итераторы на не модифицируемые элементы для контейнеров с любой модифицируемостью.

Как и в ситуации, когда нам потребовались шаблоны черт для решения проблемы с указателями, типы категории «массив» идейно похожи на контейнеры с фиксированного числом элементов. Для обобщения перечисления элементов в контейнерах и массивах стандартная библиотека имеет следующие свободные функции:

```
1  namespace std
2  {
3      template<typename Container>
4      auto begin(Container& c) -> decltype(c.begin())
5      {
6          return c.begin();
7      }
8
9      template<typename Container>
10     auto end(Container& c) -> decltype(c.end())
11     {
12         return c.end();
13     }
14
15     template<typename T,size_t N>
16     constexpr T* begin(T (&a)[N])
17     {
18         return a;
19     }
20
21     template<typename T,size_t N>
22     constexpr T* end(T (&a)[N])
23     {
24         return a+N;
25     }
26 }
```

Данные функции делегируют в общем случае результат одноимённой функции-члену класса, а для массивов возвращают указатели на начальный и послеконечный элементы. Аналогичным образом реализованы свободные функции для итераторов на неизменяемые элементы и обратных итераторов. Кроме этого, в виде свободных функций доступны `size` и `empty`, вычислимые для массивов во время компиляции (`constexpr`).

> Большинство контейнеров имеют переменный объём и, следовательно, должны работать с динамической памятью. Обычно для этого достаточно стандартных операций `new` и `delete`, но библиотека шаблонов C++ идёт дальше и абстрагирует процессы выделения памяти, а также создания и уничтожения объектов в отдельную концепцию `Allocator`. Контейнеры, позволяющие указывать способы работы с выделением памяти, удовлетворяют дополнительной концепции `AllocatorAwareContainer`. Оптимизация работы с памятью на таком уровне выходит за рамки этого текста, в котором мы будем игнорировать наличие у шаблонов контейнеров типового параметра `Allocator`. Он имеет значение по умолчанию в виде специализации шаблона класса `std::allocator` — реализации концепции `Allocator`, использующей обычные операции `new` / `delete`. Как ещё одно следствие, вместо обсуждения концепций, относящихся к свойствам специальных функций элементов контейнера (`DefaultInsertable` и подобные), в обсуждении требований к элементам будем использовать уже рассмотренные концепции, соответствующие им (`DefaultConstructible` и др.).

10.3.7. Последовательности

Последовательность (sequence) — концепция, уточняющая контейнер, обеспечивающая хранение элементов в указанном пользователем контейнера порядке. При вставке элементов в последовательность всегда указывается или подразумевается конкретное место, которое будет закреплено за элементом. Среди рассмотренных нами структур данных массивы и динмаические массивы, а также связанные списки являются последовательностями. Напротив, деревья поиска и хеш-таблицы сами определяют порядок элементов в своей структуре ради обеспечения требуемых характеристик операций над ними.

При вставке элементов в последовательность возможны следующие способы задания вставляемых элементов:

- Один конкретный элемент. Если он имеет категорию значения rvalue, он перемещается в контейнер, иначе — копируется.
- Число элементов и один элемент-образец. В контейнер вставляется указанное число копий данного элемента.
- Пара итераторов, указывающих на диапазон элементов, копии которых необходимо вставить. Обратим внимание, что это могут быть элементы произвольного контейнера, принципиально отличающегося по внутреннему устройства от данного. Кроме этого, тип элементов контейнеров не обязательно должен совпадать, достаточно наличия преобразования из типа элементов исходного контейнера к типу элементов данного.
- Списком инициализации. Вставляются копии его элементов.

Существуют следующие способы вставки элементов в последовательность:

- При конструировании, путём указания аргументов конструктору. Вставляемые элементы становятся начальным значением контейнера. В такой форме конструирование из единственного элемента невозможно, т.к. оно было бы по смыслу преобразованием элемента в контейнер, что не логично.
- Путём замены всего содержимого контейнера на указанные элементы. Такая замена выполняется одной из перегрузок функции-члена `assign`. У неё также нет перегрузки на случай одного элемента. Для замещения всех элементов контейнера содержимым списка инициализации также доступна перегрузка операции присваивания.
- Вставкой в указанное место. Вставка элемента или элементов производится функцией-членом `insert`, первый параметр которой — итератор на точку вставки, а остальные указывают на вставляемые элементы или элемент в одной из приведённых выше форм. При указании точек вставки итератор указывает на место, которое после вставки займёт первый из вставляемых элементов, или, другими словами, итератор на элемент «справа» от точки вставки.

- ***Размещением (emplacement)*** в указанном месте. Размещение — вставка в контейнер одного элемента, конструируемого прямо в требуемом конструктору месте по данному списку аргументов конструктора. При размещении гарантированно не происходит ни перемещения, ни копирования объекта, что может быть более эффективно, а также позволяет хранить в контейнерах, не требующих перемещения элементов при поддержании своей структуры (некоторых графовых), элементы, которые даже не перемещаемы. Размещение осуществляется функцией `emplace`, первый параметр которой — итератор на место вставки, а остальные — вариадические шаблонные, прозрачно передающиеся конструктору элемента.

Для удаления элементов концепция предоставляет следующие функции:

- Перегрузка `erase` с одним параметром-итератором удаляет указанный элемент;
- Перегрузка `erase` с двумя параметрами-итераторами удаляет элементы в указанном диапазоне;
- Функция `clear` без параметров удаляет все элементы контейнера.

Стандартная библиотека содержит пять шаблонов контейнеров-последовательностей. Укажем элементы их интерфейса, не соответствующие описанным концепциям, которые проявляются систематически:

- Последовательности, доступ к первому элементу которых возможен за константное время, позволяют получить ссылку на него вызовом `front`. Если то же возможно для последнего элемента, ссылка на него доступна через `back`. Эти ссылки модифицируемы так же, как и сам контейнер, за счёт перегрузки этих функций по изменяемости неявного параметра-объекта. Вызов их на пустом контейнере ведёт к неопределённому поведению.
- Последовательности, вставка в начало которых производится не больше, чем за амортизированное константное время, имеют функцию `push_front`, которая эквивалентна перегрузке `insert` для одного элемента, только место вставки не указывается — подразумевается начало последовательности. Также в этом случае имеется функция `emplace_front`, эквивалентная `emplace` по тому же принципу.

  Для последовательности, удаления элементов из начала которой так же эффективно, имеется функция `pop_front`, удаляющая первый элемент.

  Если операции вставки и удаления эффективны с конца последовательности, у последовательности присутствуют аналогичные функции `push_back`/`emplace_back` и `pop_back` соответственно.
- Контейнеры со смежными итераторами позволяют получить указатель на начало блока памяти, в котором хранят элементы, функцией `data` с const-корректными перегрузками. Такая функция существует и в виде свободной функции по рассмотренной выше схеме реализации, которая применима и к обычным массивам.
- Контейнеры с не фиксированным размером имеют функцию `resize` для установки нового количества элементов. При уменьшении размера удаляются элементы с конца, при расширении — вставляются новые, сконструированные по значению или копированием из указанного вторым параметров в другой перегрузке этой функции.
- Для большинства последовательностей определены операции порядка `<`, `>`, `<=` и `>=` между контейнерами одного типа, выполняющими сравнение в лексикографическом порядке.

Чаще всего применяется реализация динамического массива — шаблон класса `std::vector`. Он обладает всеми характеристиками, которые свойственны этой изученной нами структуре данных, но в этот раз мы имеем дело с библиотечной реализацией шаблона, пригодного для элементов различных типов, указываемых в виде его параметра.

Вектор является обратимым контейнером-последовательностью, а также предоставляет следующие возможности:

- Конструктор из числа элементов, конструируемых по значению.
- Управление ёмкостью. Для обеспечения амортизированной константной асимптотической сложности операции вставки элементов в конец, требуемой от динамического массива, вектор осуществляет геометрический рост используемого блока памяти. Его текущий размер в элементах возвращает функций-член `capacity`.

  Выделить заранее требуемое место можно, указав его функции `reserve`. Это рекомендуется делать перед вставкой заведомо известного числа элементов.

- Доступ к элементам по индексам через перегрузку `operator[]` — это один из наиболее естественных способов работы с вектором. Как и с обычным массивом, обращение к не существующим элементам вызывает неопределённое поведение.

Рассмотрим пример его использования, как одного из наиболее часто применяемых контейнеров:

```
1  // Здесь нужен пример.
```

`std::list` — обратимая последовательность, реализованная в виде двусвязного списка. Она интересна тем, что имеет набор перегрузок функции `splice`, позволяющей переносить в данный список элемент или цепочки элементов из другого такого же списка за константное время.

`std::forward_list` — последовательность-односвязанный список. Для односвязанного списка не годится указание мест операций вставки/удаления через итератор на это место, поскольку это не позволит провести операция за константное время. Вместо соответствующих операций реализация односвязанного списка имеет члены `before_begin`, `insert_after`, `erase_after` и другие с тем же суффиксом, которые работают с итератором на позицию, предшествующую требуемой.

`std::deque` — ***двусторонняя очередь (double-ended queue)***. Вопреки названию, она имеет итераторы произвольного доступа, т.е. обладает более широким набором эффективных операций, чем вектор. Вместо этого она теряет свойство смежности на итераторах и его операции произвольного доступа имеют бóльшие накладные расходы, не учитываемые в асимптотической нотации. Внутреннее устройство этой структуры данных может быть разным.

Наконец, `std::array` — попытка придать обычному массиву все возможные характеристики, свойственные контейнерам. Как и у обычного массива, его ёмкость фиксирована и определяется на этапе компиляции вторым параметром шаблона. Он не имеет членов, связанны со вставкой и удалением элементов, а также лексикографических сравнений. В отличие от обычного массива он, однако, допускает нулевое число элементов и не подвержен разложению в указатель, что иногда полезно.

10.3.8. Строки

Пока см. `http://en.cppreference.com/w/cpp/header/string`

10.3.9. Цикл for для диапазона

При перечислении всех элементов контейнера типичной конструкцией является:

```
1  Container c = {...};
2  // ...
3  for(auto i=c.begin(),e=c.end();i!=e;++i){
4      // Очередной элемент - это *i.
5  }
```

В этом примере показан типичный приём кеширования конечного итератора, что эффективнее записи `i!=c.end()`.

Поскольку это очень часто встречающаяся задача, в современном языке C++ имеется специальный вид цикла `for`, упрощающий перечисление все элементов некоторого контейнера, называемый ***цикл for для диапазона (range-based for)***:

`for (` описание `:` выражение `)` оператор
`for (` описание `:` список-инициализации `)` оператор

Описание в этой форме должны содержать ровно один описатель. Вторая форма требуется формально, поскольку список инициализации сам по себе не является выражением. Такой цикл соответствует по поведению следующему фрагменту

```
1  {
2      // Эта строка одинаково работает для обеих форм.
3      auto&& __range = выражение или список-инициализации;
4      auto __begin = начало;
5      auto __end = конец;
6      for(;__begin!=__end;++begin){
```

```
7           описание = *__begin;
8           оператор
9       }
10  }
```

Таким образом компилятор после захвата выражения в правой части цикла вычисляет итераторы начала и конца диапазона, и проходит его в стандартной форме. В теле цикла описание в форме из заголовка цикла инициализирует на каждой итерации объект значением объекта, на который указывает итератор. Для вычисления концов диапазона `начало` и `конец` применяется алгоритм, аналогичный работе свободных функций `begin` и `end`.

Таким образом, перечислить все элементы контейнера становится совсем просто и удобно, независимо от его типа:

```
1   #include <iostream>
2   #include <list>
3
4   int main()
5   {
6       // Работает с массивами
7       int array[] = {1,2,3,4,5};
8       for(int x:array)
9           std::cout << x << '\n';
10
11      // Работает с контейнерами.
12      // Описание ссылки для изменения элементов
13      // контейнера, а не временного объекта из
14      // скрытой реализации цикла.
15      // Увеличивает все элементы списка на 1.
16      std::list<int> l = {1,2,3,4,5};
17      for(int& x:l)
18          ++x;
19  }
```

В этой версии цикла `for` в левой части может быть только определение нового объекта или ссылки, выражение не допустимо. Разумеется, в сложной логике цикла, когда нужен доступ к итераторам, или нужно обработать не весь контейнер, приходится возвращаться к форме цикла из начала этого раздела.

10.3.10. Функциональные объекты. Лямбда-выражения

В языке C++ за счёт возможности перегрузки операции вызова функции для классовых типов не только функции поддерживают такой синтаксис. Концепции `FunctionObject` удовлетворяют типы объектов, к которым применима операция вызова функции. Такие объекты также для краткости называют ***функторами (functor)***. Хотя типы функций и ссылок на них не являются типами объектов, они разлагаются в указатели на функции, которые этой концепции удовлетворяют.

Имеется дальнейшая иерархия концепций по типам параметров и возвращаемому значению функциональных объектов:

- Концепции `Predicate` соответствует функциональный объект, вызываемый с одним параметром и возвращающий значение, приводимое к `bool`. Такие функциональные объекты проверяют объекты на принадлежность некоторому множеству.
- Концепции `BinaryPredicate` соответствует функциональный объект, вызываемый с двумя параметрами и возвращающий значение, приводимое к `bool`. Такие функциональные объекты проверяют некоторое отношение между двумя данными объектами.
- Концепции `Compare` удовлетворяет функциональный объект, удовлетворяющий `BinaryPredicate` и устанавливающий отношение ***строгого слабого порядка (strict weak ordering)*** на множестве объектов. Этот функциональный объект в случаях, когда необходимо указание порядка среди элементов, например, для сортировки. Его возвращаемое значение должно быть истинно, когда первый из его аргументов должен быть расположен перед вторым и ложным, если верно обратное утверждение или порядок этих двух элементов, считающихся эквивалентными, не важен. Формально, такой объект `comp` должен удовлетворять свойствам иррефлексивности (`comp(a,a)==false`), антикоммутативности (из `comp(a,b)` следует `!comp(b,a)`) и транзитивности (из `comp(a,b)` и `comp(b,c)` следует `comp(a,c)`).

Похожие идеи и требования нам встречались при рассмотрении функций сортировки и двоичного поиска стандартной библиотеки языка C, где мы столкнулись с двумя основными проблемами при задании функциональных объектов в программе. Во-первых, написание функционального объекта требует написание функции, которое не может быть размещено в другой. Таким образом, логика функтора находится в тексте программы потенциально далеко и, скорее всего, перед местом его использования, что затрудняет чтение. Во-вторых, если требуемый функтор зависит не только от значений своих параметров, то дополнительные данные приходилось хранить в глобальных переменных, что неудобно и может приводить к проблемам в сложных случаях.

В C++ имеется несколько способов решения этих проблем.

В простых случаях, когда требуется функциональный объект, соответствующий в точности одной из операций языка, можно воспользоваться готовыми реализациями из заголовочного файла `<functional>`. Приведём пример упрощённого описания одного из таких объектов:

```
template<typename T = void>
struct plus
{
    T operator()(const T& x,const T& y) const
    {
        return x+y;
    }
};

template<>
struct plus<void>
{
    template<typename T,typename U>
    decltype(auto) operator()(T&& x,T&& y)
    {
        return std::forward<T>(x)+std::forward<T>(y);
    }
};
```

В примере показано, что специализации функционального объекта `plus` для конкретного типа объектов определяют операция вызова функции, возвращающую по значению сумму двух значений, принятых по леводопустимым ссылкам не неизменяемое значение. Этот первичный шаблон не использует прозрачной передачи, поскольку был введён в стандартную библиотеку до появления в языке средств, позволяющих её осуществить. Эти категории значений используются всеми первичными шаблонами функторов, возвращаемое значение имеет тот же тип, что и аргументы для арифметических операций, и `bool` для операций сравнения и логических операций.

Явная специализация для `T = void` введена в новых версиях языка и является универсальной версией функционального объекта для аргументов любых типов. Она использует прозрачную передачу аргументов и возвращаемого значения. За счёт этого новая версия удобнее, корректнее и в большинстве случаев предпочтительнее первичного шаблона.

Перечислим имена всех имеющихся функторов, из имён которых легко понять, каким операциям они соответствуют: `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`, `bit_and`, `bit_or`, `bit_xor`, `bit_not`, `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`, `logical_and`, `logical_or`, `logical_not`.

Поскольку эти функторы не содержат членов данных, их не накладно создавать как временные объекты непосредственно на месте требования:

```
template<typename ForwardIterator,typename BinaryPredicate>
void algorithm(ForwardIterator from,ForwardIterator to,BinaryPredicate predicate);

void f()
{
    std::vector v = {1,2,3};
    // Некоторый алгоритм над последовательностью с
    // предикатом над парой элементов, соответствующим операции >.
    algorithm(v.begin(),v.end(),std::greater<>());
}
```

Когда необходим более сложный функтор, готовыми отделаться не удастся. Чтобы решить указанные нами проблемы при использовании в качестве функторов обычных функций, в языке

C++ применяют ***лямбда-выражения (lambda)***. Эти выражения — краткий способ записи значения некоторого классового типа, являющегося функциональным объектом. Этот тип называют ***замыканием (closure)***. Все стандартные части такого определения компилятор генерирует самостоятельно, избавляя от необходимости определять вспомогательные классы вручную.

Простейшее лямбда-выражение выглядит как пара квадратных скобок, за которой следует блок:

```
1  void f()
2  {
3      auto f1 = [] { return 42; };
4      int x = f1(); // x == 42
5  }
```

Тип лямбда-выражения соответствует классу, автоматически созданному компилятором и описанному в ближайшей окружающей области видимости, где это допустимо (в данном случае это будет локальный класс функции `f`), и не может быть записан явно, поэтому приходится применить заполнитель `auto`, чтобы сохранить этот функтор в именованный объект. Определения специальных функций-конструкторов копирования и перемещения в нём даны по умолчанию, операция присваивания копированием удалена. Он также имеет перегруженную операцию вызова функции без параметров над неизменяемым параметром-объектом с заданным телом и типом возвращаемого значения `auto`. Кроме этого, такое лямбда-выражение имеет операцию неявного приведения к типу указателя на функцию с той же сигнатурой, что и перегруженная операция вызова функции. Это бывает полезно для использования лямбда-выражений в контекстах, где требуется именно такой указатель, а не полноценный объект — например, для использования в старых функциях из стандартной библиотеки языка C. Теоретически это возможно, поскольку перегруженная операция в данном случае не использует неявный параметр-объект и является по смыслу статическим членом класса. Таким образом этот пример эквивалентен следующему:

```
1  void f()
2  {
3      // При упоминании типов замыканий, например,
4      // в сообщениях об ошибках, компилятор обычно идентифицирует
5      // их как "лямбда на строке/столбце таких-то".
6      // В объектных файлах их декорированные имена могут быть весьма странными -
7      // автор не сошёл с ума и помнит, что $ - недопустимый символ в идентификаторе.
8      // (Если, конечно, вы не пользуетесь нестандартными расширениями компилятора.)
9      class $_1
10     {
11     public:
12         $_1(const $_1&) = default;
13         $_1($_1&&) = default;
14         $_1& operator=(const $_1&) = delete;
15
16         auto operator()() const
17         {
18             return 42;
19         }
20
21         // Синтаксис записи описаний операций приведения типов
22         // не позволяет напрямую записать приведение к типу
23         // указателя на функцию, поэтому в нашей записи
24         // потребуется псевдоним типов.
25         using __func_type = int (*)();
26
27         // Операция приведения к типу указателя на функцию.
28         operator __func_type() const
29         {
30             // Компилятор знает, как это написать.
31             // Например, сгенерировать ещё одну копию
32             // функции с данным телом в виде статического
33             // члена класса, и вернуть его адрес.
34             // Скорее всего, он может это сделать эффективнее,
35             // чем полное дублирование перегруженной операции.
```

```
36          }
37      };
38      auto f1 = /* как-то созданный компилятором объект типа $_1 */;
39      int x = f1(); // x == 42
40  }
```

Чтобы указать список параметров функтора, его можно дать между квадратными скобками и телом. Также после списка параметров можно указать хвостовой возвращаемый тип, если обычный `auto` не годится (включая другой вариант, тоже содержащий заполнители):

```
1   void f()
2   {
3       // Показаны только описания перегруженной операции вызова функции.
4
5       // auto operator()(int x) const;
6       auto f1 = [](int x) { return !x; };
7
8       // auto operator()(int &x) const -> auto&&;
9       // (тип возвращемого значения дедуцируется в int&&)
10      auto f2 = [](int& x) -> auto&& { return std::move(x); };
11  }
```

Наиболее полезным свойством лямбда-выражений является то, что поиск имён в них реализован так, словно блок, являющийся их телом — обычной блок, расположенный там, где он и записан. Хотя обычный локальный класс не имеет доступа к видимым из него объектам с автоматическим временем хранения, это часто бывает нужно для лямбда-выражения, чтобы ***захватить (capture)*** требуемые для её функционирования значения из локальных переменных функции, в которой оно описано. Это требует введения дополнительных объектов и не выполняется автоматически — при попытке именовать такие объекты в теле лямбда-выражения без дополнительных указаний, компилятор сообщит о том что они «не захвачены». Список захватываемых выражений указывается внутри квадратных скобках в её синтаксисе. Это разделённый запятыми список, который может содержать следующие конструкции:

- `идентфиикатор` — явный захват по значению. В тип замыкания добавляется член данных с тем же именем и того же типа, что и видимый из неё объект с автоматическим временем хранения, идентифицируемый заданным именем. Когда создаётся объект типа этого класса, этот член данных будет инициализирован копированием из соответствующего автоматического объекта. Поиск имён в лямбда-выражении теперь находит эту копию, вместо настоящего объекта.
- `&идентификатор` — явный захват по ссылке. В тип замыкания добавляется член данных с тем же именем что и видимый из неё объект с автоматическим временем хранения, идентифицируемый заданным именем. Тип этого члена — ссылка на тип захватываемого объекта, которая инициализируется привязкой к ней соответствующего автоматического объекта. Поиск имён в лямбда-выражении теперь находит эту ссылку, вместо настоящего объекта. Хотя операция вызова функции на типе замыкания обычно имеет квалификатор параметра-объекта const, в её теле такие ссылки его не получают и считаются полноценными именами захваченных объектов. Следует быть аккуратным, если объект лямбда-выражения переживёт время жизни захваченного по ссылке автоматического объекта, ссылка станет висячей.
- `this` — захват указателя на неявный параметр-объект. Захватывает по значению указатель на неявный параметр-объект не статической функции-члена класса, в которой записано лямбда выражение. Ключевое слово `this` в теле лямбда-выражения теперь именует этот указатель на объект, это же значение используется и в тех случаях, когда использование `this` подразумевается поиском имён неявно. С этим захватом или без, `this` в теле лямбда выражения нельзя использовать для доступа к самому объекту типа замыкания.
- `*this` — захват значения неявного параметра-объекта. Также применимо только если лямбда выражение находится в теле нестатической функции-члена класса. В тип замыкания добавляется член данных того же типа, что и класс, в котором используется лямбда выражение. При создании объекта типа замыкания данный член инициализируется копированием из неявного параметра-обхъекта окружающей функции, и все использования `this` в этом лямбда-выражении соответствуют адресу этой копии.

- `=` — неявный захват по значению, должен быть первым в списке. Захватывает все используемые в теле лямбда-выражения объекты с автоматическим временем хранения по значению без явного перечисления их имён, включая `this`, если это нужно. Явные указания захвата по значению после него не допустимы. Применяется, когда нужно захватить много объектов и не хочется перечислять все их имена. Если бóльшую часть объектов нужно захватить по значению, но некоторые — по ссылке, исключения можно перечислить после `=` в форме `&идентификатор`.
- `&` — неявный захват по ссылке, должен быть первым в списке. То же, что и `=`, но захватывает всё используемое по ссылке. `this`, если используется, всё равно захватывается как указатель по значению. Явные указания захвата по ссылке после него не допустимы. Исключения для захвата по значению могут быть перечислены после в форме `идентификатор`.
- `идентификатор инициализатор` или `&идентификатор инициализатор` — обобщённый захват. Такой захват позволяет создать в типе замыкания произвольные члены данных с произвольными начальными значениями. Такой захват ведёт себя, словно к выражению захвата слева приписывает `auto`, а справа — `;` и такое определение включается в класс замыкания (хотя обычно определения с заполнителями в классовой области видимости недопустимы). Имя идентификатора при этом может не соответствовать никаким ранее описанным объектам. Этот синтаксис часто используют для захвата по значению с перемещением в виде `x = std::move(x)` — в член данных типа замыкания с именем `x` перемещают содержимое локального одноимённого объекта.

Замыкания, захватывающие хотя бы один объект любым образом уже не имеют операции приведения к типу указателя на функцию.

Поскольку операция вызова функции на лямбда-выражении по умолчанию имеет квалификатор параметра-объекта `const`, она не может изменить свои члены данных, захваченные по значению. Если этот квалификатор нужно отменить по этой или другой причине, то можно указать ключевое слово `mutable` после списка параметров лямбда выражения.

Приведём пример использования замыкания для задания нестандартной функции сортировки. Стандартная библиотека C++ содержит реализацию алгоритма сортировки:

```
#include <algorithm>

namespace std
{
    // Сортировка элементов из [first;last) в порядке, задаваемом операций <.
    template<typename RandomAccessIterator>
    void sort(RandomAccessIterator first,RandomAccessIterator last);

    // Сортировка элементов из [first;last) в порядке, задаваемом функтором comp.
    template<typename RandomAccessIterator,typename Compare>
    void sort(RandomAccessIterator first,RandomAccessIterator last,Compare comp);
}
```

Отсортируем целые числа в векторе по возрастанию запрошенного у пользователя десятичного разряда, подсчитав точное число вызовов нашего функтора реализацией библиотечной функции сортировки:

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

int main()
{
    std::vector<unsigned> v = {123,456,789,444,109,818,356,725,514,31};
    unsigned digit;
    std::cout << "Enter digit number in [0:2]: ";
    if((std::cin>>digit)&&digit<=2){
        std::size_t compares = 0;
        std::sort(v.begin(),v.end(),[&compares,factor=
                static_cast<unsigned>(std::pow(10,digit))]
            (unsigned a,unsigned b){
                ++compares;
                return (a/factor)%10 < (b/factor)%10;
```

```
18              });
19          for(int x:v)
20              std::cout << x << '\n';
21          std::cout << "Compares done: " << compares << '\n';
22      }
23  }
24
25  #if 0
26      Тип замыкания в этом примере:
27          class $_1
28          {
29          public:
30              $_1(const $_1&) = default;
31              $_1($_1&&) = default;
32              $_1& operator=(const $_1&) = delete;
33
34              auto operator()(unsigned a,unsigned b) const
35              {
36                  ++compares;
37                  return (a/factor)%10 < (b/factor)%10;
38              }
39          private:
40              // Захваченный по ссылке объект compares, инициализируется привязкой
41              // объекта с автоматическим временем хранения compares из тела
42              // функции main при создании лямбда выражения.
43              // Может изменяться в const-операции вызова функции, хотя
44              // обычно в C++ такое сделать нельзя, т.к. mutable к ссылкам не применим.
45              /* mutable? */ size_t& compares;
46
47              // Обобщённый захват. digit в выражении берётся с точки зрения
48              // функции main на момент создания объекта лямбда-выражения.
49              auto factor = static_cast<unsigned>(std::pow(10,digit));
50          };
51  #endif
```

Наконец, лямбда-выражения — единственный случай, когда в типе параметра функции может применяться заполнитель `auto`. В таком случае вместо операции вызова функции у типа замыкания определяется шаблон такой операции, и для каждого вхождения заполнителя `auto` создаются соответствующие типовые параметры шаблона — такие лямбда-выражения называют ***обобщёнными (generic)***. При использовании этого заполнения можно использовать пачки параметров функции для создания вариадических обобщённых лямбда-выражений:

```
1  // Вариадическое обобщённое лямбда выражение, вызывающее свой первый
2  // аргумент-функтор с остальными аргументами и прозрачной передачей
3  // всех аргументов и возвращаемых значений.
4  auto pf = [](auto&& f,auto&&... args) -> decltype(auto) {
5      return std::forward<decltype(f)>(f)(std::forward<decltype(args)>(args)...);
6  };
7
8  // Тип-замыкание имеет следующее описание шаблона операции вызова функции:
9  // template<typename T1,typename... T2>
10 // auto operator()(T1&& f,T2&&... args) const -> decltype(auto);
```

10.3.11. Алгоритмы

Пока см. `http://en.cppreference.com/w/cpp/algorithm`

10.3.12. Ассоциативные контейнеры

При рассмотрении последовательностей мы убрали из рассмотрения контейнеры, основанные на деревьях поиска и хэш-таблицах. Эти контейнеры относят к ***ассоциативным (associative)***, поскольку они предназначены для быстрого поиска значений, ассоциированных с искомыми, называемыми ключём. Тип ключа задаётся параметром шаблона контейнера `Key`, который становится вложенным типом класса шаблона под именем `key_type`. Всего в

стандартной библиотеке языка C++ имеется 8 шаблонов классов ассоциативных контейнеров, которые соответствуют всем возможным комбинациям трёх характеристик.

Если интерес представляет только наличие элемента типа ключа в контейнере без связи с ним дополнительных данных, применяются ***простые (simple)*** ассоциативные контейнеры. В них тип элемента контейнера совпадает с ключом.

Если дополнительные данные имеются, их тип указывают параметром шаблона `T`, отражаемый вложенным типом контейнера `mapped_type`. В таком случае тип элементов контейнера — `std::pair<const Key,T>`, а контейнер называют ***парным (paired)***. Запрет модификации ключа в паре запрещает произвольное изменения ключа у элементов контейнера по модифицируемой ссылке на них, что без ведома контейнера привело бы к нарушению его инвариантов.

По способу внутренней организации ассоциативные контейнеры делятся на ***сортированные (sorted)*** и ***неупорядоченные (unordered)***.

Элементы сортированных контейнеров упорядочены по критерию, задаваемому бинарным предикатом `Compare`, тип которого является одним из параметров шаблона класса. По умолчанию сортированные контейнеры используют специализацию `std::less`. Наличие порядка обеспечивает логарифмическую сложность операций поиска, вставки и удаления. Стандартной реализацией такого контейнера является один из видов дерева двоичного поиска.

Элементы неупорядоченных контейнеров хранятся в хэш-таблице, что и определяет её скоростные характеристики. Для устройства хэш-таблицы необходимо указание типов двух функторов: `Hash` — функции хэширования и `Pred` — отношения равенства. Хэш-функции для стандартной библиотеки должны возврщать значения типа `std::size_t` в его полном диапазоне, приведение к меняющемуся числу ячеек хэш-таблицы контейнер осуществляет самостоятельно. По умолчанию используется специализация шаблона класса `std::hash`, имеющего в стандартной библиотеке готовые специализации для арифметических типов и указателей. Этот функтор — пример, когда разрешена специализация библиотечного шаблона для пользовательских типов, если они хотят предоставить стандартное определение собственной хеш-функции. Для предиката сравнения по умолчанию используется `std::equal_to`.

Наконец, по допустимости наличия нескольких элементов с равными ключами, ассоциативные контейнеры делят на ***одиночные (single)*** и ***множественные (multi)***.

Имена ассоциативных контейнеров стандартной библиотеки формируются по следующим правилам:

1. Приставка `unordered_` для неупорядоченных контейнеров;
2. Приставка `multi` для множественных контейнеров;
3. Корень `set` для простых контейнеров или `map` для парных.

Например, `unordered_map` — неупорядоченный, одиночный, парный ассоциативный контейнер.

Далее см.

- `http://en.cppreference.com/w/cpp/container/set`
- `http://en.cppreference.com/w/cpp/container/map`
- `http://en.cppreference.com/w/cpp/container/multiset`
- `http://en.cppreference.com/w/cpp/container/multimap`
- `http://en.cppreference.com/w/cpp/container/unordered_set`
- `http://en.cppreference.com/w/cpp/container/unordered_map`
- `http://en.cppreference.com/w/cpp/container/unordered_multiset`
- `http://en.cppreference.com/w/cpp/container/unordered_multimap`

10.4. Обработка ошибок

10.4.1. Исключения

Глава 11

Стандартная библиотека языка C++

Мы уже рассмотрели стандартную библиотеку шаблонов и некоторые другие части стандартной библиотеки языка C++. В этой части мы познакомимся с её разделами, которые требуют знания большей части языка.

11.1. Потоки ввода-вывода

11.1.1. Задачи на потоки ввода-вывода

Глава 12

Язык C

Глава 13

Использование библиотек

13.1. Пример использования libusb

Данный пример показывает использования библиотеки libusb для перечисления имён всех USB-устройств, подключённых к компьютеру.

Для получения строк необходимо устанавливать соединения с устройствами и отправлять им запросы, для чего необходимо запускать программу от лица суперпользователя или другого уполномоченного. Если получить доступ не удаётся, выводятся только числовые идентификаторы производитель/модель.

Также данный пример показывает, как оборачивать в RAII-классы ресурсы, предоставляемые библиотеками на языке C.

```
# usb-enum.pro

TEMPLATE = app
CONFIG += console c++1z strict_c++
CONFIG -= app_bundle
CONFIG -= qt
gcc:QMAKE_CXXFLAGS += -pedantic-errors

gcc:QMAKE_CXXFLAGS_DEBUG += -fsanitize=address,undefined
gcc:QMAKE_LFLAGS_DEBUG += -fsanitize=address,undefined

SOURCES += main.cpp

# Установка всех опций, требуемых для работы библиотеки,
# через pkgconfig (последняя опция мастера Add Library...)
CONFIG += link_pkgconfig
PKGCONFIG += libusb-1.0
```

```
1  // main.cpp
2
3  #include <algorithm>
4  #include <cassert>
5  #include <iomanip>
6  #include <iostream>
7  #include <stdexcept>
8  #include <string>
9  #include <utility>
10 #include <vector>
11
12 #include <libusb.h>
13
14 // Устройство USB - только данные, идентифицирующие устройство,
15 // имеющие собственный счётчик разделяемого владения.
16 // Обёртываем в класс, сохраняя семантику - этот класс будет
17 // копируем и перемещаем, поскольку даже копирование не копирует
18 // "устройство", а только увеличивает число его владельцев,
19 // по той же причине конструктор не explicit.
20 // Вместе с самим устройством храним дескриптор - информацию
```

```
21  // об устройстве, которая известна без подключения к нему,
22  // чтобы не запрашивать каждый раз.
23  class usb_device
24  {
25  public:
26      usb_device(libusb_device* device = nullptr)
27          : device_(device)
28      {
29          if(device_){
30              // Увеличить счётчик владельцев.
31              libusb_ref_device(device_);
32              // Получить информацию из дескриптора.
33              libusb_get_device_descriptor(device,&desc);
34          }
35      }
36
37      ~usb_device()
38      {
39          if(device_)
40              // Уменьшить счётчик владельцев, это уничтожит
41              // устройство, если оно больше не никому не нужно.
42              libusb_unref_device(device_);
43      }
44
45      usb_device(const usb_device& other)
46          : usb_device(other.device_)
47      {}
48
49      usb_device(usb_device&& other)
50          : device_(std::exchange(other.device_,{})),
51            desc(other.desc)
52      {}
53
54      usb_device& operator=(const usb_device& other)
55      {
56          if(this!=&other){
57              this->~usb_device();
58              device_ = other.device_;
59              libusb_ref_device(device_);
60              desc = other.desc;
61          }
62          return *this;
63      }
64
65      usb_device& operator=(usb_device&& other)
66      {
67          assert(this!=&other);
68          this->~usb_device();
69          device_ = std::exchange(other.device_,{});
70          desc = other.desc;
71          return *this;
72      }
73
74
75      libusb_device* device() const
76      {
77          return device_;
78      }
79
80      const libusb_device_descriptor& descriptor() const
81      {
82          return desc;
83      }
84  private:
85      libusb_device* device_;
86      libusb_device_descriptor desc;
```

```
87  };
88
89  // Класс контекста libusb, необходимого для работы с библиотекой.
90  class usb_context
91  {
92  public:
93      usb_context()
94      {
95          if(libusb_init(&context))
96              throw std::runtime_error("Failed to init libusb");
97      }
98
99      ~usb_context()
100     {
101         if(context)
102             libusb_exit(context);
103     }
104
105     // Конструктор перемещения для передачи владения между функциями.
106     usb_context(usb_context&& other)
107         : context(std::exchange(other.context,{}))
108     {}
109
110     usb_context& operator=(const usb_context&) = delete;
111
112     // Вернуть вектор имеющихся устройств.
113     std::vector<usb_device> devices() const
114     {
115         std::vector<usb_device> res;
116         libusb_device** list;
117         ssize_t count = libusb_get_device_list(context,&list);
118         if(count<0)
119             throw std::runtime_error("Failed to get list of devices");
120         res.reserve(count);
121         std::copy(list,list+count,std::back_inserter(res));
122         // Освободить список и уменьшить число пользователей на каждый элемент.
123         // Наши объекты usb_device, созданные выше,
124         // уже увеличили счётчик пользователей и ресурсы самих
125         // устройств останутся в их владении.
126         libusb_free_device_list(list,true);
127         return res;
128     }
129 private:
130     libusb_context* context;
131 };
132
133 // Устройство, с которым установлен связь и можно взаимодействовать.
134 // Такой ресурс обеспечивает эксклюзивное пользование устройством и
135 // разделяемого владения сам по себе не предполагает.
136 // Как и многие другие непрозрачные типы, идентифицирующие ресурсы,
137 // называется "handle" - в смысле, вот "ручка", держась за которую
138 // объектом, частью которого она является, можно управлять.
139 class usb_device_handle
140 {
141 public:
142     // Передача по значению для сохранения в член класса.
143     usb_device_handle(usb_device ud)
144         : ud(std::move(ud))
145     {
146         if(libusb_open(ud.device(),&handle))
147             // Скорее всего, не хватает прав.
148             throw std::runtime_error("Failed to open device");
149     }
150
151     ~usb_device_handle()
152     {
```

```
153         if(handle)
154             libusb_close(handle);
155         // Деструктор
156     }
157
158     // Конструктор перемещения для передачи владения между функциями.
159     usb_device_handle(usb_device_handle&& other)
160         : udd(std::move(other.ud)),
161           handle(std::exchange(other.handle,{}))
162     {}
163
164     usb_device_handle& operator=(const usb_device_handle&) = delete;
165
166     std::string manufacturer() const
167     {
168         return get_string(ud.descriptor().iManufacturer);
169     }
170
171     std::string product() const
172     {
173         return get_string(ud.descriptor().iProduct);
174     }
175 private:
176     usb_device ud;
177     libusb_device_handle* handle;
178
179     std::string get_string(uint8_t id) const
180     {
181         std::string s(256,'\0');
182         // libusb_get_string_descriptor_ascii работает с символами unsigned char,
183         // а std::string - просто char. Узкие символьные типы могут рассматриваться
184         // как другие такие же за счёт понятия представления объектов, приведение
185         // типов правил псевдонимов не нарушает.
186         int count = libusb_get_string_descriptor_ascii(handle,id,
187             reinterpret_cast<unsigned char*>(s.data()),s.size());
188         if(count<0)
189             return {};
190         s.resize(count);
191         return s;
192     }
193 };
194
195 int main()
196 {
197     try{
198         usb_context ctx;
199         std::cout << std::hex << std::fill('0');
200         for(auto&& ud:ctx.devices()){
201             auto& desc = udd.descriptor();
202             std::cout << "Vendor " << std::setw(4) << desc.idVendor
203                       << " Product " << std::setw(4) << desc.idProduct;
204             try{
205                 usb_device_handle h(ud);
206                 std::cout << " : " << h.manufacturer() << ' ' << h.product();
207             }
208             catch(std::runtime_error& e){
209                 std::cout << e.what();
210             }
211             std::cout << '\n';
212         }
213     }
214     catch(std::exception& e){
215         std::cerr << e.what() << '\n';
216         return 1;
217     }
218 }
```

Глава 14

Построение графического интерфейса пользователя с использованием библиотеки Qt

14.1. Итоговый проект

14.1.1. Общие рекомендации

- Проверьте список файлов, установленных пакетом в виртуальной машине по команде `equery files имя-пакета`. Имя пакета указано в скобках после имени проекта, если отличается от него. Если он включает файлы в `/usr/lib64/pkgconfig`, то можно использовать pkgconfig для настройки проекта на работу с библиотекой. Иначе изучите заголовочные файлы и библиотеки, устанавливаемые в системные каталоги `/usr/include` и `/usr/lib64`, и при необходимости добавьте к проекту их подкаталоги.
- Для запроса имён обрабатываемых файлов у пользователя используйте метод `QFileDialog::getOpenFileName`, имён каталогов — `QFileDialog::getExistingDirectory`.
- Имена всех файлов в каталоге можно узнать с помощью функции `QDir::entryList`.
- Для доступа к файлам используйте класс `QFile` вместо потоков стандартной библиотеки. Сторонние библиотеки часто позволяют открывать файла не только по имени, но и по его дескриптору уровню ОС, который может быть получен через `QFileDevice::handle`, это более «чмстый» способ, чем решать проблемы кодировок при передаче библиотекам имён файлов. Убедитесь, что библиотека не будет закрывать этот дескриптор, поскольку объект `QFile` им владеет.
- Для обмена другим текстом с библиотеками, используйте соответствующие преобразования у класса `QString`, желательно в формы Unicode, например, `fromUtf8`/`toUtf8`. Для запроса имён ещё, скорее всего, не существующих выходных файлов используйте метод `QFileDialog::getSaveFileName`.
- В задачах обработки файлов вам необходимо выбрать размер блока, например, 1 мегабайт, и обрабатывать файл по частям, поскольку легко представить файл, который не поместится в ОЗУ целиком. При этом вам потребуется обработать ситуацию, в которой размер входного файла не кратен размеру блока. При использовании некоторых алгоритмов обработки данных, например, криптографических, размер блока диктуется их требованиями.
- Для выполнения периодических действий по таймеру используйте класс `QTimer`.
- Если вы работаете с библиотекой на языке C, оберните в RAII-классы все парные функции выделения/освобождения ресурсов, предоставляемых библиотекой. Желательно всю связанную с этими классами функциональность также предоставить через члены этих классов, в крайнем случае можно сделать лишь операция неявного приведения типов этого класса к типу ресурса, которым он владеет, для прямого использования в функциях библиотеки, не включающих передачу владения.

14.1.2. Темы проектов

gmp

Сайт: `http://gmplib.org`

Задание: реализовать калькулятор, выполняющий четыре основные арифметические операции над целыми числами не ограниченной каким-либо типом ширины.

Материал:

- Функции основного интерфейса на языке C: `https://gmplib.org/manual/Integer-Functions.html`
- Общие положения о C++-интерфейсе библиотеки: `https://gmplib.org/manual/C_002b_002b-Interface-General.html`
- Дополнительная информация о классе обработки целых чисел: `https://gmplib.org/manual/C_002b_002b-Interface-Integers.html`

Интерфейс на языке C++ в текущей версии виртуальной машины из под используемого в Qt Creator набора инструментальных средств не доступен.

libzip

Сайт: `http://www.nih.at/libzip/index.html`

Задание: вывести дерево файлов, содержащихся в указанном пользователем архиве.

Материал:

Для поддержки не-ASCII путей используйте функцию `zip_fdopen`. Далее потребуется узнать число файлов в архиве через `zip_get_num_entries`, после чего информацию о каждом отдельном файле по его номеру можно получить с помощью `zip_stat_index`. Закрывается архив функцией `zip_discard`.

exiv2

Сайт: `http://www.exiv2.org`

Задание: вывести метаданные выбранного пользователем файла изображения в виде таблицы.

Материал: пример считывания данных с выводом в консоль: `http://www.exiv2.org/doc/exifprint_8cpp-example.html`. Для поддержки не-ASCII путей используйте вместе приведённого в этом примере вызова `Exiv2::ImageFactory::open` его перегрузку, принимающую адрес блока данных в памяти и его размер, а сами данные считайте с помощью `QFile`.

python 3 (>python-3)

Сайт: `http://python.org`

Задание: выполнить скрипт, введённый в интерфейсе программы на языке python и отобразить в графическом интерфейсе выводимый им текст.

Материал: документация API C: `https://docs.python.org/3/c-api/index.html`. Для инициализации/завершения работы интерпретатора используются функции `Py_Initialize` / `Py_Finalize`, между вызовами которых можно исполнять исходный текст на python вызовом `PyRun_SimpleString`, например, такой:

```
1 print("Hello, world!")
```

Стандартный поток вывода в стандартной библиотеке python представлен объектом `sys.stdout`, который можно заменить своим, реализующим требуемый интерфейс — в нашей задаче достаточно реализации метода `write`. Python — язык с динамической системой типов, так что смена типа значения переменной проблемы не представляет, интерфейс также никакой формальной спецификации не требует, достаточно реализовать функцию с нужным именем. Наиболее простой способ — реализовать модуль с требуемоей функцией на C++, а потом выполнить фрагмент на python, привязывающий её к нужному объекту.

Для реализации модуля на C++ и включения его в список встроенных, необходимо передать его имя и функцию его создания функции `PyImport_Appendinittab` перед вызовом `Py_Initialize`. Функция создания должна создать модуль функцией `PyModule_Create` и вернуть его. Этой функции потребуется описание модуля, задаваемое структурой `PyModuleDef`,

которая содержит список функций модуля в виде структур `PyMethodDef`. Эта структура позволит, наконец, задать указатель на функцию, которая будет вызвана из Python в качестве реализации. При использовании флага `METH_VARARGS`, функция реализации должна иметь тип `PyObject* (PyObject*,PyObject*)`. Первый параметр указывает на объект или модуль python, на котором вызвана функция и нам не интересен. Второй параметр является списком аргументов функции. Для преобразования его в типы языка C/C++ может использоваться функция `PyArg_ParseTuple`, в нашем случае мы ожидаем один аргумент строкового типа, так что для простоты достаточно использовать спецификатор `"s"` в качестве строки формата. Если эта функция неудачно разобрала список аргументов, она возвращает ложь. Исключения уровня языка Python уже ей инициировано, осталось вернуть нулевой указатель. Иначе преобразование успешно и возвращается истина. В таком случае требуется вернуть осмысленный объект в качестве результата функции — поскольку в нашем случае возвращать нечего, можно вернуть специальное значение `None` с помощью макроса-оператора `Py_RETURN_NONE;`.

Предположим, что созданный модуль называется `redirection`, и он содержит функцию `write`, тогда замену можно выполнить следующим фрагментом:

```
1  import io,sys,redirection
2
3  class StdoutRedirector(io.IOBase):
4      write = redirection.write
5
6  sys.stdout = StdoutRedirector()
```

poppler

Сайт: `http://poppler.freedesktop.org`

Задание: разрешите открывать пользователю pdf-файлы в отдельных окнах и просматривать их постранично. Разместите в окнах документов кнопку, которая позволяет быстро открыть новое окно с тем же документом. Воспользуйтесь средствами автоматического управления общим владениям, чтобы каждый документ открывался (с точки зрения используемых им ресурсов) только один раз, независимо от числа окон, в которых он отображается.

Материал: документация по интерфейсу для использования с Qt 5: `http://people.freedesktop.org/~aacid/docs/qt5/`

QtNetwork: доступ к http (подключение собственными средствами qmake)

Сайт: `http://qt-project.org/doc/qt-5/qtnetwork-programming.html`, раздел «High Level Network Operations for HTTP and FTP».

Задание: получить информацию в формате JSON по HTTP и отобразить её в графическом интерфейсе.

Материал: найдите сайт, выдающий информацию в формате JSON (например, прогноз погоды или другие сводки, новости и др.) Для чтения ответов на http-запросы необходимо использовать класс `QNetworkAccessManager`, пример использования дан в его документации. Ответ в формате JSON можно разобрать с помощью метода `QJsonDocument::fromBinaryData`.

taglib

Сайт: `http://taglib.org`

Задание: отобразить в виде таблицы теги и их значения для указанного пользователем музыкального файла.

Материал: документация с примерами: `http://taglib.org/api/`. Чтобы избежать перекодирования имён файлов, предоставьте содержимое файла библиотеке через класс `TagLib::ByteVectorStream`.

graphviz

Сайт: `http://www.graphviz.org`

Задание: по заданному пользователем графу построить его планарное представление.

Материал:

- Необходимо использовать graphviz в качестве библиотеки без вызова внешних программ. Исходные данные можно запрашивать в любом виде (матрица инцидентности, смежности, ...).

- Документация по библиотеке описания графов: `http://www.graphviz.org/pdf/Agraph.pdf`, главы 1–5. К сожалению, все строки библиотека принимают в виде параметров типа `char*`, т.е. даже для обычных строковых литералов требуется `const_cast`, хотя библиотека их менять и не собирается.
- Документация по алгоритмам раскладки графа: `http://www.graphviz.org/doc/libguide/libguide.pdf`.
- После получения от библиотеки координат вершин и дуг, их можно нарисовать на экран средствами `QPainter`.

libgcrypt

Сайт: `http://www.gnu.org/software/libgcrypt/`

Задание: вычислите значения криптографической хеш-функции ГОСТ 34.11-2012 для всех файлов в указанном пользователем каталоге и отобразите в виде таблицы.

Материалы:

- Инициализация библиотеки: `https://www.gnupg.org/documentation/manuals/gcrypt/Initializing-the-library.html`
- Интерфейс хеш-функций: `https://www.gnupg.org/documentation/manuals/gcrypt/Working-with-hash-algorithms.html`
- 512-битная версия ГОСТ 34.11-2012 имеет в этой библиотеке идентификатор `GCRY_MD_STRIBOG512`.

mpfr

Сайт: `http://www.mpfr.org`

Задание: вычислите значения функций синуса и косинуса для данного значения угла в градусах с произвольной точностью в битах, указанной пользователем. Выведите результаты и графическое представление в виде диаграммы с единичной окружностью, лучом, и высотами, опущенными на оси координат.

Материал: вычисление можно провести с помощью разложения в ряд Маклорена, а входящие в него арифметические операции выполнить с помощью библиотеки. Документация: `http://www.mpfr.org/mpfr-current/mpfr.html#MPFR-Interface`.

libstatgrab

Сайт: `http://www.i-scream.org/libstatgrab/`

Задание: выведите и обновляйте в реальном времени индикаторы загруженности процессора, использования памяти, диска и сети.

Материалы: инициализация и завершение работы с библиотекой осуществляется функциями `sg_init` и `sg_shutdown` соответственно. Необходимую статистку можно получить вызовами `sg_get_cpu_percents`, `sg_get_mem_stats`, `sg_get_disk_io_stats_diff` и `sg_get_network_io_stats_diff`. Документацию по функциям библиотеки libstatgrab можно получить командой `man имя-функции`.

boost.math (boost)

Сайт: `http://www.boost.org/doc/libs/1_62_0/libs/math/doc/html/index.html`

Задание: выведите график Дзета-функции Римана. Реализуйте возможность перемещаться по плоскости для осмотра разных частей графика перетаскиванием мышью, а также управление масштабом с помощью колеса мыши. Кроме этого реализуйте отображение и ручной ввод координат области координатной плоскости, отображаемой на экране.

Материалы: функция вычисляется с помощью указанной библиотеки, документация по требуемой функции: `http://www.boost.org/doc/libs/1_62_0/libs/math/doc/html/math_toolkit/zetas/zeta.html`. Вместо ручного вычисления всех преобразований систем координат можно воспользоваться встроенными в `QPainter`, см. `http://doc.qt.io/qt-5/qpainter.html#coordinate-transformations`

lzo

Сайт: `http://www.oberhumer.com/opensource/lzo/`

Задание: распакуйте или запакуйте указанный пользователем файл одним из алгоритмов lzo, с демонстрацией коэффициента сжатия в реальном времени.

Материал: в библиотеке имеется множество алгоритмов с одинаковым интерфейсом, различающихся по производительности и сжатию. Возьмите алгоритм lzo1z_999, его описания даны в файле `lzo/lzo1z.h`: `lzo1z_999_compress` и `lzo1z_decompress`. Объём временных данных для работы алгоритма сжатия задаётся макросом `LZO1Z_999_MEM_COMPRESS`, для распаковки это 0. Полный пример можно найти в архиве дистрибутива под именем `examples/simple.c`. Этот пример показывает сжатие блока памяти, для сжатия файла произвольной длины потребуется разбить его на блоки и сжать каждый отдельно. Выбор размера блока следует сделать опцией сжатия, предложив степени двойки в районе от 16 КБ до 16 МБ. Формат файла, содержащий информацию о размере блока и их последовательность разработать самостоятельно. Для отображения текущей степени сжатия можно использовать виджет `QProgressBar`.

libsndfile

Сайт: `http://www.mega-nerd.com/libsndfile/`

Задание: выведите амплитудный график звуковых данных из указанного пользователем файла формата WAV.

Материал: библиотека позволяет получить доступ к звуковым файлам простых форматов в распакованном виде. В таком виде данные представлены в формате Pulse Code Modulation: с некоторой частотой значение входного сигнала на каждом канале преобразовывается в число в некотором диапазоне значений и сохраняется. Например, PCM-формат «44100 герц, 16-бит, стерео» означает, что 44100 раз в секунду значение сигнала на двух каналах замеряется и кодируется числами от 0 до $2^{16} - 1$. Одно такое число библиотека называет элементом (item), а набор элементов для всех каналов в одну единицу времени — фреймом (frame). Для корректной обработки не-ASCII имён файлов используйте функцию `sf_open_fd`. В структуру `SF_INFO`, переданную ей, будет занесено в том числе число фреймов в файле и число каналов. Для чтения фреймов используйте функцию `sf_readf_double`, она преобразует данные к диапазону $[-1; 1]$, независимо от формата файла. Достаточно учитывать данные только одного канала (или их среднее арифметическое, по желанию), если их в файле несколько. Усредните данные фреймов на ширину окна программы в пикселях и выведите. Усреднение следует пересчитывать при изменения ширины окна вашей программы.

fftw

Сайт: `http://www.fftw.org`

Задание: выведите результат применения дискретного преобразования Хартли к выбранному пользователем изображению.

Материал: для загрузки изображения используйте класс `QImage`. Для упрощения дальнейшей обработки преобразуйте его в формат `QImage::Format_RGB8888` — по 8 бит на цветовую составляющую (красную/зелёную/синюю) и ещё один байт заполнения — с помощью функции `QImage::convertToFormat`. Теперь доступ к данным отдельных строк пикселей можно получить с помощью функции `QImage::scanLine`, приведите возвращаемое ей значение к `const QRgb*`. Бибилотека fftw обрабатывает данные в формате `double`, приведите значения типа `QRgb` для пикселей изображения к одной компоненте яркости функцией `qGray` и поделите на `255.`, чтобы получить значения в диапазоне $[0; 1]$, их сохраните в двумерный массив с шаговым доступом.

Для использования библиотеки сначала необходимо «запланировать» требуемое преобразование. В нашем случае понадобится функция `fftw_plan fftw_plan_r2r_2d` (`http://www.fftw.org/fftw3_doc/Real_002dto_002dReal-Transforms.html`). Поскольку необходимы преобразования в `double` из `QRgb`, этот же массив можно использовать и для выходных данных, т.е. совершить преобразование на месте. Созданный план можно выполнить функцией `fftw_execute` (`http://www.fftw.org/fftw3_doc/Using-Plans.html`). Просканируйте полученные данные и найдите их диапазон значений. Приведите эти данные назад к интервалу $[0; 255]$, и с помощью функции `qRgb` сделайте из них значения типа `qRgb`, передав одно и то же значение во все 3 параметра, вы получите соответствующие оттенки серого. Эти значения запишите назад в другое изображение, созданное с тем же размером, что и исходное. Отобразите оба изображения в виджетах `QLabel`, задав их через свойство `pixmap`.

qrencode

Сайт: `http://fukuchi.org/works/qrencode/`

Задание: отобразите на экране QR-код для введённой пользователем текстовой строки.

Материал: кодирование осуществляется функцией `QRcode_encodestring`. Вызывайте её с параметром `hint`, равным `QR_MODE_8` для прямого кодирования 8-битных значений.

Результатом вызова функции является указатель на структуру типа `QRcode`. Она содержит указатель на квадратный массив символов указанной в нём ширины, в котором младший бит каждого значения определяет цвет соответствующего элемента кода. Для вывода их на экран создайте объект `QImage` нужного размера, заполните его требуемыми данными, и выведите на экран в виджете `QLabel`, задав изображение через свойство `pixmap`.

hunspell

Сайт: `http://hunspell.github.io`

Задание: проверьте орфографию введённого пользователем текста и предложите варианты исправления для ошибок.

Материал: документация содержится в самих заголовочных файлах `hunspell.h`/`hunspell.hxx` или может быть выведена командой `man 3 hunspell`. Вы можете использовать интерфейс как на C, так и на C++. При инициализации библиотеки необходимо указать пути к файлам аффиксов и словарю — `/usr/share/hunspell` для текущей виртуальной машины.

Функция `Hunspell_spell` или метод `spell` позволяют проверить отдельное слово на корректность, а `Hunspell_suggest` или метод `suggest` позволяют получить список исправлений для слова с ошибкой, который после использования надо освободить функцией `Hunspell_free_list` или методом `free_list`.

При разбиении текста на слова можно воспользоваться функцией `QChar::isLetter`, которая работает для любых языков, поддерживаемых Unicode.

libnova

Сайт: `http://libnova.sourceforge.net`

Задание: по указанным пользователем времени, дате и географическому положению, отобразите положение Луны и планет солнечной системе на небе, видимом наблюдателю.

Материал: вначале потребуется перевести время к формату используемому библиотекой. Для ввода информации пользователем можно использовать виджет `QDateTimeEdit`, результат ввода имеет тип `QDateTime` и доступен в свойстве `dateTime`. Его можно преобразовать к типу `time_t` вызовом `QDateTime::toTime_t`, а это значение, в свою очередь, к типу `double`, используемому библиотекой для хранения Юлианского дня, функцией `ln_get_julian_from_timet` (`http://libnova.sourceforge.net/group__calendar.html#gb382e9eaf1e6a10d346a31d598a`

Для каждого небесного тела библиотека имеет свой набор функций: `http://libnova.sourceforge.net/modules.html`. Для каждого из них есть функций вычисления экваториальных координат по заданному Юлианскому дню, например, для Луны — `ln_get_linar_equ_coords` (`http://libnova.sourceforge.net/group__lunar.html#gc9ddb9e6b5be5486f25b1b8e4b1bc8`
Полученное значение типа `ln_equ_posn` можно преобразовать к горизонтальным координатам типа `ln_hrz_psn`, исходя из положения наблюдателя типа `ln_lnlat_posn` и времени функцией `ln_get_hrz_from_equ`. Полученный азимут и склонение можно привести к координатам точки на плоскости: отрицательные склонения находятся за горизонтом и не должны выводится. Положительные находятся в интервале $[0; 90]$ и должны быть отображены на всю высоту картинки, а азимуты лежат в $[0; 360)$ и должны быть отображены на всю ширину картинки. В вычисленных местах нарисуйте небольшие окружности соответствующих цветов или более сложные изображения.

Поскольку необходимые функции вычисления координат имеют одинаковый тип для всех небесных тел, можно поместить их в массив указателей на функции для упрощения.

hwloc

Сайт: `http://www.open-mpi.de/projects/hwloc/`

Задание: выведите иерархию процессоров и кеш-памяти вашего (возможно, виртуального) компьютера.

Материал: топология локальной системы создаётся и уничтожается вызовами `hwloc_topology_init` / `hwloc_topology_destroy`. Перечислить объекты топологии можно функциями из раздела «Object levels, depths and types»: `http://www.open-mpi.de/projects/hwloc/doc/`

v1.11.4/a00080.php. Функции определения типа объектов и их атрибутов расположены в разделе «Object types»: http://www.open-mpi.de/projects/hwloc/doc/v1.11.4/a00076.php. В данной задаче интересны объекты типа HWLOC_OBJ_PACKAGE (процессор), HWLOC_OBJ_CORE (ядро процессора), HWLOC_OBJ_PU (исполнительный элемент ядра процессора) и HWLOC_OBJ_CACHE (кеш-память). Для проверки сравните ваш вывод с выводом программы lstopo.

geoip

Сайт: https://github.com/maxmind/geoip-api-c

Задание: показать на карте местоположение данного хоста, исходя из базы соответствия сетевых адресов их географическому расположению.

Материал: преобразование имени в IP-адрес можно сделать с помощью объекта класса QDnsLokkup. Пример нахождения географических данных IP-адресу имеется в дистрибутиве в файле test/test-geoip-city.c. Не забудьте изменить путь к файлу базы данных на установленный в системе в каталоге /usr/share/GeoIP. Из указанного примера можно получить широту/долготу искомого места. Отображение веб-страницы сайта с картами можно выполнить с помощью виджета QWebEngineView, сформировав по полученным данным адрес, например, для сайта Яндекс-карты.

libelf

Сайт: https://fedorahosted.org/elfutils/

Задание: выведите круговую диаграмму распределения объёма указанного пользователем образа программы по секциям: код, данные и данные только для чтения.

Материал: основная информация содержится в заголовочном файле libelf.h. Получить значение типа Elf*, идентифицирующее обрабатываемый в памяти образ программы, можно функцией elf_memory, а освободить его через elf_end. Информация о секциях доступна по их номерам через функцию elf_getscn. Из полученного значения следует извлечь заголовок функцией elf32_getshdr или elf64_getshdr — только одна из них вернёт ненулевой указатель, в зависимости от архитектуры файла, выходные структуры похожи и могут быть обработаны с применением шаблонов. В полученной структуре поле sh_size задаёт размер секции, а поле sh_flags — битовую маску флагов. Нас интересуют только секции, загружаемые в память (SHF_ALLOC). Для секций кода установлен флаг SHF_EXECINSTR, для записываемых данных — SHF_WRITE.

imagemagick

Сайт: https://legacy.imagemagick.org/script/magick++.php

Задание: загрузить выбранное пользователем изображение и применить к нему различные фильтры.

Материал: все требуемые преобразования документированы как методы класса Image: http://legacy.imagemagick.org/Magick++/Image.html, например, blur, colorize, edge и так далее, предоставьте интерфейс работы минимум с тремя из них на ваш выбор. Загрузить данные в такой объект без передачи имён файлов можно через использование BLOB. Для извлечения обработанного изображения из объекта для отображения без использования промежуточных файлов, может использоваться последняя перегрузка метода write — полученные данные могут быть использованы для конструирования объекта QImage по указателю на данные, размерам изображения и его форматом (рекомендуется формировать QImage::Format_ARGB32).

aalib

Сайт: http://aa-project.sourceforge.net/aalib/

Задание: указанное пользователем изображение превратить в ASCII-art, т.е. псевдографическое изображение.

Материал: документация по библиотеке доступна на сайте. Инициализацию библиотеке потребуется провести, как указано в разделе «Initialization as an ascii art renderer», но с явным указанием размера иллюстрации, которая будет создаваться, через заполнение структуры aa_hardware_params. Указатель на данные, которые нужно заполнить оттенками серого изображения, может быть получен через aa_image. После его заполнения, вызов aa_render

выводит изображение на «экран», для нашего драйвера, работающего над памятью, этот результат может быть считан через вызовы функций `aa_text` и `aa_attr`, возвращающие адреса буферов символов и цвета соответственно. Результат может быть выведен в виде раскрашенного текста в графическом интерфейсе в виджет `QTextEdit`, настроенный на использование моноширинного шрифта.

uchardet

Сайт: `https://wiki.gnome.org/Projects/uchardet`

Задание: открыть и показать содержимое текстового файла в неизвестной кодировке.

Материал: все требуемые функции описаны и документированы в заголовочном файле `uchardet.h`. Полученное от библиотеки имя кодировки может быть указано семейству функций перекодирования `iconv_open` / `iconv` / `iconv_close`, документация по ним доступна по команде `man 3 имя-функции`, дополнительных библиотек для их использования подключать не требуется.

libunrar

Задание: вывести дерево файлов в RAR-архиве.

Материал: интерфейс библиотеки содержится в заголовке `dll.hpp`, документирован он сторонним проектом на странице `https://github.com/LiquidFM/libunrar`. Откройте архив функцией `RAROpenArchiveEx` с указанием имени через широкие символы, закрывается архив через `RARCloseArchive`. Между этими двумя вызовами чередуйте последовательность `RARReadHeaderEx` для чтения заголовока очередного файла и `RARProcessFile` для перехода к следующему. Чтобы последний вызов только пропускал данные, не забудьте при открытии архива указать флаг `RAR_OM_LIST`. Для поддержки многотомных и/или зашифрованных архивов можно установить функцию обратного вызова через `RARSetCallback`.

gpsd

Сайт: `http://catb.org/gpsd/`

Задание: вывести схему расположения спутников GPS в реальном времени по данным приёмника.

Материал: для получения данных с приёмника, поскольку прямой доступ к ниму в виртуальной машине маловероятен, следует использовать приёмники смартфонов с ПО, обеспечивающим gpsd-совместимый протокол, например, ShareGPS для ОС Android. Вам потребуется обеспечить доступ ВМ и смартфона в одну локальную сеть. Документация по библиотеке доступна по команде `man 3 libgps`. Вся требуемая информация содержится в структуре `gps_data_t`, из которой можно извлечь азимуты и склонения видимых спутников и их номера. Отобразить эту информацию графически можно в виде точек внутри круга, где угол соответствует азимуту, а расстояние от цента — склонению.

ffmpeg

Сайт: `https://www.ffmpeg.org/`

Задание: Вывести дерево устройств и имеющихся в них источников звука, присутствующих в системе и доступных для использования с библиотекой ffmpeg.

Материал: основная документация находится в заголовочном файле библиотеки `libavdevice`, входящей в ffmpeg: `libavdevice/avdevice.h`. Вначале необходимо зарегистрировать все имеющиеся устройства вызовом `avdevice_register_all`. Перечисление устройств осуществляется функцией `av_input_audio_device_next`. Обратите внимание, что не все устройства имеют полные `long_name` имена. Получить имена источников можно вызовом `avdevice_list_input_sources`.

libssh2

Сайт: `https://www.libssh2.org/`

Задание: отобразить список файлов каталога на удалённой машине по протоколу SFTP и предоставить возможность перемещения по каталогам.

Материал: полный пример с загрузкой файла дан в примере `https://www.libssh2.org/examples/sftp_RW_nonblock.html`. С ним потребуется сделать следующее:

- Заменить вызовы `socket`, `connect` и `close` на использование класса `QTCPSocket`, из которого используется дескриптор сокета через акцессор `socketDescriptor`.

- Заменить вызов `select` на запоминание операции, которая вернула `LIBSSH2_ERROR_EAGAIN` и возврат в цикл обработки сообщений. Отслеживая сигналы `readyRead` и `bytesWritten` на используемом `QTCPSocket` можно узнать, когда соответствующую операцию можно повторить.
- Заменить чтение файла на чтение списка файлов, как в примере `https://www.libssh2.org/examples/sftpdir_nonblock.html`.

libgit2

Сайт: `https://libgit2.github.com/`

Задание: вывести в виде таблицы названия последних коммитов указанного локального репозитория системы контроля версия `git`.

Материал: открытие репозитория осуществляется функцией `git_repository_open`, а закрытие — `git_repository_free`. Ссылку на текущую вершину возвращает функция `git_repository_head` (ссылки освобождаются через `git_reference_free`). Функция `git_reference_peel` позволит развернуть ссылку до коммита, на который она указывает. Функции `git_commit_message_raw` и `git_commit_message_encoding` позволяют получить текст комментария к коммиту и его кодировку, названием считается первая строка этого текста. Перейти к родительским коммитам по их номерам можно функцией `git_commit_nth_gen_ancestor`.

libtorrent (rb_libtorrent)

Сайт: `http://www.libtorrent.org`

Задание: вывести дерево файлов, соответствующих торрент-файлу.

Материал: декодирование файла осуществляется функцией `bdecode`, принимающей интервал указателей на байты, и заносящей результат в объект класса `bdecode_node`, см. `http://www.libtorrent.org/reference-Bdecoding.html#bdecode_node`. Таким образом вы получите корневой узел дерева данных в торрент-файле. Каждый узел хранит один из типов данных, задаваемых вложенным перечислением `type_t`, в зависимости от типа узла на нём можно вызывать разные функции. Выведите полное содержимое torrent-файла полным обходом дерева, чтобы разобраться в его структуре. Не выводите содержимое узла `pieces`, это не строка, а двоичные данные хеш-функции. Разобравшись в структуре, извлеките и выведите дерево файлов. Обратите внимание на способ хранения имён файлов с каталогами (список компонент) и на то, что структура торрент-файла отличается для торрентов из одного файла и нескольких.

libmagic (file)

Сайт: отсутствует.

Задание: вывести типы файлов в заданном каталоге, исходя из их содержимого.

Материал: документация доступна по команде `man libmagic`. Работа с библиотекой сигнатур начинается и заканчивается вызовами `magic_open` / `magic_close`. Для этой библиотеки важно самостоятельно открыть файл, поэтому передайте имя каждого из них функции `magic_file`.

flint

Сайт: `http:/flintlib.org`

Задание: разложите на множители многочлен над кольцом вычетов по заданному модулю и выведите результат в форме произведения, например $x^5 + 3x^4 + -x^2 + 2x + 4 = (x^2 + 3x + 4)(x^3 + x + 1) \pmod 5$.

Материал: вам потребуются функции семейства `fmpz_mod_poly` (раздел 35) и `fmpz_mod_poly_factor` (раздел 36), документация: `http://flintlib.org/flint-2.5.pdf`. Для вывода результата вам потребуется виджет отображения форматированного текста — `QTextBrowser`. Для добавления текста используйте метод `QTextEdit::appendHtml`, заключая показатели степени в теги `<sup>`.

gssdp

Сайт: `https://wiki.gnome.org/Projects/GUPnP`

Задание: вывести обновляемый в реальном времени список устройств локальной сети, обнаружимых по протоколу SSDP.

Материал: функции обнаружения объектов описаны в группе `GSSDPResourceBrowser`: `https://developer.gnome.org/gssdp/stable/GSSDPResourceBrowser.html`. Пример их использования и тестовая программа, которой можно заменить наличие реальных SSDP-устройств в локальной сети, доступны здесь: `https://github.com/GNOME/gssdp/tree/master/examples`.

Ббилиотека GObject предоставляет сервисы, аналогичные таковым для `QObject` в `Qt`: сигналы и свойства. При интеграции с Qt создавать цикл обработки сообщений glib не требуется, т.к. он интегрирован с соответствующим циклом Qt. При использовании `g_object_connect` можно использовать последний параметр `data` для передачи произвольных данных в статическую функцию, по которым может быть восстановлен требуемый контекст. Все подобные объекты содержат встроенный счётчик владельцев, снижаемый по `g_object_unref`.

lz4

Сайт: `http://lz4.github.io/lz4/`

Задание: Распакуйте или запакуйте указанный пользователем файл с помошью библиотеки lz4.

Материал: изучите документацию в заголовочном файле `lz4frame.h` в разделе Advanced Compression Functions. Они осуществляют сжатие/распаковку отдельных блоков, на которые вам понадобится разбить входной файл. Стандартный формат разбиения документирован и может использоваться для тестирования вашей программы с официальном архиватором `lz4`.

libudev (systemd)

Сайт: `https://www.freedesktop.org/software/systemd/man/libudev.html`

Задание: выведите дерево устройств системы с помощью библиотеки libudev.

Материал: для этого потребуется создать контекст библиотеки вызовом `udev_new` и контекст поиска вызовом `udev_enumerate_new`. Фильтры можно не задавать, требуется полное дерево. После вызова `udev_enumerate_scan_devices` можно получить первое устройство вызовом `udev_enumerate_get_list_entry`, а следующие — `udev_list_entry_get_next`. Имя устройства можно получить из записи вызовом `udev_list_entry_get_name`. Имена устройств соответствуют путям в файловой системе sysfs и состоят из компонент, разделённых прямыми слешами. С учётом этого иерархию можно отобразить виджетом `QTreeWidget`.

libisofs

Сайт: `http://libburnia-project.org/wiki/Libisofs`

Задание: вывести дерево файловой системы образа диска в формате ISO-9660.

Материал: краткое введение в библиотеку дано в файле `doc/Tutorial` дистрибутива. Всё, что связано с созданием образов, нам не интересно, но обратите внимание на пример расширения имеющегося образа. После получения объекта `IsoImage` для имеющегося образа, узел корневого каталога может быть получен через `iso_image_get_root`. Дальнейший обход осуществляется через `iso_dir_get_children` / `iso_dir_get_children_count`. В процессе обхода имена узлов доступны через `iso_node_get_name`. Для отображения дерева используйте виджет `QTreeView`, создав требуемую модель на основе функций, описанных выше.

djvu

Сайт: `http://djvu.sourceforge.net/`

Задание: предоставьте пользователю возможность постраничного просмотра указанного им djvu-файла.

Материал: документация содержится в заголовочном файле `libdjvu/ddjvuapi.h`. Для работы с библиотекой потребуется создать контекст функцией `ddjvu_create_context`. Дальнейшая работа с библиотекой осуществляется путём реакции на сообщения, посылаемые ей в очередь в ответ на ваши запросы. Для работы с этой очередью в приложении с использованием Qt следует установить функцию возврата вызовом `ddjvu_message_set_callback`, и из соответствующей функции вызвать один из слотов отложенным образом через `QMetaObject::invokeMethod`. В этом слоте следует обработать все имеющиеся сообщения вызовами `ddjvu_message_peek` и `ddjvu_message_pop`. Также следует заранее создать формат генерации изображения страницы функцией `ddjvu_format_create` вида `DDJVU_FORMAT_RGBMASK32` с масками, соответствующими формату `QImage::Format_RGB32`, и установить требуемый порядок строк вызовом `ddjvu_format_set_row_order`. Документ djvu для простоты можно открыть вызовом

`ddjvu_document_create_by_filename_utf8` по указанному имени файла. После получения сообщения `DDJVU_DOCINFO` можно будет узнать число страниц документа через `ddjvu_document_get_pagenum`. Запрос декодирования отдельных страниц выполняется через `ddjvu_page_create_by_pageno` — если после вызова этой функции `ddjvu_page_decoding_done` истинно, страница взята из кеша ранее декодированных и уже готова к отображению, иначе она будет готова при получении уведомления `DDJVU_REDISPLAY`. Для отображения страницы следует выяснить её размер вызовами `ddjvu_page_get_width` и `ddjvu_page_get_height`, выделить требуемый блок памяти и отрендерить в него страницу вызовом `ddjvu_page_render`. Эти данные могут затем быть использованы для создания из низ объекта `QImage` для отображения на экране.

libcpuid

Сайт: `http://libcpuid.sourceforge.net/index.html`

Задание: Выведите имя центрального процессора, используемого вашей системой и его приблизительную частоту. Выведите список всех технологий SSE/AVX в виде списка, показав цветовой раскраской те из них, которые вашим процессором поддерживаются.

Материал: пример использования дан здесь: `http://libcpuid.sourceforge.net/documentation.html`. Остальная документация содержится в заголовочном файле `libcpuid/libcpuid.h`. Требуемые технологии — константы перечисления `cpu_feature_t`, содержащие последовательности `SSE` или `AVX` в названии. Для вывода списка достаточно применение стандартных средств виджета `QListWidget`.

fluidsynth

Сайт: `http://www.fluidsynth.org/`

Задание: создать графический интерфейс для воспроизведения нот с помощью клавиатуры и мыши. Достаточно одной октавы, любым инструментом.

Материал: документация интерфейса библиотеки дана здесь: `http://fluidsynth.sourceforge.net/api/`, вам потребуется часть до и включая раздел Sending MIDI Events. Для работы синтезатора требуется библиотека инструментов, требуемые файлы в виртуальной машине установлены в виде пакета `fluid-soundfont`. Соответствие номеров нот MIDI приведено здесь: `http://tonalsoft.com/pub/news/pitch-bend.aspx`.

Приложение А

Список операций языка C++

В следующей таблице приведены операции языка C++ в порядке убывания приоритета. Группы операций, находящиеся между горизонтальными строками, имеют одинаковый приоритет. Ассоциативность операций в каждой группе указана стрелками. Объяснение вспомогательных понятий приоритета и ассоциативности операций дано в разделе 4.2.

Операции	Название	Ассоциативность
`::`	Разрешение области видимости	→
`++ --`	Постфиксные инкремент/декремент	→
`type() type{}`	Приведение типов в функциональном стиле	
`()`	Вызов функции	
`[]`	Индексация	
`.`	Выборка	
`->`	Косвенная выборка	
`++ --`	Префиксные инкремент/декремент	←
`+ -`	Унарный плюс/минус	
`! ~`	Логическое и побитовое отрицания	
`(type)`	Приведение типов в стиле языка C	
`*`	Разыменование	
`&`	Взятие адреса	
`sizeof`	Вычисление размера	
`new new[]`	Выделение динамической памяти	
`delete delete[]`	Освобождение динамической памяти	
`.* ->*`	Обращение по указателю на член класса	→
`* / %`	Умножение, деление, взятие остатка	→
`+ -`	Сложение, вычитание	→
`<< >>`	Побитовый сдвиг влево/вправо	→
`< <= > >=`	Отношения	→
`== !=`	(Не)равенство	→
`&`	Побитовое И	→
`^`	Побитовое исключающее ИЛИ	→
`|`	Побитовое ИЛИ	→
`&&`	Логическое И	→
`||`	Логическое ИЛИ	→
`?:`	Условная (тернарная) операция	←
`=`	Простое присваивание	
`*= /= %=`	Составное присваивание	
`+= -=`		
`<<= >>=`		
`&= ^= |=`		
`throw`	Бросание исключения	←
`,`	Запятая	→

Операции `const_cast`, `static_cast`, `dynamic_cast`, `reinterpret_cast`, `typeid`, `sizeof...`,

`alignof` и `noexcept` в таблице не представлены, т.к. их синтаксис содержит скобки, что всегда приводит к однозначной трактовке порядка вычислений с их участием.

Литература

[1] ***N4618***, Working Draft, Standard for Programming Language C++, 2016-11-28. https://github.com/cplusplus/draft/raw/master/papers/n4618.pdf

[2] ISO/IEC/IEEE 60559:2011 — Information technology — Microprocessor Systems — Floating-Point arithmetic http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57469

[3] ***Donald E. Knuth***. Big Omicron and big Omega and big Theta, SIGACT News, Apr.-June 1976, pp. 18-24.

[4] ***Donald E. Knuth***. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973. Second edition, 1998.

[5] ***Henry S. Warren***. Hacker's Delight, 2nd edition. Addison-Wesley Professional, 2012.

[6] ***Кауфман В.Ш.*** Языки программирования. Концепции и принципы. — М. ДМК Пресс, 2010 — 464 с.

[7] ***Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein***. Introduction to Algorithms, Third Edition — The MIT Press, 2009, 1292 p.

[8] ***Нестеренко А.Ю.*** Теоретико-числовые методы в криптографии // Московский государственный институт электроники и математики, 2012, 224 с.

[9] ***Столяров А.В.*** Программирование на языке ассемблера NASM для ОС Unix: Уч. пособие. — 2-е изд. // М. МАКС Пресс, 2011, 188 с.

История версий

В данной главе приведена история изменений между версиями, которые автор предоставлял в общий доступ. Следует понимать, что над книгой ведётся активная работа, так что её структура может меняться достаточно серьёзно. Для отслеживания этих изменений читателями и предназначена эта глава.

Следующие недочёты текущей версии книги автор знает и планирует это исправить:

1. Нет большинства ссылок на более детальное рассмотрение тех или иных тем сразу после окончания их поверхностного разбора («Эта тема будет подробнее рассмотрена в главе [X]»).
2. Нет алфавитного и систематического списков терминов со ссылками.
3. Имеются помарки вёрстки: некоторые слова вылезают на поля или переносятся по слогам в странных местах, листинги программ некрасиво разрываются началом новой страницы и т.д.

- ***06.12.2016***
  - Добавлен раздел «Диспетчеризация по тегам. if constexpr».
  - Исправлены ошибки.
- ***22.11.2016***
  - Добавлены дополнительные итоговые проекты.
  - Убраны не адаптированные задания.
  - Исправлены ошибки.
- ***15.11.2016***
  - Удалена прошлогодняя история изменений.
  - Изменён стиль фрагментов кода.
  - Раздел «Наследование классов» перенесён перед рассмотрением шаблонов и дополнен.
  - Добавлен пример использования внешних библиотек.
  - Добавлены задания на итоговый проект.
  - Исправлены ошибки.