C++: устройство и применение

Лебедев П.А.

13 апреля 2018 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 [8].

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?»

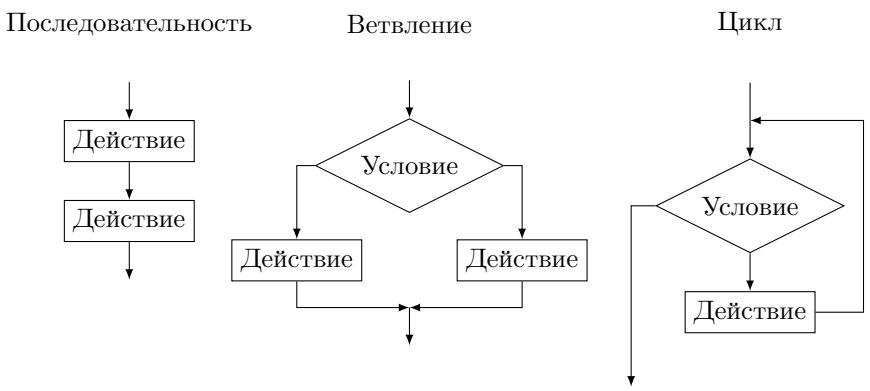


Рис. 1.1: Конструкции структурного программирования

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 (in)*** и ***выходных (out)*** параметров, определяющих данные, передаваемые процедуре при каждом вызове и получаемые от неё по завершению работы соответственно. 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 каче-

стве результатов своей работы. В рамках этой парадигмы строятся основные инструментальные средства программирования — генераторы кода и трансляторы. Средства ***рефлексивной (reflection)*** 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. Типизация

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

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

- По времени, когда осуществляется проверка допустимости совершаемых программой операций к элементам программы в соответствии с их реальными типами: ***статическая (static)*** (во время трансляции) или ***динамическая (dynamic)*** (во время выполнения).
- По возможности использования значений одного типа в контекстах, где требуются значения другого, 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 запросу программиста).
- По наличию или отсутствию ***механизма вывода типов (type inference)*** — автоматическому определению типов элементов программы в противоположность необходимости явного указания типов программистом.

1.5. Инструментальные средства

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

1.5.1. 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, который используется, если нет более точного.

Компилятором (compiler) называют транслятор, понижающий уровень представления программы, т.е. осуществляющий перевод на язык программирования более низкого уровня. 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 более 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.

1.5.2. 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)***.

Особую роль играют инструментальные средства, предназначенные для поиска ошибок в программах сверх тех, которые могут быть обнаружены в процессе их сборки.

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. К средствам ***динамического анализа (dynamic analysis)*** кода относят средства, которые путём инструментации добавляют в него дополнительные проверки на корректность выполняемых программой действий сверх тех, что предусмотрены самим языком. Они чрезвычайно важны для языков, которые сами по себе многих таких проверок не предусматривают, например, языки C и C++, которые мы собираемся рассматривать.

Другим применением средств инструментации является сбор статической информации о частоте выполнения отдельных элементов программы. После сбора этой информации программа может быть перекомпилирована с её учётом, таким образом компилятор получает доступ к информации, полезной для оптимизации, которая обычно доступна только интерпретатору. Такие оптимизации называют ***оптимизациями, основанными на профилировании (profile guided optimization)***.

Также существуют средства ***статического анализа (static analysis)*** кода, т.е. анализа исходного кода без выполнения программы. Такие средства оперируют аналогично компилятору, но, поскольку не генерируют машинный код на выходе, могут потратить своё рабочее время на поиск потенциальных ошибок в программе, на которые компилятору нет времени тратить в обычном режиме и не требуется диагностировать по стандарту языка.

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.

Для отслеживания истории изменений исходных текстов, составляющих процесс разработки ПО, применяют ***системы контроля версий (version control systems, VCS)***. Эти системы незаменимы в случае ведения разработки группой программистов, но также полезны и при индивидуальной работе, т.к. структурируют процесс внесения изменений и позволяют легко отменять проблемные.

Для нахождения таких проблемных изменений по возможности в автоматизированном режиме, написанное ПО должны содержать в себе код для тестирования своей функциональности или использовать дополнительные инструментальные средства, обеспечивающие тестирование. В современной практике все эти инструментальные средства используются совместно, например, каждое изменение перед занесением в основную ветвь системы контроля

версий должно успешно пройти статический анализ и все тесты под управлением средств динамического анализа.

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

Функционирование программ

В этой главе мы рассмотрим основные принципы функционирования программ как с пользовательской точки зрения, так и с точки зрения аппаратуры и системного ПО.

2.1. 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 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 будет рассматриваться платформа x86 в 64-битном защищённом режиме с плоской моделью памяти (далее просто x86-64), лежащего в основе большинства современных ПК. 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: 64 бита для рассматриваемой архитектуры x86-64. Таким образом, на данной архитектуре имеется 2^{64} байт = 16 ЭБ (эксабайт) ***адресного пространства (address space)***. Это лишь теоретический максимум на количество различных адресуемых байтов памяти, реальный физический объём ОЗУ в настоящее время значительно меньше.

Сопоставлением частей этого адресного пространства реальной оперативной памяти занимается операционная система с помощью средств поддержки виртуальной памяти, встроенных в ЦП, а большой объём запаса этого пространства обеспечивает дополнительную гибкость этого процесса. Первично в адресное пространство программы отображаются её машинный код и требуемые для его работы данные из исполняемого файла, полученного в результате работы трансляторов, дополнительную память программа запрашивает у ОС в процессе выполнения по мере надобности. Код программы и её данные хранятся в едином адресном пространстве, что соответствует архитектуре фон Неймана среди архитектур с хранимой программой. Подобное отображение, т.е. виртуальное адресное пространство, своё для каждой выполняющейся программы, которой операционная система выделяет отдельные, не зависящие от других программ ресурсы. Использование механизма виртуальной памяти обеспечивает изоляцию программ друг от друга в многозадачной среде, а также позволяет разделять между программами код библиотек без его дублирования, увеличивать эффективный объём памяти за счёт страничного обмена с диском и другие функции ОС.

Отображение физической памяти в виртуальное адресное пространство осуществляется за счёт создания ОС в ОЗУ таблицы виртуальной памяти, содержащей соответствия между физическими и логическими адресами для всех адресных пространств. При это сопоставление задаётся не по отдельным байтам, а целыми ***страницами (page)***, для x86-64 их стандартный размер — 4 КБ, но для эффективных отображений большого объёма поддерживается и более крупные страницы, обычно 2 МБ. Попытка обратится из-за ошибки программы к виртуальному адресу, которому не соответствует физической памяти, заставляет процессор передать управление ОС, которая завершает процесс аварийно. Если же соответствие отсутствует не из-за ошибки в программе, а из-за того, что ОС записала соответствующие данные на диск, чтобы освободить ОЗУ для других нужд, операционная система считает данные в ОЗУ обратно (вероятно, по другим физическим адресам), восстановит соответствие в таблице виртуальной памяти и даст команда процессоры повторно выполнить инструкцию обращения к памяти прикладной программы так, что с её точки зрения данные никуда не пропадали.

С другой стороны, процессор также является ресурсом, который необходим всем программам. Каждый ***process (процесс)***, соответствующий запущенной программе, включает в себя несколько ***нитей (thread)*** выполнения, соответствующих одновременно выполняемым частям логики программы, которые работы с данными в едином адресном пространстве. ОС разделяет один или несколько процессоров (их ядер, или вычислительных элементов), отслеживая состояние всех нитей процессов в системе в одном из трёх состояний:

- ***Выполняющиеся (running)*** — нити, выполняющиеся в данный момент. Их число не может превышать число реальных вычислительных элементов в системе. Нити проводят в этом состоянии не более кванта времени, который им отводится ОС для работы, после чего управление принудительно возвращается операционной системе по таймеру. Состояние процессора в этот момент сохраняется и нить переходит в следующее состояние.
- ***Готовые (ready)*** — нити, которые желают выполняться, но не выполняются в данный момент, ожидая своей очереди. По истечении кванта времени одна из нитей освобождает процессор и ОС выбирает из списка готовых очередную задачу. Её состояние восстанавливается на процессоре и ей передаётся управление. ОС осуществляет выбор готовых задач по разным критериям, включая порядок в очереди и приоритеты.
- ***Заблокированные (blocked)*** или ожидающие — это нити, которые в данный момент не рассматриваются как готовые, поскольку ожидают наступления некоторого события. В такое состояние нити переводятся ОС по их собственной команде или в ситуации, когда их запросы не могут быть обработаны без задержек. Здесь находятся, например, нити, ожидающие истечения времени по таймеру или ответа от внешних устройств на запрос операций ввода-вывода.

В качестве кванта времени выбирается такой промежуток, что затраты на переключение состояния процессора между задачами приемлемы, но достаточно малые с точки зрения человека, что создаёт для него иллюзию одновременной работы программ даже на однопроцессорной машине.

Чтобы гарантировать, что прикладные программы могут выполнять только те действия, на которые у них имеются привелегии, в том числе не нарушать функционирование самой операционной системы, ЦП содержит средства поддержки ***защищённого режима (protected mode)***, в котором каждый выполняемый код имеет свой уровень привилегий по выполнению команд. Неограниченные права обычно сохраняются только за самой ОС и её частями, например, драйверами устройств, при этом прикладные программы прямого доступа к устройствам

не имеют, и обращаются к ним через запросы к ОС. По этой причине рассматриваемая нами схема выполнения программ и включает в себя только ЦП и 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.
- Рассматриваемый набор инструкций x86-64 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.

Регистровый файл и есть то состояние процессора, о котором вы упомянули, обсуждаю реализацию многозадачности.

Архитектура x86 была создана в 1978 году и изначально являлась 16-битным расширением 8-битного процессора. В то время основных регистров было 10: `AX`, `BX`, `CX`, `DX`, `BP`, `SP`, `SI`, `DI`, `FLAGS` и `IP`. Последний чаще всего трактуется как набор отдельных бит, определяющих состояния и режимы работы процессора, доступные для изменения прикладными программами.

Остальные регистры обычно рассматриваются как единые 16-битные значения. Первые четыре из этих регистров изначально допускали отдельный доступ к их нижним и верхним байтовым половинам, например `AH` и `AL` для регистра `AX` (High и Low).

Регистры процессора делят на регистры ***общего назначения (general purpose)*** и ***специальные (special purpose)*** регистры. Первые могут использоваться программами для любых необходимых им целей. Специальные регистры в свою очередь играют фиксированную роль в функционировании процессора с точки зрения их содержимого, как, например, рассмотренный выше регистр флагов. С другой стороны, регистры можно относить к специализированным, если они играют особую роль в выполнении некоторых команд. Наличие имён у всех регистров (в отличие от их нумерации) x86 подсказывает, что в первых представителях этой архитектуры большинство команд не могло использовать произвольные регистры для своего функционирования, а работало только с конкретными: Acuumulator, Base, Counter, Data, Base Pointer, Stack Pointer, Source Index, Destination Index, так что они во многом были скорее специализированными, нежели общего назначения — подобная классификация часто неоднозначна. В последующем развитии архитектуры эти ограничения были частично сняты, но её проявления нередко видны и в современных представителях.

Через семь лет архитектура была расширена для 32-битной: все вышеперечисленные регистры были расширены вдвое, и получили имена с префиксом `E`: `EAX`, `EBX` и т.д. Нижние 16-битные и 8-битные части остались доступными под старыми именами, отдельных имён для старших 16-битных половин предусмотрено не было. Этот этап развития примечателен также возникновением рассмотренных нами механизмов виртуальной памяти, плоского адресного пространства и защищённого режима работы процессора, поддерживающего разделение на привилегированный код операционной системы и прикладные программы. 32-битные программы до сих пор составляют бóльшую долю имеющегося ПО и способны выполнятся даже в современных полноценных 64-битных операционных системах.

Выпущенный в 1997 году процессор Pentium with MMX Technology положил начало развитию в составе регистров процессора векторных регистров и наборов команд по их использования. Эти инструкции позволяют применять к таким регистрам, хранящим сразу несколько значений, операции, выполняющие несколько одинаковых действий одновременно в рамках подхода к параллельному программированию SIMD (Single Instruction Multiple Data). Развитие этих наборов команд шло параллельно развитию самой архитектуры, увеличивая наборы доступных команд и ширину векторных регистров с 64 бит (MMX) до 512 бит (AVX-512)

на момент написания этого текста.

64-битная версия архитектуры x86, разработанная компанией AMD, получила первую реализацию в 2003 году, и была создана по той же причине, что и 32-битная в своё время — превышение размеров адресного пространства требованиями развивающегося ПО и практическая возможность установки соответствующих объёмов ОЗУ. Регистры вновь были расширены вдвое, в этот раз получив префикс `R`. Кроме этого появилось ещё 8 регистров общего назначения `R8`–`R15` и это оказалось не менее важным нововведением для архитектуры, которая исторически отличалась малым количеством регистров. В настоящее время у архитектуры x86-64 имеется множество дополнительных регистров, но они обычно не играют большой роли в работе прикладных программ, и нами рассматриваться не будут.

Упомянутый выше регистр `IP` (Instruction Pointer), `RIP` для x86-64, хранит адрес инструкции, следующей за исполняемой в настоящий момент. Регистр такого специального назначения также называют `PC` (Program Counter), независимо от конкретной архитектуры. Инструкции архитектуры x86-64 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 и реализации конструкций структурной парадигмы программирования, что обеспечивает полноценность машинного кода x86-64 с точки зрения реализации на нём алгоритмов.

В настоящее время нельзя не упомянуть ещё один компонент, расположенный между основными блоками процессора и оперативной памятью — ***кэш-память (cache memory)***. Из-за различия темпов развития процессоров и памяти, последняя отстала от ЦП по быстродействию — по крайней мере, в тех рамках, в которых её применение в необходимых количествах диктует невозможность использования самой быстрой памяти по экономическим соображениям. Чтобы процессор не простаивал в ожидании запросов к ОЗУ, между ним и ей располагается компромиссная по объёму и производительности кэш-память. В настоящее время на архитектуре x86-64 она физически является частью кристалла самого ЦП и занимает значительную часть его площади. Поскольку размер кэш-память также влияет на время ответа на запросы к ней, в современных ЦП применяется многоуровневая кэш-память, в которой также скорость работы обратно пропорциональна объёму. Например, на современном процессоре рассматриваемой архитектуры может имеется по 32 КБ кэш-памяти декодированных инструкций и столько же кэш-памяти данных первого уровня L1 на каждое ядро процессора, 256 КБ кэш памяти L2 на каждое ядро, и 6 МБ общего кэша L3 на весь многоядерный процессор.

Кэш-память, имея меньший, чем оперативная, объём, работает в предположении, что в отдельные промежутки времени программы работают не со всем объёмом ОЗУ, а лишь с его ограниченной частью, и стараются сохранять в себе только наиболее часто и недавно используемые данные. Это позволяет ускорять повторные чтения данных, т.к. они могут быть удовлетворены из более быстрой кэш-памяти. При записи в ОЗУ можно записать данные в более быструю кэш-память, а в само ОЗУ — позднее, и тоже сэкономить время. Для однопроходных алгоритмов, которые не содержат повторных чтений данных, современная кэш-память также позволяет ускорить работу за счёт анализа паттернов обращения к памяти и осуществлению упреждающих чтений: если обнаруживается, что происходит, например, чтение памяти строго подряд, кеш-память может сама считывать последующие данные из ОЗУ ещё до поступления соответствующих запросов от программы, чтобы к моменту их возникновения данные уже были в ней. Последний пример показывает, что эффективность работы кэш-памяти сильно зависит от успешности предсказания дальнейших действий программы. В настоящее время эффекты кеш-памяти могут иметь первостепенное значение для производительности работы алгоритмов над структурами данных, поэтому последние необходимо рассматривать не только с теоретико-математической точки зрении, но и с точки зрения их соответствия благоприятной работе кэш-памяти. Мы обязательно будем делать это в последующих главах.

Рассмотренная нами схема взаимодействия программы с окружающим её видом аппаратных компонентов приведена на рис. 2.1. Код и данные программы 1 загружены в ОЗУ и отображены в её адресное пространство, образуя её процесс. Относительно содержимого этого адресного пространства и выполняются инструкции чтения и записи памяти в нитях этого процесса. Код и данные процесса 2 с точки зрения процесса 1 не существуют. Код процесса 2 также находится в ОЗУ, а данные были перемещены на диск операционной системой для освобождения ОЗУ в рамках страничного объёма с диском — эти данные с точки зрения процесса 2 никуда не делись, и будут возвращены ОС при обращении к ним.

Остальные регистры процессора и конкретные команды мы рассмотрим по мере надобности.

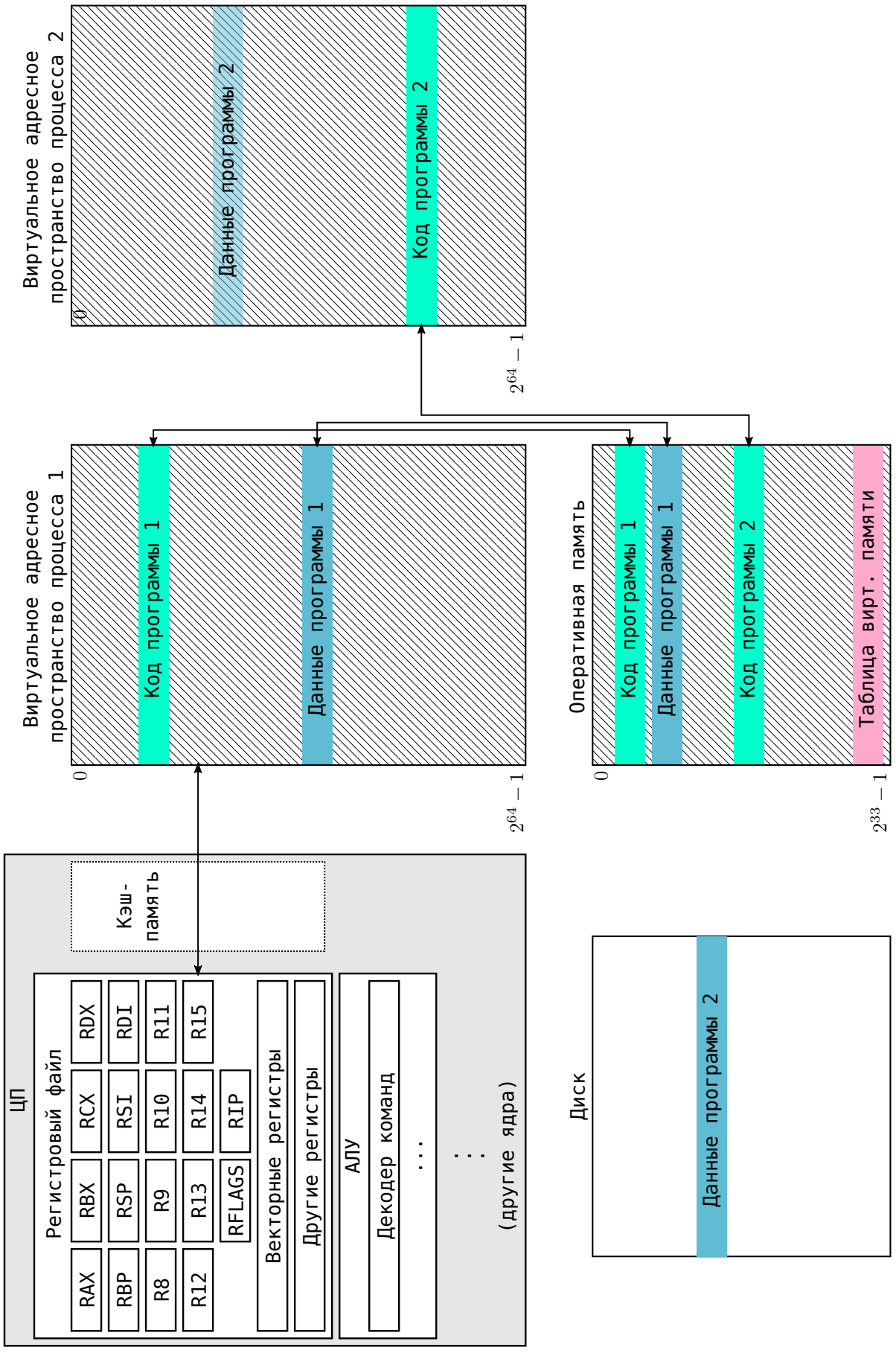


Рис. 2.1: Вид аппаратных компонентов со стороны прикладной программы.

Для желающих подробнее ознакомится с языком ассемблера автор предлагает воспользоваться книгами [11] и [12]. В первой язык ассемблера рассматривается в связке с ОС Linux и языком C, из второй можно почерпнуть вопросы использование ассемблера в ОС Windows, а также современные векторные расширения ЦП.

2.2. Взаимодействие программ с ОС и пользователем

Одной из фундаментальных программ является ***операционная система (ОС) (operating system (OS))*** — программа, обеспечивающая распределение вычислительных ресурсов компьютера между остальными программами. Современные ОС — не одна программа, а сложные комплексы программного обеспечения, детальное рассмотрение которых даётся в отдельном курсе «Операционные системы».

Операционная система обычно загружается автоматически при включении компьютера, и встаёт вопрос о том, как пользователю запустить требующуюся ему прикладную программу. Программа, предназначенная для запуска других программ, также запускается автоматически и называется ***оболочкой (shell)***. В большинстве ОС все данные, которые сохранены на устройствах постоянного хранения, представлены пользователю и прикладным программам в виде иерархии файлов, которые рассматриваются как последовательности байтов. В современных ОС с графическим пользовательским интерфейсом оболочки используют его для представления интерфейса рабочего стола или аналогичного.

Исторически, физическим интерфейсом пользователя являлся ***терминал (terminal)*** — набор, обычно включавший монитор и клавиатуру, подключённый к большой ЭВМ, обслуживающий множество таких терминалов — этот термин возник до широкого использования персональных компьютеров. Информация, выводимая на терминал, обычно являлось чисто текстовой, и таковым был интерфейс большинства программ. В терминалах использовался моноширинный шрифт, так что экран мог отображать прямоугольную таблицу фиксированной ширины и высоты в символах. Дальнейшее развитие дало возможность менять цвета текста и фона отдельных символов и некоторые другие атрибуты.

Оболочки для терминалов являлись интерпретаторами предметно-ориентированных языков, областями специализации которых были запуск и контроль других программ и операции с файловой системой. Несмотря на подавляющее использование графических интерфейсов рядовыми пользователями в наши дни, все ОС до сих пор поддерживают программы с текстовым интерфейсом, которые по прежнему удобнее в задачах автоматизации, администрирования, удалённого доступа к системам и др. Для работы с ними используются программы-***эмуляторы терминала (terminal emulator)***. Их окно используется для отображения текстовой информации и заменяет монитор реального терминала, а при активации их окна вводимые с клавиатуры данные направляются программе, активной на терминале так, словно набраны на клавиатуре от настоящего терминала, неразрывно связанной с этим монитором. В ОС Windows такой программой является ***Командная строка (Command Prompt)***, в ОС Linux эмуляторов терминала множество, например Konsole или XFCE4-Terminal. Программирование текстового интерфейса проще, и потому будет использоваться в большинстве программ, рассматриваемых в этой книге, посвящённой, в первую очередь, конкретным языкам программирования (основы построения современных графических интерфейсов будут затронуты в последних главах).

При открытии эмулятора терминала в нём обычно запускается оболочка с текстовым интерфейсом. Мы будем рассматривать работу с эмулятором терминала в ОС Linux, останавливаясь на принципиальных различиях от ОС Windows, с которой читатель, вероятно, лучше знаком. Рассмотрим работу в одной из наиболее популярных оболочек для данной ОС — bash.

При готовности принять очередную команду пользователя отображается ***приглашение (prompt)*** оболочки, которое может выглядеть так:

```
user@tsugumi ~ $
```

Оно состоит из следующих частей:

1. `user` — имя пользователя, от имени которого выполняются команды, в данном случае буквально «user».
2. `@` — знак, отделяющий имя пользователя от имени компьютера.
3. `tsugumi` — имя компьютера, выполняющего вводимые команды.
4. `~` — положение в файловой системе, относительно которого трактуется вводимая команда.

5. `$` — разделитель, заканчивающий приглашение к вводу (также может использоваться `%` или `#` для привилегированного режима).

Появление приглашения означает, что предыдущая команда выполнена, и можно вводить следующую. При вводе для коррекции строки можно использовать клавиши Backspace/Deleteи стрелки влево-вправо для перемещения по строке. Стрелками вверх-вниз можно вызвать предыдущие выполненные команды. Наконец, клавиша Enter запускает команду на выполнение.

Простые команды оболочки bash состоят из имени команды и её аргументов, разделённых пробелами, например, чтобы узнать время загрузки ОС, можно выполнить следующее:

```
user@tsugumi ~ $ uptime -s↵
2017-01-23 20:49:33
user@tsugumi ~ $
```

> Чтобы включить сам знак пробела в аргумент или имя команды, перед ним следует поставить знак `\`, чтобы «отменить» его специальный смысл. Особый смысл самого знака обратного слеша можно отменить его удвоением. Существуют и другие способы, которые читатель может узнать из документации. Такой способ отмены специальных значений символов в строках встретится нам далее и в самом языке C++.

Аргументы, состоящие из одного дефиса и символа являются короткими опциями, включающими разные режимы работы программ, их часто можно объединять, указывая все буквы слитно после одного дефиса. Встречаются также длинные опции, начинающиеся с двух дефисов и имеющие длинные имена, их нужно давать отдельными аргументами. Такие аргументы также называют ключами. Важен ли порядок опций и остальных аргументов, и придерживается ли синтаксис аргументов этих правил вообще — зависит от команды. Большинство команд при запуске с единственным аргументом `--help` выводят справку об использовании самих себя. Более полная документация может быть получена из системных баз данных справки командами `man имя-программы` или `info имя-программы`. В ОС Windows аргументы ключи начинаются обычно со знака `/`, общесистемной базы справочной информации в этой ОС нет.

Все вводимые команды выполняются относительно текущего места в файловой системе, которое отображается в приглашении. Файловая система ОС Linux является цельной иерархией, начинающейся с ***корневого каталога (root directory)***. (Нескольких корней в виде букв дисков, как в ОС Windows, в ней нет, вместо термина «***папка (folder)***» применяется более традиционное «***каталог (directory)***».) Корневой каталог содержит файлы и другие каталоги, которые, в свою очередь, могут также содержать другие каталоги и файлы, образуя древовидную иерархию, все они имеют свои имена. Часть имени файла после последний точки (если она есть) называют ***расширением (extension)***, которое является подсказкой о формате содержимого данного файла (каталоги редко имеют расширения). Каталоги в данном каталоге называют его ***подкаталогами (subdirectory)***, а он для них является ***надкаталогом (up-directory)***.

В одном каталоге не может быть двух файлов или каталогов с одним и тем же именем, но в файловых системах ОС UNIX регистр имён не только сохраняется, как в ОС Windows, но и учитывается: `A` и `a` — разные имена. В разных каталогах одинаковые имена не конфликтуют. Любой объект файловой системы может быть однозначно поименован ***абсолютным (absolute)*** именем, включающем именя всех каталогов, начиная с корневого. Абсолютное имя самого корневого каталога — `/`. Абсолютное имя файла `file`в корневом каталоге — `/file`. Абсолютное имя файла в глубине иерархии начинается с имени корневого каталога, за которым следуют имена каталогов на пути к требуемому объекту, и имя самого объекта, разделённые символом `/` (который в самиих именах не допускается). Например, файл с именем `file` в каталоге `b`, расположенном в подкаталоге `a` корневого каталога имеет абсолютное имя `/a/b/file`. Абсолютные имена носят такое название потому, что их смысл не зависит от текущего положения в файловой системе — они указывают весь путь от корня иерархии.

Поскольку использование абсолютных имён зачастую громоздко, используется понятие ***текущего каталога (current directory)***, относительно которого можно давать сокращённые имена. Текущий каталог отображается в приглашении оболочки и определяет текущий каталог для запускаемых программ и, следовательно, смысл не абсолютных имён файлов в них. В рассмотренном примере в качестве текущего каталога указано `~` — сокращение оболочки, означающее домашний каталог текущего пользователя, его можно использовать в командах оболочки и оно будет автоматически раскрыто. Его настоящее абсолютное имя в данном случае — `/home/user` — в подкаталоге `home` корневого каталога содержатся домашние каталоги обычных пользователей с их именами.

Имена файлов, не начинающиеся с имени корневого каталога, называют ***относительными (relative)***. При их использовании к ним слева приписывается значение текущего каталога. Например, если текущий каталог — домашний для рассматриваемого пользователя, то указание имени файла ilca/b соответствует файлу с абсолютным именем `/home/user/a/b`. Отдельное имя файла без слешей-разделителей тоже является примером относительного имени и означает файлы в текущем каталоге. Использование относительных имён позволяет переместиться в ближайшее место к файлам, работа с которыми ведётся, и далее использовать короткие имена, не включающие длинный общий путь к ним.

В каждом каталоге всегда есть две специальных записи: `.` (точка) — сам данный каталог и `..` (две точки) — надкаталог (коренвой каталог для корневого). Факт существования нескольких полных имён, соответствующих одному объекту файловой системы объясняется тем, что объекты в файловых системах UNIX идентифицируются на самом деле числами, а записи в каталогах — лишь ***ссылки (link)*** на них. Приведём пример: абсолютное имя `/home/user/../other_user/a/./file` эквивалентно `/home/other_user/a/file`.

Для изменения текущего каталога оболочки используется команда `cd`: при указании одного аргумента-каталога, она делает его текущим, без аргументов она делает текущим домашний каталог текущего пользователя:

```
user@tsugumi ~ $ cd /usr/lib↵
user@tsugumi /usr/lib $ cd ..↵
user@tsugumi /usr $ cd↵
user@tsugumi ~
```

Этот пример показывает использование обозначения надкаталога для выхода на один уровень вверх.

Содержимое каталогов отображает команда `ls`, текущего, или указанного аргументом. С ключом `-l` она отображает не только имена файлов в указанном каталоге, но и их характеристики в виде таблицы (чтобы размеры файлов отображались в привычных обозначениях, а не в блоках, следует добавить ключ `-h`). Ключ `-A` позволяет увидеть «скрытые» объекты — те, имена которых начинаются с точки (в UNIX нет отдельного от имени атрибута скрытости, как в Windows), ключ `-a` покажет кроме этого и записи `.` и `..`.

```
user@tsugumi ~ $ ls↵
Desktop    Downloads  n4618.pdf  Projects  readme.txt  Videos
Documents  Music      Pictures   Public    Templates
user@tsugumi ~ $ ls -lh↵
total 32M
drwxr-xr-x   2 user user   82 окт 18 13:45 Desktop
drwxr-xr-x   2 user user   32 дек 18 10:36 Documents
drwxr-xr-x   2 user user  100 ноя 29 16:15 Downloads
drwxr-xr-x   2 user user   38 дек 20 14:47 Music
-rw-r--r--   1 root root 6.0M авг  8 21:14 n4618.pdf
drwxr-xr-x   2 user user   37 дек 20 21:22 Pictures
drwxr-xr-x 105 user user 4.0K янв 27 14:42 Projects
drwxr-xr-x   2 user user    6 окт 12  2014 Public
-rw-r--r--   1 user user  22K янв 24 03:51 readme.txt
drwxr-xr-x   2 user user    6 окт 12  2014 Templates
drwxr-xr-x   2 user user    6 окт 12  2014 Videos
user@tsugumi ~ $ ls -lh /etc/samba
total 12K
-rw-r--r-- 1 root root 1.1K сен 26 20:50 smb.conf
-rw-r--r-- 1 root root 7.8K янв 20 22:30 smb.conf.default
```

В табличной форме приведены: тип файла (первая буква, `d` — каталог, дефис `-` обычный файл), права доступа (следующие 9 симвлов), число ссылков, имена пользователя и группы, владеющих файлом, размер файла, дата последнего изменения и, наконец, само имя файла. В короткой форме типы объектов можно различать по цвету (в этом примере не показано) или по маркерам, которые можно добавить ключом `-F`.

Этих команд достаточно для ориентации в файловой системе, остальные команды по операциям с файлами и другие детали языка оболочки командной строки можно найти во встроенной документации и соответствующей литературе. Для упрощения многих из них в большинстве систем можно использовать программу `mc` (Midnight Commander) — псевдографический панельный файловый менеджер, включающий простой текстовый редактор.

Среди рассматоренных команда `cd` является ***внутренней (internal)*** — она входит в язык оболочки и выполняется ей самой. Если оболочка встречает команду, которая ей неизвестна, она считается именем ***внешней (external)*** программы, которая ищется в системных каталогах и запускается с указанными аргументами. В обычном режиме оболочка ждёт завершения запущенной программы и отображает новое приглашение к вводу только после завершения работы запущенной программы. Выполнение зависших или попросту слишком долго работающих программ можно прервать нажатием комбинации клавиш Control+C. В результате своей работы внутренние и внешние команды предоставляют запустившей ей программе ***код возврата (exit code)*** — небольшое целое число, означающее успешность выполнения поставленной им задачи. Язык оболочки предполагает использование кода 0 для обозначения успешности, а ненулевых — для различных ошибок. Успешность работы программы может использоваться для ветвлений в программах на языке оболочки. Для отображения кода возврата последней выполненной команды можно использовать команду `echo $?`. Покажем её действие с внешними командами `true` и `false`, которые ничего не делают «успешно» и «неудачно» соответственно:

```
user@tsugumi ~ $ true↵
user@tsugumi ~ $ echo $?↵
0
user@tsugumi ~ $ false↵
user@tsugumi ~ $ echo $?↵
1
```

Аргументы программы и её код возврата позволяют передавать в и из программы лишь очень мало информации, основным средством взаимодействия с внешним миром для программы являются файлы. Одной из фундаментальных идей, которые были положены в основу ОС UNIX (для написания которой и создавался язык C), была «файлы — это все объекты ОС, допускающие операции ввода-вывода». Под операциями ввода-вывода понимаются, в первую очерель, чтение и запись последовательностей байтов. Этих принципов во многом придерживаются современные представители семейства UNIX, включая Linux, и частично другие ОС. Таким образом, к файлам относят не только поименованные последовательности байтов на устройствах хранения, но и большинство устройств, как физических, так и существующих только в терминах операционной системы. В первую очередь файлом является сам терминал: с точки зрения программы чтение с терминала — это получение вводимых с клавиатуры символов, а запись — вывод на терминал последовательной текстовой информации.

Для работы с тем или иным файлом программа должна его сначала «открыть» — указать ОС требуемый файл и запрашиваемые над ним операции. ОС проверяет наличие требуемых файлов и прав у выполняемой программы на совершение с ним указанных операций, выделяет ресурсы для работы программы с файлом, и, в случае успеха всего вышеперчисленного, возвращает программе некоторый идентификатор файла, который может использоваться программой для идентификации последующих операций над ним. По завершению работы с файлом, его необходимо «закрыть», сообщив ОС, что связанные с открытым файлом ресуры более не требуются. Обратите внимание, что эти опеределения понятия открытия и закрытия файла не обязаны (и часто не соответствуют) понятиям открытия/закрытия файла с точки зрения пользователя в прикладных программах. Это и другое взаимодействие с ОС требует специальных машинных инструкций для передачи управления ОС, современным вариантом на 32-битной архитектуре x86 является инструкция `sysenter`.

Поскольку вообще без использования файлов мало какая программа может обойтись вообще, на момент запуска программы три файла для неё уже открыты самой ОС под общеизвестными идентификаторами: файлы ***стандартного (standard) ввода (input)***, ***вывода (output)*** и ***ошибок (error)***. Большинство программ, особенно с текстовым интерфейсом, используют их для соответственно чтение входных данных, записи выходных данных и записи диагностических сообщений об ошибках, если иное не указано аргументами программы. Какие файлы соответствуют этим открытым идентификаторам, определяет программа, запускающая данную, если это специально не указано иначе, они такие же, как у этой программы. Когда эмулятор терминала запускает первую программу в новом окне (обычно оболочку), он создаёт виртуальное устройство-терминал, и устанавливает для запускаемой программы все три стандартных открытых файла в него (первый открыт на чтение, последние два — на запись). Таким образом, без специальных указаний терминал является основным файлом, с которым осуществляется обмен информацией оболочкой и запускаемыми из неё программами.

Это можно изменить с помощью элементов языка оболочки, называемых ***перенаправлениями (redirections)*** (каждый из них сам может трактоваться как команда, что позволяет

их комбинировать):

- `команда < имя-файла` — открыть в качестве стандартного ввода указанной файл для запускаемой команды. Это позволяет подготавливать входные данные для программ заранее и использовать их многократно.
- `команда > имя-файла` — открыть в качестве стандартного вывода указанной файл для запускаемой команды (содержимое файла уничтожается). Это позволяет сохранить результат работы программы в файл вместо вывода на экран.
- `команда | команда` — так называемый ***конвейер (pipeline)***. Оболочка открывает пару специальных файлов — ***канал (pipe)***, один на чтение, другой на запись. Канал — пара специальных файлов, при записи в один конец которой те же данные считываются из другого. Входная часть трубы назначается стандартным выходом первой команды, а выходная — стандартным входом второй. Обе команды запускаются одновременно, оболочка дожидается завершения обеих. С помощью конвеера можно организовать параллельную обработку информации несколькими программами по цепочке без создания промежуточных файлов на диске.

Предположим, что имеется текстовый файл `a.txt`, содержащий список слов по одному на строку. Команда `sort < a.txt | uniq > b.txt` создаёт файл `b.txt` с отсортированным списком без дубликатов:

1. Программа `sort` считывает строки из стандартного файла ввода, который перенаправлен из `a.txt`.
2. Считанные строки сортируются и записываются в стандартный файл вывода, которым является входная часть трубы.
3. Программа `uniq` считывает строки из стандартного файла ввода, которым является труба, т.е. выходные данные программы `sort`.
4. Среди считанных строк удаляются подряд идущие одинаковые. Т.к. они уже отсортированы, это устраняет все дубликаты. Результат записывается в файл стандартного вывода, который перенаправлен в `b.txt`.

В наших первых программах мы будем работать только со стандартными файлами, чтобы обойтись без явного их открытия и закрытия.

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. Помимо нескольких версий 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. Перечислим основные опубликованные версии:

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 в настоящее время завершена. Ей осталось преодолеть последние формальные процедуры принятия в ISO, которые должны завершиться до конца 2017-го года. Её неформальное обозначение C++1z, уже начинают менять на C++17. Основные компиляторы C++ уже поддерживают большинство возможностей этого стандарта. Эта версия содержит достаточно большое количество изменений, которые в сумме также весьма значительны, кроме этого некоторые расширения языка, до этого существовавшие в качестве отдельных необязательных документов, стали официальной частью языка. Это первая версия C++, в которой видны результаты перехода на разработку отдельных крупных частей языка в виде самостоятельных стандартов.
7. В июле 2017 года началась работа над следующей версией стандарта языка C++, планируемой к 2020-му году. В рабочей версии, называемой C++2a, пока содержится несколько незначительных изменений. Основной интерес к этой версии связан с вероятным включением двух давно ожидаемых крупных изменений — концепций и модулей.

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 настоящее время в виде документа N4700 [**?**]. Этот документ является рабочей версией нового стандарта, свежие его версии публикуются на официальном сайте `http://isocpp.org`.

3.1. Язык C++ в виде стандарта ISO/IEC

Международный стандарт языка является основным документом, определяющим отношения между программистами и создателями сред, в которых могут транслироваться и исполнятся программы на языке C++. Не имеющие данного статуса черновики, тем не менее, имеют ту же самую структуру.

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. Элементами ***видимого поведения (observable behavior)*** программы являются:

1. Записанная в файлы информация;
2. Динамика интерактивных устройств: запросы к вводу данных должны появляться до ожидания этих данных;
3. Обращения к особо помеченным участкам памяти, которые имеют особое, неизвестное транслятору поведение.

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 для повторного

использования множества объектных файлов) объединяются для получения ***образа программы (program image)*** — полного объёма информации, необходимого для выполнения программы в среде выполнения.

3.3. Классификация ошибок в программах

На всех рассмотренных этапах трансляции и выполнения программы, она может быть подвержена наличию в ней ошибок. Ошибкам необходимо дать классификацию, что позволит в дальнейшем говорить о видах ошибок с точки зрения способов их поиска и устранения.

1. ***Синтаксические ошибки (Syntax errors)*** — ошибки, являющиеся нарушением правил синтаксиса языка. Это самый простой в обнаружении класс ошибок, поскольку практически с гарантией распознаётся компилятором в процессе его работы. Исправление их обычно также достаточно легко. Непарные скобки и другие забытые элементы синтаксиса — типичные представители этого класса ошибок. Основные трудности на этом уровне — иногда синтаксис нарушается таким образом, что транслятор не может восстановиться и продолжить анализировать текст программы далее, что приводит к некорректной диагностики места реальной ошибки, а также нахождению многих несуществующих ошибок в дальнейшем тексте.

   В случае с интерпретируемыми языками обнаружение ошибок синтаксиса может отложиться на момент выполнения программы, что является слабым местом таких языков.

2. ***Семантические ошибки (semantic errors)*** — ошибки, нарушающие семантику языка. Эти ошибки могут обнаруживаться как на этапе трансляции (при использовании компиляторов), так и на этапе выполнения. Стандарт языка C++ предписывает, какие из таких ошибок должны диагностироваться компилятором, для остальных это вопрос качества реализации компилятора.

3. ***Логические ошибки (logic errors)*** — несоответствие задачи, решаемой программой, с требуемой. Программа может быть полностью корректна с точки зрения самого языка программирования, но решать не ту задачу, которую должна была решать по задумке программиста. Для императивного языка это ошибочный перевод алгоритма на язык программирования, например ошибки в формулах (программа удвоения чисел утраивает их) или выполняемых действиях (кнопка удаления элемента списка в интерфейсе программы добавляет новый). Логические ошибки, вызванные опечатками, обычно легко исправить, но нередки и сложные случаи, когда в процессе отладки выявляется редкий крайний случай решаемой задачи, который не был рассмотрен при построении и программировании её решения.

4. ***Ошибки времени выполнения (runtime errors)*** — ошибки, вызванные средой выполнения программы. Даже полностью корректная с точки зрения предыдущих пунктов программа может не смочь выполнить свою задачу, если в процессе её работы возникнет ошибка в компоненте среды выполнения программы, от неё не зависящей. Например, программа, требующая для работы чтения данных из некоторого файла может не обнаружить его при попытке открытия, поскольку он был удалён пользователем. Избежать возникновения ошибок времени выполнения в общем случае невозможно, но они не являются сами по себе фатальными для программы. Основной задачей программиста в борьбе с этими ошибками является проверка всех ситуаций, в которых они могут возникать, и принятие решения о действиях в случае их возникновения. Такие действия могут быть различными в зависимости от того, насколько невозможность выполнения действий, завершившихся ошибочно, критично для функциональности программы, а также от роли самой программы в функционировании компьютера.

   Например, разрыв соединения с удалённым сервером при загрузке файла из Internet может быть обработан попыткой повторно установить соединение некоторое фиксированное число раз, и только многократное возникновение этой ошибки приведёт к заврешению работы программы с сообщением об ошибке. Даже в этом случае, это штатное завершение программы в соответствии с семантикой языка программирования. Если речь идёт о серверном ПО, позволяющим множеству пользователей всемирной сети загружать файлы, то разрыв отдельного соединения с клиентом никак не влияет на его общую функциональность и, скорее всего, даже не требует действий по обработке — кроме освобождения используемых соединением ресурсов и журналирования. С другой стороны, обнаружение нехватки оперативной памяти для функционирования большинства прикладного ПО может быть

фатально, но системные приложения, обслуживающие основные функции операционной системы будут пытаться сохранить свою остальную функциональность до последнего.

Мы перечислили разновидности ошибок в порядке, в котором о них обычно следует говорить, поскольку наличие ошибок предыдущего уровня часто делает разговор о следующих уровнях бессмысленным, по крайней мере в том же рассматриваемом элементе программы.

Глава 4

От теории к первой программе

В этой главе мы введём минимальные сведения о языке и инструментальных средствах, которые позволят нам написать и запустить простейшую программу.

4.1. 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 одному из четырёх 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  ?    ::  .    .*    ;    :    ...
+    -       *    /   %    ^     &    |    ~
!    =       <    >   +=   -=    *=   /=   %=
^=   &=      |=   <<  >>   >>=   <<=  ==   !=
<=   >=      &&   ||  ++   --    ,    ->*  ->
```

> В языке также имеются так называемые ***альтернативные представления (alternative representation)*** некоторых токенов, которые полностью им эквивалентны. Наиболее известные из низ — `and`, `or` и `not`, являющиеся альтернативными способами записи токенов `&&`, `||` и `!` соответственно. Такая форма записи может быть более знакома читателю по другим языкам программирования, но в языке C++ не принята, и практически никогда не используется. Не все программисты в принципе знакомы с тем, что подобные формы существуют и допустимы, кроме этого отдельные компиляторы имеют проблемы с их использованием. По этим причинам применять их не рекомендуется.
>
> Другая группа альтернативных форм избегает в написании некоторых операций символа `#`, квадратных и фигурных скобок, отсутствующих в некоторых исторических кодировках. Наконец, ещё одна группа альтернативных форм — триграфы — была упразднена в C++17.
>
> Для подавляющего большинства программистов эта информация несёт лишь историческую ценность.

- ***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; // Два токена: int (ключевое слово) и a (идентификатор).
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 `;`.

4.2. 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, также являются выражениями — их простейшей формой.

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 наделяется ***категорией значения (value category)*** — дополнительной характеристикой, которая определяет применимость этого значения в других конструкциях языка.

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» (в англоязычном термине отрицательного оттенка нет) — 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, соответствующие входным параметрам процедурной парадигмы программирования. Функция также ***возвращает (return)*** некоторое значение в качестве результата своего выполнения — всегда один выходной параметр с точки зрения процедурной парадигмы.

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) вводят в программу имена сущностей и определяют их свойства. Тем самым описания задают смысл используемых в программе имён. ***Определение (defini-***

tion) — частный случай описания, который помимо перечисления минимально необходимых свойств имени задаёт содержимое сущности, ему соответствующей. Одним из основополагающих правил в языке C++ является ***правило одного определения (one definition rule (ODR))***: для каждой элемента программы не может существовать более одного определения. При этом для тех сущностей, для которых в программе требуется знание их содержимого, такое определение должно присутствовать.

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.***

4.3. 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          std::cerr << "Input error!\n";
9          return 1;
10     }
11     c = a*b;
12     std::cout << a << " * " << b << " = " << c << '\n';
13 }
```

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 возвращаемое значение передаётся операционной системе в качестве сигнала успешности работы программы. 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 во внутренних 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 преобразуем к логическому значению, характеризующему успешность этой операции. После применения к нему операции логического отрицания, это значение истинно в случае неудачи ввода. 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 не успешен.

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

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

   Токенов, являющихся литералами, в этой строке нет.

8. Выражение на этому строке выводит в стандартный файл вывода ошибок сообщение о неудаче ввода.

9. Оператор на данной строке завершает выполнение функции `main` и, следовательно всей программы, с ошибочным ненулевым кодом возврата 1.

10. Закрывающая фигурная скобка на данной строке обозначает конец блока условно выполняющихся команд, начатый в строке 7.

11. 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`.

12. Выражение в данной строке в качестве побочного эффекта выводит на стандартное устройство вывода исходные данные и результат вычислений.

13. 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.

В оставшейся части этого раздела мы дадим минимальные сведения о структуре только что рассмотренной 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.

4.4. Основные арифметические типы данных

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

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)***.

4.4.1. 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.

	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

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`.

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: $-(2_{(10)}) = -(010_{(2)}) = \mathit{bitwiseNot}(010_{(2)}) + 1 = 101_{(2)} + 1 = 110_{(2)} = -2_{(10)}$, где $\mathit{bitwiseNot}$ — 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` на большинстве архитектур играет роль типа минимальной ширины, 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 рассматрива-

ются 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 x86-64, 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 x86-64 использует, представление little-endian, которое исторически не было доминирующим, но в настоящее время становится основным для большинства новых архитектур.

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`.

Для представления специальных значений внутри символьных литералов используются ***escape-последовательности***, начинающиеся со знака `\`. В процессе предварительной об-

работки 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.

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. Для представления последовательностей символов можно использовать строковые литералы, имеющие другой синтаксис — их мы рассмотрим позднее.

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.

4.4.2. 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, больша́я 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 [3], 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

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

4.4.3. 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 арифметическими 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 преоб-

разования 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.

- ***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.

4.5. Основные выражения с арифметическими операндами

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

4.5.1. 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`.

4.5.2. 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. Для двух арифметических типов 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:

$$(a/b) * b + a\%b = a. \tag{4.8}$$

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.

4.5.3. 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.

4.5.4. 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
    …

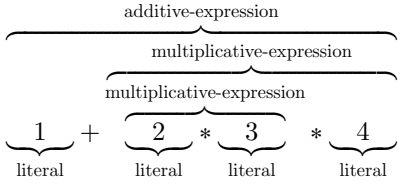
pm-expression:
    cast-expression
    …

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.

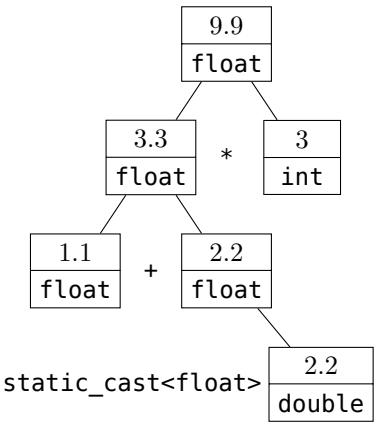


Рис. 4.1: Дерево выражения `(1.1f+static_cast<float>(2.2))*3`

4.6. Объекты и доступ к ним

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

4.6.1. Описания

Описание (declaration), как было сказано ранее, служит для введения в программу имён и задания характеристик соответствующих им сущностей. В языке C++ существует несколько форм описаний, рассмотрим основную из них.

Простое описание (simple declaration) состоит из трёх частей:

1. Последовательность ***спецификаторов описания (declaration specifier)***, задающих свойства, общие для всех вводимых в этом описании имён. Они могут быть даны в любом порядке. Обязательным среди них является ***спецификатор типа (type specifier)***, задающий основу типа описываемых сущностей.
2. Последовательность ***описателей (declarator)*** — частей описания, каждая из которых указывает вводимое в программу имя и, возможно, конструкции создания производных типов, задающие построение типа соответствующей сущности относительно типа, заданного спецификаторами типа в начале описания. Описатели разделяются пунктуатором «запятая», если их несколько.
3. Пунктуатор точка с запятой.

Для нашей текущей задачи потребуются два дополнительных утверждения. Во-первых, имена фундаментальных типов в виде последовательностей соответствующих ключевых слов являются спецификаторами типа. Во-вторых, простейшая форма описателя без конструкций создания производных типов состоит из одного идентификатора, который и вводит в программу соответствующее имя как имя сущности типа, заданного спецификаторами типа в описании.

Мы будем в дальнейшем расширять эту схему описания по мере изучения системы типов языка.

> Простое описание может не содержать описателей вообще, поскольку имеются формы спецификаторов описания, также вводящие имена в программу — мы познакомимся с ними в будущем. Если последовательность спецификаторов описания сама по себе имён не вводит, в описании должен присутствовать хотя бы один описатель, чтобы описание выполняло свою основную роль. В качестве исключения, полное отсутствие первых двух частей в виде конструкции из одного пунктуатора «точка с запятой» называется ***пустым описанием (empty declaration)*** и допускается без каких-либо эффектов для содержащей его программы.

4.6.2. Простые определения объектов

Рассмотрим пример простого описания:

```
int x,y;
```

Неформально его следует читать как «следующие конструкции соответствуют сущностям типа `int`: идентификатор `x` и идентификатор `y`».

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`. Отдельные идентификаторы — опять же лишь частный случай имён, которые в общем случае могут состоять из нескольких различных токенов.

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 имена, в частном случае — идентификаторы, могут выступать в качестве элементов выражений.

4.6.3. Основные категории значений

В языке C++ значения выражений классифицируются с точки зрения ***категорий значений (value category)***, которые являются ещё одной характеристикой, помимо типа, определяющей допустимость применения этих значений в том или ином контексте. Основных характеристик значения в языке C++ две.

Результат вычисления ***обобщённого леводопустимого выражения (glvalue)*** идентифицирует объект или функцию в программе. Такие выражения необходимы там, где требуется подобная идентификация. Большинство обобщённых леводопустимых выражений, которые мы пока будем рассматривать, являются частным случаем, называемым просто ***леводопустимыми (lvalue)*** выражениями — они идентифицируют объекты, которые всё ещё нужны в программе. Для таких выражений идентифицируемая сущность также называется их ***результатом (result)***. Имена объектов являются леводопустимыми выражениями, идентифицирующими объекты, которым эти имена были назначены соответствующими определениями. Мы будем использовать оба термина в соответствии с правилами языка, разница между ними будет показана позднее, когда нам потребуется вторая половина обобщённых леводопустимых выражений — ***истекающие (xvalue)***, идентифицирующие объекты, значения которых более не требуются и соответствующие им ресурсы могут быть использованы для других нужд.

Результат вычисления ***чисто праводопустимого выражения (prvalue)*** предназначен для использования в качестве операндов дальнейших операций или задания начальных значений объектов. Эту категорию значения имеют все ранее рассмотренные литералы и результаты арифметических и логических операций. Кроме этого, данные операции, как и любые другие, если не указано обратное, требуют чисто праводопустимых значений в качестве своих операндов. Другая роль чисто праводопустимых значений — задание начального значения объектов — будет рассмотрена далее.

> Операция `static_cast` с известными нам типами возвращает чисто праводопустимый результат, поэтому её формально можно использовать для явного выполнения преобразования леводопустимого значения за счёт приведения объекта к его собственному типу. Подобного для арифметических типов можно также добиться операцией сохранения знака, которая также выполнит целочисленные повышения. Такие преобразования сами по себе требуются редко, поскольку в большинстве случаев выполняются неявно там, где требуются.

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

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, однако имена объектов являются леводопустимыми выражениями, а рассмотренные операции в качестве операндов требуют чисто праводопустимые значения. Использование имён объектов в качестве таких операндов достигается за счёт неявного стандартного преобразования, называемого ***преобразованием леводопустимого выражения (lvalue-to-rvalue)***: обобщённое леводопустимое выражение может быть преобразовано к чисто праводопустимому того же типа со значением, соответствующим текущему значению идентифицированного им объекта. Это преобразование есть формальное представление в языке понятия «чтение 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.

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

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 вычисленное значение её левого операнда, также сохраняющее категорию значения lvalue. 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.

Рассмотрим примеры.

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

Перед нами два определения. Теперь в нашем распоряжении есть три объекта, хранящие значения типа `int`, идентифицируемые как `a`, `b` и `c` — это строго формальная интерпретация. Обычно говорят проще, например, 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 и объектами нет отношения один-к-одному: одному объекту может соответствовать несколько выражений, в том числе более сложных, чем отдельное имя, бывают и безымянные объекты. Во-вторых, сам объект является просто некоторой областью памяти, которая может иметь разную интерпретацию. И объекты в памяти, и значения, идентифицирующие их имеют типы, которые должны быть одинаковы, чтобы интерпретация памяти в качестве того или иного значения была осмысленной. В данном случае описание задаёт тип сущности, соответствующей описанному имени, и именно его тип используется для интерпретации содержимого объекта 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 пунктуатор

«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.

Рассмотрим более сложный случай:

```
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.

Рассмотрим пример, в котором имеют место оба вида доступа к объектам:

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

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.

Ещё одной классической ошибкой является трактовка операции присваивания или другой важной операции, несущей основной смысл в выражении оператора-выражения, как самостоятельного оператора. Например, ошибочно упоминают «оператор присваивания». С одной стороны, это проблема запутывающего перевода терминологии: «statement» — оператор, но «operator» – операция. С другой — это следствие того, что оператор-выражение может выполнять бесконечное количество принципиально различных действий, зависящих от операций в его выражении.

Специальная форма оператора-выражения, которая не содержит выражения, т.е. оператор из одного токена «точка с запятой» называется ***пустым оператором (empty statement)***. Он не предписывает никаких действий и применяется там, где в языке синтаксически требуется оператор, но действий по алгоритму не требуется.

4.7. Составной оператор (блок)

В предыдущих разделах нами были рассмотрены описания объектов и оператор-выражение. Последний вместе с изученными нами операциями позволяет производить вычисления с использованием памяти в виде объектов для хранения значений. Для записи последовательностей действий из больше чем одной такой конструкции потребуется рассмотреть вопрос о реализации в языке C++ последовательности в смысле парадигмы структурного программирования.

Составной оператор (блок) (compound statement (block)) — оператор, предписывающий выполнить последовательно указанные в нём операторы. Синтаксически он состоит из последовательности операторов в порядке их требуемого выполнения, заключённой в фигурные скобки. Многие формы описания, включая рассмотренное нами простое, могут выступать в роли операторов и, следовательно, входить в блоки. Также отметим, что использование подобных описаний в блоках также является определениями.

```
1  {   // Начало блока, содержимое выполняется последовательно.
2      int x;
3      x = 1;
4      x = x+1;
5      // После выполнения определения и двух операторов-выражений,
6      // текущее значение объекта x есть 2.
7  }   // Конец блока
```

С одной стороны, блок содержит в себе операторы, а с другой — сам считается оператором, поэтому блоки могут содержать в себе другие блоки:

```
1  {   // Блок предписывает выполнить своё содержимое по порядку:
2      // Сначала описание.
3      double z;
```

```
4       // Затем ещё один блок, который как часть своего выполнения
5       {
6           // Предписывает выполнить оператор-выражение,
7           z = 1;
8           // а затем ещё один.
9           z = 2;
10      }
11      // Вложенный блок выполнен, дальше ещё один
12      // оператор-выражение.
13      z = 3;
14  }   // Выполнен первоначальный блок.
```

Пустой блок `{}` допустим и аналогичен пустому оператору по требуемым к выполнению действиям — никаким.

> Там, где требуется оператор, одинокая точка с запятой трактуется именно как пустой оператор, а не пустое определение. Технически, от другого варианта трактовки ничего не изменилось бы, но отдельная точка с запятой в реальности всегда является именно пустым оператором, т.к. случаи где требуется синтаксически, но не нужен оператор на практике имеются, а для описаний таких контекстов нет.

4.7.1. Характеристики сущностей в описаниях

Как было заявлено ранее, описания задают характеристики описываемых ими сущностей. Две из них нам уже известны: тип и является ли описание определением. Введём оставшиеся характеристики, поскольку они задают поведение уже используемых нами описаний. В дальнейшем мы рассмотрим как они зависят от положения описания в программе и наличия в описании дополнительных опциональных частей.

В первую очередь положение описания в программе определяет его ***область видимости (scope)***.

Каждое упоминание имени в программе требует от транслятора нахождения описания, в котором оно описано, чтобы понять, о какой сущности в данном месте говорит программист. (Кроме, разумеется, использования имени в описателе собственного описании.) Алгоритм сопоставления имени описанию называется ***поиском имён (name lookup)***. Этот в целом непростой алгоритм мы будем рассматривать по частям.

Максимальная область программы, в которой поиск имён может находить данное имя, называется ***потенциальной областью видимости (potential scope)***, хотя действительная область видимости может оказаться уже за счёт других описаний. Разные варианты алгоритма поиска имён сводятся к последовательному просмотру некоторой последовательности областей видимости до тех пор, пока в очередной из них не будет найдено описание искомого имени. Если все области видимости исчерпаны, а описание имени не найдено — это синтаксическая ошибка в программе.

Два основных варианта алгоритма поиска имён выбираются исходя из устройства имени. В языке C++ имя может содержать в себе указания областей видимости, в которых его следует искать, такие имена называют ***квалифицированными (qualified)***. Имеющиеся в нашем распоряжении на данный момент имена в виде отдельных идентификаторов не содержат такой информации и относятся к ***неквалифицированным (unqualified)***.

В общем случае несколько сущностей в программе могут использовать одно и то же имя, если даны в разных областях видимости. Какая из них будет найдена в том или ином случае и определяется алгоритмом поиска имён. Однако не каждое описание имени, в том числе, в той же области видимости, обязательно соответствует новой, уникальной сущности — правило одного определения не запрещает, помимо одного определения, иметь сколько угодно описаний той же самой сущности. Характеристика имени в описании, задающая потенциальную возможность других описаний того же имени в программе именовать ту же самую сущность, называется ***связанностью (linkage)***. Все связанные описания должны давать сущности одно и то же имя, тип и другие характеристики, т.к. речь идёт об одной и той же сущности.

Отметим, что возможны ситуации, когда в одной области видимости может находиться несколько описаний с одним именем, и алгоритм поиск имён встретит их одновременно. Если все они являются связанными описаниями одной и той же сущности, проблемы не возникает, т.к. результат поиска всё равно однозначен. Другое очень важное исключение, где выбор между несколькими одноимёнными сущностями может быть осуществлён как дополнительный

шаг алгоритма поиска имён, будет рассмотрен позднее, но за этим исключением несколько описаний разных одноимённых сущностей в одной области видимости не допускаются.

Пока мы рассматривали только описания объектов, т.к. все известные нам типы являются типами данных. ***Для объектов определение отличается от описания, им не являющегося, тем, что отвечает за выделение памяти среды выполнения под описываемые объекты.*** В таком случае встаёт вопрос о том, когда эта память выделяется и, соответственно, освобождается, чтобы вернуть использованную под объект память. Характеристика объекта, определяющая это, называется ***время хранения (storage duration)***.

4.7.2. Характеристики простых описаний объектов в блоках

Рассмотрим характеристики простых описаний известного нам вида: описания объектов в блоке без каких-либо спецификаторов описания, кроме типа.

Имена, описанные в блоке, обладают ***блочной видимостью (block scope)***. Их потенциальная область видимости и есть соответствующий блок.

Алгоритм поиска неквалифицированного имени, использованного в блоке, просматривает последовательность блоков, начиная с того, в котором имя упомянуто. Следующим при неудаче нахождения описания просматривается блок, окружающий данный, и далее по последовательности окружающих блоков. В каждом блоке поиск осуществляется до точки использования имени, но не после.

Возможна ситуация, когда в области видимости, просматриваемой поиском имён раньше, описана другая сущность с тем же именем, что и в следующей. Исходя из описанного выше алгоритма поиска имён, при использовании имени в первой области видимости находится описание, данное в ней, и на этом алгоритм останавливается. Описания вида, рассматриваемого нами, ***связанности не имеют***, поэтому каждое такое описание говорит о новой сущности, даже если у неё имя, тип и другие характеристики такие же. В таком случае неквалифицированным именем поименовать сущность, соответствующую описанию в области видимости, просматриваемой позднее, нельзя — такая ситуация называется ***скрытием имён (name hiding)***.

Также такие описания являются определениями. Определяемые в них объекты имеют ***автоматическое (automatic)*** время хранения: они создаются перед выполнением операторов блока, в котором описаны, и уничтожаются сразу после завершения выполнения этого блока. Для рассматриваемой нами формы описателя и автоматического времени хранения значение объекта после создания не детерминировано. Хотя соответствующая область памяти содержит некоторую информацию, которую можно пытаться трактовать с точки зрения типа этого объекта, это бессмысленно в силу непредсказуемости результата этого процесса. Из-за наличия специфики работы некоторых аппаратных архитектур, чтение не детерминированных значений большинства типов ведёт к неопределённому поведению, поэтому первым доступом к таким объектом должна быть запись, задающая конкретное значение. Несмотря на наличие исключений, автор предлагает чтение не детерминированного значения любого типа считать ошибкой в программе, поскольку даже если это не является не определённым поведением, дальнейшее поведение программы, зависящее от непредсказуемого значения корректным являться не может. (Вопросы построения не детерминированных программ, например, игр и моделирования случайных процессов будут рассмотрены отдельно и этого правила не нарушают.)

Может показаться, что область видимости и время хранения для этих объектов в некотором роде «совпадает» и ограничено блоком, но в общем случае такой связи не наблюдается. Рассмотрим примеры:

```
1  {   // Вход в блок: выделяется память для объектов a и b
2      // в следующем определении.
3      int a,b;
4      // Находит a, описанное выше в этом блоке.
5      a = 3;
6      {   // Выделяется память под объект с именем b, описанный ниже.
7          // Это другой объект, отличный от одноимённого, созданного выше.
8
9          // Нет описаний a до точки использования в этом блоке,
10         // выходим в окружающий блок где до точки использования
11         // описание имеется.
12         a = 5;
13         // Описание, которое скрывает данное в окружающем блоке.
14         // Реальная область видимости первого описания b выше
```

```
15          // имеет "дырку" в виде области от описания ниже до конца
16          // этого блока.
17          int b;
18          // Находит описание b в этом блоке и останавливается.
19          // Описание в окружающем блоке скрыто и не доступно.
20          b = 4;
21      }   // Конец внутреннего блока. Объект b уничтожается и
22          // занимаемая им память освобождается.
23      // Просматривается данный внешний блок до этого момента,
24      // так что ближайшее описание во внутреннем блоке на находится.
25      // Вместо этого находится первое описание, т.к. оно более не скрыто.
26      b = 7;
27
28      // ОШИБКА: использование не описанного имени: просмотр осуществляется
29      //         только до точки использования.
30      c = 9.;
31
32      double c;
33  }
```

4.8. Основные виды инициализации

Процессы создания и уничтожения объекта, на самом деле, не обязательно связаны с выделением и освобождением памяти. Создание объекта может включать задание его начального значения — ***инициализацию (initialization)***. Некоторые типы объектов также требуют действий, которые необходимо выполнить для их уничтожения. Однако в большинстве случаев, включая рассматриваемые нами на данный момент, эти действия выполняются неделимыми парами: определение выделяет под хранение объекта память и инициализирует в ней объект, создавая его, а при выходе из блока происходят действия, необходимые для уничтожения объектов, а затем память освобождается.

В языке C++ имеется несколько видов инициализации, имеющие как синтаксические проявления, так и происходящие в тех или иных контекстах языка без явного указания. Рассмотрим наиболее часто использующийся вид инициализации — ***инициализация копированием (copy initialization)***. Опциональной частью описателя, записываемой в его конце, является ***инициализатор (initializer)*** — конструкция, указывающая способ инициализации объекта. Синтаксическая форма инициализатора в форме инициализации копированием — пунктуатор точка с запятой и выражение за ним. Оно предписывает задать объекту значение указанного выражения после его неявного преобразования в значение типа инициализируемого объекта категории prvalue. Как мы определили ранее, это и есть вторая задача значений этой категории — задавать начальные значения объектов. В этой роли у prvalue имеется ***объект-результат (result object)*** — тот объект, для которого оно задаёт начальное значение:

```
1  int x = 3, // Начальное значение x - 3.
2      y = x; // x - lvalue, преобразуется в prvalue 3,
3             //   и это начальное значение y.
```

В случаях, когда инициализатор в описателе отсутствует, производится ***инициализация по умолчанию (default initialization)***. В целом по смыслу этот вид инициализации предписывает выполнить минимальные необходимые действия для задания начального значения объекта. Для всех фундаментальных типов в языке такие минимальные действия отсутствуют, так что выполнение над объектами таких типов инициализации по умолчанию оставляет в их представлении те значения байтов, которые хранились в только что выделенной памяти ранее, т.е. неуточняемые значения. В случае, когда инициализация объекта не включает выполнение явного указанных программистом действий, а действий, фактически выполненные компилятором, нет, говорят, что инициализация ***тривиальна (trivial)***. В общем случае чтение неуточняемых значений объектов ведёт к неопределённому поведению, поэтому для объектов, инициализированных по умолчанию, первой операцией доступа должна быть запись:

```
1  {
2    int x = 1;
3    // x инициализировано копированием и имеет конкретное значение,
4    // можно его считывать.
5    int y = x+2;
```

```
6
7     int z1,z2;
8
9     // z1 инициализировано по умолчанию, неучтоняемое значение
10    // может быть заменено конкретным операцией записи.
11    z1 = 3;
12
13    // Чтение z1 допустимо, в него уже была запись, значение детерминировано.
14    z1 = z1+1;
15
16    // ОШИБКА: неопределённое поведение, чтение недетерминированного значения.
17    z1 = z2+1;
18  }
```

Предпочитайте по возможности создание объектов с детерминированными значениями, поскольку изменения алгоритма использования объекта после его создания могут незаметно привести к изменению первой операции доступа к нему.

> Автор в данном разделе несколько ужесточил требования, относительно реальных правил языка C++. Для некоторых типов чтение их недетерминированных значений заведомо не является неопределённым поведением. Однако трудно представить себе корректный алгоритм, опирающийся на непредсказуемые значения в его вычислениях. Автор предлагает вместо перечисления этих типов-исключений, указанных в стандарте, объявить чтение недетерминированных значений любых типов неопределённым поведением. В том числе такой подход считают корректным многие инструментальные средства, которыми нам предстоит пользоваться.
>
> Как вообще чтение недетерминированного значения может вести к неопределённому поведению? Некоторые значение специальных типов, например, с плавающей точкой могут иметь специальные значения, сам факт формирования которых в результате вычисления ведёт к исключительным ситуациям, не описываемым стандартом языка C++. Чтение памяти с недетерминированным содержимым всегда имеет шанс чтения такого значения в регистр процессора и возникновения подобной ситуации.

4.9. 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, за исключением неприменимости их к операнду типа `bool`. 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 постфиксных форм операций инкремента и декремента является чисто праводопустимым.

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

```
6       y = x++; // x == 9 && y == 8
7   }
```

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.

Когда выражение содержит побочный эффект и другие доступы к объекту, следует быть особенно осторожным. В общем случае следует избегать использования значения объекта, который изменяется побочным эффектом выражения, в том же полном выражении, за исключением простого случая, когда происходит чтение значения объекта-левого операнда операции присваивания справа от него. Следующее выражение корректно:

```
i = i+1;
```

В свою очередь неопределённое поведение имеют следующее выражение:

```
++i+i;
```

В сложных случаях проще разбить выражение на несколько более простых, чтобы не рисковать.

Формально данная проблема описывается следующим образом. ***«Упорядочен перед» (sequenced before)*** — ассиметричное, транзитивное, попарное отношение между выполняемыми вычислениями. Т.е. это потенциальное отношение порядка между двумя любыми вычислениями. Введём обозначение: $A \triangleleft B$ — « упорядочен перед B», что означает, что выполнение A происходит перед выполнением B. Ассиметричность отношения определяет, что

$$A \triangleleft B \rightarrow \neg(B \triangleleft A). \tag{4.10}$$

Если A упорядочен перед B, то B ***упорядочен после (sequenced after)*** A. Свойство транзитивности операции означает, что

$$A \triangleleft B \wedge B \triangleleft C \rightarrow A \triangleleft C. \tag{4.11}$$

Если A не упорядочен ни до, ни после B, то A и B ***не упорядочены (unsequenced)***. Если A упорядочен либо до, либо после B, но когда конкретно — не уточняется, то A и B ***неопределённо упорядочены (indeterminately sequenced)***. В общем случае, если ничего не сказано особо, два вычисления не упорядочены. Следующие утверждения задают упорядоченность вычислений для известных нам конструкций:

- Все вычисления и побочные эффекты, связанные с полным выражением, упорядочены до вычислений и побочных эффектов, связанных со следующим вычисляемым полным выражением.
- Вычисление значений операндов упорядочено до вычисления значения операции.
- Для префиксных форм операций инкремента и декремента побочный эффект упорядочен до вычисления результата, для постфиксных — после.
- Вычисление значения и побочные эффекты в левом операнде операций `||`, `&&` и операции запятая упорядочены до вычисления значения и побочных эффектов правого операнда.
- Побочный эффект обновления значения объекта, идентифицируемого левым операндом операций присваивания, упорядочен после вычисления обоих операндов (но не их побочных эффектов) и до вычисления значения самой операции.

Правило, определяющее корректность выражений с побочными эффектами таково: ***если побочный эффект над скалярным объектом не упорядочен относительно другого вычисления значения или побочного эффекта над тем же объектом и не применимы специальные правила для параллельных вычислений,***

поведение не определено. Если допустимы несколько различных упорядочиваний подвыражений, поведение не определено, если указанное происходит хотя бы в одном из вариантов.

Постараемся разобрать приведённые выше примеры с учётом данной теории.

```
i = i+1;
```

Это выражение имеет побочный эффект операции присваивания, связанный с объектом `i`. Значение этого объекта в выражении вычисляется дважды: как левый и как подвыражение в правом операнде. Вычисление левого операнда упорядочено до побочного эффекта по определению операции присваивания. Вычисление правого операнда операции присваивания также упорядочено до побочного эффекта. Поскольку вычисление операнда `i` также упорядочено до вычисления значения операции `+`, то по свойству транзитивности вычисление значения `i` в правом подвыражении упорядочено до побочного эффекта изменения значения соответствующего объекта. Таким образом, вычисление обоих вхождений `i` в данном выражении упорядочены до побочного эффекта над ним, и поведение определено.

```
++i+i;
```

В данном случае имеются два доступа к объекту, оба затрагивающих объект `i`: побочный эффект и чтение, имеющиеся правила не позволяют упорядочить их каким-либо образом — поведение не определено.

Новые стандарты языка C++ значительно сократили объём случаев, где возможно подобное неопределённое поведение. Хотя на практике они встречаются редко, следует иметь о них понятие, чтобы правильно интерпретировать возможное предупреждение транслятора о них. Чаще всего подобные выражения являются результатом непонимания работы операций инкремента и декремента и лишних записей вида

```
i = ++i;
```

Подобная запись является неопределённым поведением в стандартах языка до C++11, но не в текущем. В любом случае, такие потенциально опасные конструкции использовать не следует.

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.

4.10. Пример компиляции выражения

Рассмотрим пример трансляции простого выражения в машинный код архитектуры x86-64, 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 LP64 требуется 4 байта, что соответствует половине машинного слова рассматриваемого 64-битного процессора. Поскольку размер типа `int` в данной модели памяти — 32 бита, т.е. не меньше, данный процессор работает и с такими, меньшими по размеру значениями, достаточно эффективно.

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 x86-64 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, обозначающими места в памяти.

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. В данной инструкции местом назначения является регистр `EAX`, а 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 x86-64, 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. 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.

С одной стороны, нехватка регистров может потребовать от компилятора сохранения промежуточных результатов в памяти, хотя им не соответствуют объекта, а, с другой стороны, при их достаточном количестве, объекты могут быть размещены непосредственно в регистрах без использования адресуемой памяти. Это далеко не единственные ситуации, где формальное наличие или отсутствие объектов может не соответствует машинному коду.

Ещё раз обратим внимание: пока результирующий код выполняет операции согласно контракту, закреплённому стандартом языка в виде понятия видимого поведения программы, 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.

4.11. Введение в функции

Процедуры как понятие процедурной парадигмы программирования в языке C++ называются ***функциями (function)***. Их преимущества таковы:

- Заданная функцией последовательность действий получает имя, которое облегчает понимание смысла программы её читателями. Даже если повторного выполнения не требуется, программа получает внешняя структуру с названиями для каждой части.
- 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.

Даже при записи простейших последовательных вычислительных алгоритмов часто возникают ситуации, когда та или иная последовательность действий встречается более одного раза. В нашей первой программе без функций обойтись не удастся по другой причине — язык C++, как наследник процедурного языка C, требует, чтобы все операторы, задающие выполняемые программой действия, находились в той или иной функции. В некоторых языках программирования допускаются операторы, записанные непосредственно в файле исходного текста без окружающих их конструкций, но к C++ это не относится.

Функции описываются синтаксисом простых описаний с некоторыми дополнениями. Тип описываемых в таких описаниях сущностей, т.е. функций, формируется с использованием конструкции создания производного типа «функция» в соответствующем описателе.

4.11.1. 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. «Параметры функции» в языке C++ соответствуют входным параметрам процедур.
- Тип возвращаемого функцией значения — тип значения (всегда ровно одного), которое становится доступно в точке вызова данной функции по завершению её исполнения в качестве «результата». У всех функций языка C++ формально всегда ровно один выходной параметр.

Описание функции связывает с именем функции тип, имеющий остальные перечисленные характеристики. Определение функции, в отличие от описания, им не являющегося, помимо этого задаёт саму последовательность действий, ей соответствующую. Синтаксически 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  }
```

Данное определение говорит о типе `double`, соответствующему конструкции `part_volume(double size,` являющейся сложным описателем — конструкция создания производного типа «функция» записывается как пара круглых скобок после части описателя, к которому применена, а в скобках даётся описание её параметров (входных). Тип, заданный спецификатором типа, указывает на тип возвращаемого значения функции — её выходной параметр.

Параметры перечисляются через запятую и задаются тем же образом, что и рассмотренные нами определения объектов. Каждый параметр является отдельным описанием с одним описателем, так что тип необходимо указывать для каждого параметра, даже если он один для нескольких подряд идущих. Определение функции отличается синтаксически от простого описания тем, что содержит единственный описатель, а точка с запятой в конце заменена блоком — телом функции. При выполнении тела функции описания параметров считаются определениями объектов с теми же характеристиками (блочная видимость, нет связанности, автоматическое время хранения), как если бы они были даны в начале этого тела. В отличие от других подобных объектов, их начальные значения детерминированы и задаются при вызове данной функции, что и реализует требуемый процедурной парадигмой механихм передачи входных параметров. 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`.

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, но так или иначе его выполнением необходимо завершать выполнение функции — выход из блока-тела функции естественным образом по его окончании ведёт к неопределённому поведению.

Определения функций не считаются простыми описаниями и не могут встречаться внутри других определений функций. Вложенных функций в чистом виде в языке C++ нет, альтернатива будет рассмотрена нами позднее.

4.11.2. Операция вызова функции

Рассмотрим теперь осмысленное использование имён функций, расширив предыдущий пример. Будем считать, что имя первой из двух данных функций видно поиску имён во второй, где используется.

```
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  }
10
11 double weight_diff_factor(double size,
12                            double old_radius,
13                            double new_radius,
```

```
14                          double thickness)
15 {
16     return part_volume(size,new_radius,thickness)/
17            part_volume(size,old_radius,thickness);
18 }
```

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. 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, инициализируются копированием из соответствующих значений аргументов, вычисленных значениям аргументов 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. Категория значения результата операции вызова функции определяется по его типу и, кроме специальных случаев, является prvalue. Результат для glvalue или результирующий объект для prvalue при этом инициализируется копированием из выражения в операторе `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++.

4.11.3. Структура единицы трансляции. Пространства имён.

Структура единицы трансляции в языке C++ проста — это последовательность описаний:

translation-unit:
    declaration-seq$_{opt}$

Формально, 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*$_{opt}$ `{` *declaration-seq*$_{opt}$ `}`

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. Эта конструкция является описанием имени пространства имён, вводя его имя в ту область видимости, в которой дано, хотя и не является частным случаем описания в стандарной форме.

Определение пространства имён является описанием, которое может быть дано в любом другом пространстве имён, включая глобальное, но не в других областях видимости, включая блоки. За счёт вложения определений пространств имён друг в друга можно строить произвольные иерархии, подобные дереву каталогов файловой системы, структурируя содержимое единицы трансляции. Приведём пример:

```
1  // Определение функции в глобальном пространстве имён.
2  int identity(int x)
3  {
4          return x;
5  }
6
7  // Определение пространства имён A
8  namespace A
9  {
10         // Определение функции в пространстве имён A
11         double sum(double x,double y)
12         {
13                 return x+y;
14         }
```

```
15
16          // Определение вложенного пространства имён B
17          namespace B
18          {
19                  // Определение функции в пространстве имён B, вложенном в A.
20                  bool always_true()
21                  {
22                          return true;
23                  }
24          }
25  }
```

Поиск не квалифицированных имён после выхода из блока-тела функции рассматривает пространства имён в порядке окружения, как обычно, до точки использования.

```
1   int f(int x)
2   {
3           return 2*x+3;
4   }
5
6   namespace A
7   {
8           int g(int x)
9           {
10                  // Поиск f: до точки использования в
11                  // - блоке-теле g,
12                  // - пространстве имён A,
13                  // - глобальном пространстве имён --- найдено!
14                  // Это была последняя просматриваемая область видимости,
15                  // если бы описание не было найдено и в нём, имела бы
16                  // место семантчиеская ошибка.
17                  return f(3);
18          }
19  }
```

Таким образом, описания, находящиеся в окружающих точку использования имён облатсях видимости, находятся без дополнительных указаний. Чтобы найти имена в других ветвях дерева пространств имён, необходимо явное указание информации о том, где искать эти имена, т.е. использование ***квалифицированных имён (qualified name)***. Квалифицированные имена — последовательность не квалифицированных имён, разделённых ***операцией разрешения области видимости (namespace resolution operator)***, которая также может с него начинаться. Если квалифицированное имя начинается с операции разрешения области видимости, его называют ***полностью квалифицированным (fully qualified)***, подобные имена по назначению аналогичны абсолютным именам в файловых системах — задают уникальные имена сущностей независимо от их использования. Поиск квалифицированных имён осуществляется следующим образом:

1. Если квалифицированное имя начинается с операции разрещения области видимости, начать с глобального пространства имён, иначе осуществить поиск первой компоненты как неквалифицированного имени. В таком поиске рассматриваются только именованные области видимости (для нас пока — пространства имён).
2. В текущей области видимости осуществить поиск описания следующей компоненты квалифицированного имени, что вновь должна дать имя области видимости, если это не последняя компонента.
3. Повторять предыдущий пункт для всех оставшихся компонент. Последний результат, который может быть любым описанием, есть результат всего поиска квалифицированного имени.

Пример:

```
1   // Глобальное пространство имён именуется просто ::.
2
3   // Пространство имён A в глобальном пространстве имён,
4   // его полное квалифицированное имя - ::A.
5   namespace A
```

```
6  {
7          // Определение f в A, полное квалифицированное имя - ::A::f.
8          int f()
9          {
10                 return 17;
11         }
12 }
13
14 // ::B
15 namespace B
16 {
17         // ::B::g
18         int g()
19         {
20                 // Поиск квалифицированного имени A::f:
21                 // - не полностью квалифицированное, ищем A
22                 //   как не квалифицированное до точки использования
23                 //   - в теле функции ::B::g,
24                 //   - в пространстве имён ::B,
25                 //   - в пространстве имён :: - найдено!
26                 // - ищем в текущей области видимости ::A имя f - есть!
27                 // Это была последняя компонента, так что ::A::f
28                 // есть результат всего поиска имён.
29                 return A::f();
30         }
31
32         // ::B::C
33         namespace C
34         {
35                 // Это определение ::B::C::g
36                 // скрывает описание из ::B.
37                 int g()
38                 {
39                         return 13;
40                 }
41
42                 int h()
43                 {
44                         // Скрытие в блоках, как безымянных областях
45                         // видимости было непреодолимо.
46                         // Для именованных облатсей видимости квалифицированные
47                         // имена позволяют именовать сущности, скрытые от
48                         // поиска неквалицированных имён.
49                         // Здесь первое имя именует скрывающее ::B::C::g,
50                         // а второе - скрытое ::B::g, т.к. явно указано
51                         // начало поиска, который дальше осуществляется
52                         // вглубь, а не наружу.
53                         return g()+B::g();
54                 }
55         }
56 }
```

Попытка дать определение пространству имён с тем же полностью квалифицированным именем, как у уже имеющегося не является нарушением правила одного определения: новое определение не заменяет, а расширяет имеющееся. Таким образом, потенциальная область видимости описания в области видимости пространства имён — объединение всех его определений, включая предшествующие и последующие.

```
1  namespace A
2  {
3      namespace B
4      {
5          int f()
6          {
7              return 0;
8          }
```

```
 9      }
10  }
11
12  // Повторное определение прострнства имён ::A
13  // расширяет предыдущее.
14  namespace A
15  {
16      namespace B
17      {
18          int g()
19          {
20                  // После поиска в теле g, осуществляется поиск
21                  // в объединении всех определений ::A::B до
22                  // точки использования, где и находится ::A::B::f.
23              return f();
24          }
25      }
26  }
27
28  // Несколько вложенных определений пространств имён можно
29  // для краткости сократить до одного, указав в качестве
30  // имени последовательность, разделённую операциями
31  // разрешения области видимости: вот ещё одно расширение ::A::B.
32  namespace A::B
33  {
34      int h()
35      {
36              // Находит имя A как имя пространства имён ::A
37              // неквалифицированным поиском и просматривает
38              // объединение трёх его фргаментов до точки использования
39              // в поисках B. Это тоже пространство имён.
40              // В объединении его трёх определений ищется и находится
41              // ::A::B::g.
42          return A::B::g();
43      }
44  }
```

Таким образом, как и предполагает процедурная парадигма, единица трансляции в наших первых программах — это последовательность определений функций, разбитых для структуризации на иерархию пространств имён.

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

Рассмотрим механизм реализации операции вызова функций и автоматического времени хранения. 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 x86-64 поддерживает работу с одним стеком на уровне своего набора инструкций. Чтобы отличить его от других стеков, которые может использовать програм-

ма 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 `RSP`, 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 rsp,8
    mov [rsp],value
```

а

```
pop location
```

эквивалентно

```
    mov location,[rsp]
    add rsp,8
```

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 `RIP`) 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 `RIP`.

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 часто используется ещё один регистр процессора `RBP`. 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   long f(long a,long b,long c,long d,
2          long e,long f,long g,long h,long i)
3   {
4       long t = a+b;
5       long t2 = c+d;
6       return t+t2+e+f+g+h+i;
7   }
8
9   int main()
10  {
11      f(1,2,3,4,5,6,7,8,9);
12  }
```

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

```
main:
    ; ...
    mov rdi,1
    mov rsi,2
    mov rdx,3
    mov rcx,4
    mov r8,5
    mov r9,6
    push 9
    push 8
    push 7
    sub rsp,8
    call f
    add rsp,32
    ; ...
```

Первые шесть аргументов записываются в регистры `RDI`, `RSI`, `RDX`, `RCX`, `R8` и `R9` соответственно. Остальные помещаются на стек в порядке справа налево. Ещё 8 не используемых байт на стеке резервирует следующая инструция вычитания этого значения из RSP.

После этого осуществляется вызов функции инструкцией `call`. Эта инструкция сначала записывает на стек текущее значение `RIP`, то есть адрес следующей за ней инструкции, а затем заменяет его своим операндом. 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 `main` 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`:

```
f:
    ; Пролог
    push rbp
    mov rbp,rsp
    sub rsp,16
    ; t = a+b;
    mov rax,rdi
    add rax,rsi
    mov [rbp-8],rax
    ; t2 = c+d;
    mov rax,rdx
    add rax,rcx
    mov [rbp-16],rax
    ; return t+t2+e+f+g+h+i;
    mov rax,[rbp-8]
    add rax,[rbp-16]
    add rax,r8
    add rax,r9
    add rax,[rbp+24]
    add rax,[rbp+32]
    add rax,[rbp+40]
    ; Эпилог
    mov rsp,rbp
    pop rbp
    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 `RBP`, 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 2 объекта `t` и `t2` по 8 байт каждый = 16 байт). Кадр стека функции `f` после исполнения её пролога показан на рис. 4.2.

⋮	
`i`	← `[rbp+40]`
`h`	← `[rbp+32]`
`g`	← `[rbp+24]`
(не используется)	← `[rbp+16]`
адрес возврата в `main`	← `[rbp+8]`
сохранённое значение `rbp` из `main`	← `[rbp]`
`t`	← `[rbp-8]`
`t2`	← `[rbp-16]`, `[rsp]`

Рис. 4.2: 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`. x86-64 использует гибридную схему передачи входных параметров, в который шесть первых передаются для скорости через регистры, а остальные — через стек.

Значения возвращаются из функции в регистре `RAX`. Эпилог проделывает обратные прологу операции: восстанавливает положение указателя стека из регистра `RBP`, после чего восстанавливает старое значение `RBP` из стека. Инструкция `ret` (RETurn) осуществляет снятие со стека значения и запись его в регистр `RIP`, тем самым выполнение 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++ для платформы x86-64 в большинстве UNIX-производных ОС согласно System V ABI **??**:

- Параметры передаются в регистрах `RDI`, `RSI`, `RDX`, `RCX`, `R8` и `R9`, а затем через стек, куда помещаются по порядку, начиная с последнего.
- Возвращаемое функцией значение передаётся вызывающей через регистр `EAX`.
- `RSP` играет роль указателя стека.
- `RBP` (опционально играющий роль указателя кадра стека), `RBX` и `R12-r15` есть ***сохраняемые (calee-saved)*** регистры. Если вызываемая функция их изменяет, она должна запомнить их старые значения и восстановить перед возвратом.
- Регистры `R10`, `R11`, а также те, что используются для передачи аргументов и возврата значения — ***временные (temporary, caller-saved)***. Они могут изменяться вызываемой функцией по её усмотрению.
- На момент выполнения команды `call` содержимое указателя стека должно быть кратно 16. Закрепление этого требования позволяет прологу удовлетворять аналогичные требования для объектов с автоматическим временем хранения, которые он создаёт. Эти требования ***выравнивания (alignment)*** соответствуют специфике работы контроллеров памяти, и будут нами рассмотрены, когда найдут своё отражение в C++. Это причина выделения не используемых 8 байт на стеке в нашем примере.
- ***Листовые (leaf)*** функции могут использовать ***красную зону (red zone)*** — область в 128 байт под `RSP` — без его изменения. Вообще, листовым функциям некому указывать на границу своего кадра стека, и хотелось бы сэкономить пару инструкций на изменение `RSP` в прологе и эпилоге, но ОС иногда вмешивается в устройство стека прикладной программы, поэтому в качестве компромисса подобное разрешено только с данным ограничением.
- Освобождение места на стеке, выделенного для хранения параметров функции, осуществляет вызывающая функция. Если возложить ответственность за это на вызываемую функцию, можно было бы устранить дублирование одной инструкции, но данное

соглашение о вызовах поддерживает функции с переменным числом параметров, и в этой ситуации размер параметров на стеке варьируется в каждой точке вызова.

Это не полные соглашения о вызовах, которые даже в рамках языка C++ 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 `RBP`, 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 `RBP` и `RSP` 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)***.

4.11.5. Функция main

Одна из функций программы имеет специальную роль функции, выполнение которой и составляет выполнение всей программы — ***функция, именуемая `main`***. Она должна быть расположена в глобальном пространстве имён. 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 единицы трансляции простейшей, ничего не делающей программы. Возвращаемое значение функции `main` является кодом возврата, передаваемым вызвавшей нас программе, как было показано ранее.

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.

> Из-за особого статуса функции `main`, вызов её из самой программы не допускается. Также по историческим причинам ей разрешено завершаться выходом из своего тела без выполнения оператора `return` — это определённое поведение, эквивалентное `return 0;`. Программа без определения функции `::main` не будет успешно откомпонована, поскольку её определение необходимо элементам стандартной библиотеки, осуществляющие начальные действия при запуске процесса, прежде чем передать управление этой функции и начать выполнение программы с формальной точки зрения.

4.12. Условный оператор if

Для записи алгоритмов, включающих условные переходы, нам потребуется соответствующие языковые средства. В соответствии со структурной парадигмой программирования, мы в первую очередь рассмотрим операторы, соответствующие двум оставшимся пока без рассмотрения конструкциям структурного программирования — ветвлениям и циклам. Начнём рассмотрение с ветвления, как конструкции, необходимой для корректности работы даже простейших программ, взаимодействующих с пользователем.

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 неявно преобразуемо к типу `bool`. Короткая форма оператора предписывает: «если значение выражения expression истинно, выполнить оператор statement». В данном случае элементом синтаксиса одного оператора являются другие произвольные операторы. Длинная форма предписывает: «если значение выражения истинно, выполнить statement-1, иначе выполнить statement-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  double abs(double x)
2  {
3      if(x<0)
4          x = -x;
5      return x;
6  }
```

Тело этой функции «взятие модуля вещественного числа» содержит два оператора: `if` и `return`, предписывающих следующие действия над полученным в качестве параметра значением в объекте `x`:

1. Если $x < 0$, заменить его значение противоположным.
2. Вернуть это значение из функции.

Оператор присваивания, осуществляющий саму смену знака, является частью оператора `if`, иначе говоря, подчинён ему, что при записи принято отображать смещением его в записи текста программы. Хотя сам оператор `if` соответствует ветвлению на две ветви, можно расположить другие операторы `if` в этих ветвях и тем самым реализовать альтернативу из любого числа вариантов. В таком случае важным является выбор порядка записи проверок в управляющих выражениях, чтобы избежать по возможности повторения их частей:

```
1  double f(double x)
2  {
3      if(x<-1.)
4          return -1.;
5      else
```

```
6           if(x<=1.)
7               return x;
8           else
9               return 1.;
10  }
```

В данном случае `f` — кусочно заданная функция из трёх интервалов $(-\infty;-1)$, $[-1;1]$ и $[1;\infty]$, и во втором контролирующем выражении не требуется дополнительной проверки `x>=-1.`, поскольку это гарантируется нахождением в отрицательной ветви предыдущего условия `x<-1.`. Этот пример также показывает, что в функции может быть несколько операторов `return`. С точки зрения тела самой функции, в ней всего один условный оператор, пусть и со сложной внутренней структурой.

Размещение составного оператора в ветвях условного позволяет связать с ними произвольное число операторов. Именно поэтому синтаксис самого условного оператора не включает возможность записи нескольких действий в своих ветвях:

```
1   double f(double a,double b)
2   {
3       // Если b левее a на числовой оси,
4       if(b<a){
5           // Обменять значения a и b местами.
6           // (Поскольку одновременно считать и записать
7           //  два значения нельзя, чтобы не потерять ни
8           //  одного из них, нужен третий временный
9           //  вспомогательный объект.)
10          double t;
11          t = a;
12          a = b;
13          b = t;
14      }
15      // Теперь гарантировано a>=b.
16      // ...
17  }
```

Исходя из синтаксиса языка C++, при наличии нескольких вложенных операторов `if`, ключевое слово `else` и оператор за ним соответствуют ближайшему из них, у которого отрицательной ветви ещё нет. В показанной ранее кусочной функции 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   int c;
4   // ...
5   if(a)
6       if(b)
7           c = 1;
8   else
9       c = 2;
```

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   int c;
4   // ...
5   if(a)
6       if(b)
```

```
7           c = 1;
8       else
9           c = 2;
```

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  int c;
4  // ...
5  if(a){
6      if(b)
7          c = 1;
8  }else
9      c = 2;
```

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.

Иногда в самом начале изучения языка складывается привязанность считать выражения с логическим результатом чем-то, что неразрывно связано с контролирующим выражением условного оператора. Это ведёт к появлению кода следующего вида:

```
1  // Не рациональный пример!
2  int a;
3  bool b;
4
5  // ... задание значения a ...
6
7  if(a>7)
8      b = true;
9  else
10     b = false;
```

Подобного рода конструкции корректны, но избыточны, и применять их не следует — есть гораздо более простое решение:

```
1  // Более простой вариант оператора в примере выше.
2  b = a>7;
```

Встречаются и несколько более сложные случаи, которые так или иначе можно записать с учётом наличия неявных преобразований между `bool` и другими арифметическими типами:

```
1  int a,b;
2  // ... задание значения a ...
3
4  // Вместо
5  if(a)
6      b = 1;
7  else
8      b = 0;
9
10 // следует использовать:
11 b = !a;
```

Пока не следует пытаться, например, заменить присваивание 5 или 7 в зависимости от условия на прибавление «5 плюс 2 умножить на результат логического выражения, соответствующего этому условию».

Подобные замены могут в некоторых случаях быть полезны, если соответствующие условия записаны в форме, которая может быть переведена на машинный код без использования ветвлений, что выгодно для многих современных процессоров. Грамотное использование этого приёма опирается на знание конкретных архитектур, поэтому применять его следует только при необходимости, т.к. соответствующий код абсолютно точно будет значительно сложнее для понимания, а даст ли заметный выигрыш в скорости — вопрос открытый.

4.12.1. Условная операция ?:

Альтернативой некоторым формам оператора `if` является операция, позволяющая прямо в выражении сделать выбор одного из двух значений по условию. ***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 как общий тип двух последних операндов, для арифметических типов — по правилам обычных арифметических преобразований.

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;
```

4.13. Изменение порядка выполнения команд в машинном коде

В архитектуре x86-64, как нам уже известно, регистр процессора `RIP` 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 `RIP` 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 значений битов регистра флагов `RFLAGS`. Регистр `RFLAGS` — специальный 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 x86-64 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 регистра по модулю 2 в степени его ширина в битах.
- `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 типом.
- `OF` (Overflow Flag) — выставляется в 1, если при сложении двух чисел с одинаковым старшим битом старший бит результата отличается от старшего бита операндов. 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 дополнительным 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 x86-64 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.

4.14. Директивы препроцессора. Директива include.

Деление программы на C++ на отдельно транслируемые части обеспечивают раздельно транслируемые файлы исходного текста. С точки зрения модульной парадигмы программирования, каждая единица трансляции состоит из средств, которые она предоставляет для использования другим единицам трансляции, и всех вспомогательных элементов, которые требуются для их функционирования, которые необходимо скрыть от окружающих. Описание правил взаимодействия с внешне доступной частью программы называются ***интерфейсом (interface)***, и включают в себя как конструкции уровня языка (описания в C++), так и сопутствующую документацию, где излагаются правила, не выразимые синтаксисом и семантикой языка напрямую. Все вспомогательные конструкции, или ***детали реализации (implementation details)***, в соответствии с этой парадигмой не описаны в интерфейсе и не должны быть доступны из других частей программы, чтобы избежать избыточных внутренних связей, нарушающих иерархию и структуру программы. Это необходимое ограничение для любых

нетривиальных программ, которые в своём развитии не могут себе позволить зависимостей «отовсюду куда угодно», поскольку это сделает практически невозможным любое изменение и развитие программы.

К сожалению, непосредственных средств поддержки модульной парадигмы язык C++ не имеет до сих пор, хотя работы в этом направлении ведутся. В отсутствии полноценной поддержки модульной парадигмы, для её эмуляции современный C++ опирается, к сожалению, на примитивные, с точки зрения уровня их функционирования, средства предварительной обработки. В процессе разбиения файла исходного текста на токены, препроцессор также выполняет имеющиеся в нём для него команды — ***директивы препроцессора (preprocessing directive)***. Это строки файла исходного текста, первый непробельный символ которых — решётка **#**. Содержимое директив препроцессора, хоть и состоит в некотором приближении из токенов, на основную фазу трансляции не поступает, то есть, эффективно, удаляется после обработки.

Если за решёткой ничего не следует, это так называемая пустая директива, которая ничего не делает, иначе далее следует имя директивы и её аргументы.

В нашей первой программе нам потребуется воспользоваться элементами стандартной библиотеки, поскольку именно на этом уровне, а не на уровне семантики самого языка, реализуются необходимые нам средства ввода/вывода. Чтобы воспользоваться сущностями из других единиц трансляции, в том числе, содержащихся в библиотеках, необходимо описать их в той единице трансляции, где они требуются. Из-за отсутствия понятия «модуль» на уровне семантики языка, такие описания включают в явном виде, возлагая на препроцессор обязанность по копированию этих описаний в каждую единицу трансляции, где они требуются.

Директива `include` предписывает препроцессору вставить на её место содержимое другого текста, имя которого задано её аргументом. Чаще всего эта директива применяется для включения не произвольных текстов на языке C++, а ***заголовочных файлов (header file)*** — файлов с описаниями интерфейсов единиц трансляции, защищённых от повторного включения. Нужная нам в данном случае форма этого аргумента — имя текста в угловых скобках. Такая форма предписывает поиск текста с указанным именем в последовательности мест, определяемых транслятором. Мы используем тот факт, что эта последовательность в правильно установленнном и настроенном компиляторе включает местоположение заголовочных файлов стандартной библиотеки языка C++.

Директивы `include` для всех заголовочных файлов, используемых единицей трансляции, размещают перед всем остальным её текстом:

```
1  // Включить описания из заголовочного файла iostream, который понадобится нам далее.
2  #include <iostream>
3
4  int main()
5  {
6          // Можно использовать имена, описанные в iostream.
7      return 0;
8  }
```

Также правильно настроенные инструментальные средства автоматически находят и включают в процесс компоновки программы стандартную библиотеку языка C++, чтобы использованные по включённым описаниям сущности имели свои определения. Начинающие программисты нередко использование директивы `include` называют «подключением библиотек» с соответствующими именами, что в корне неверно: это лишь включение описаний из соответствующих заголовочных файлов, библиотеки, включая стандартную, не участвуют в трансляции файлов исходного текста вовсе.

Всё содержимое стандартной библиотеки описано в пространстве имён `std`, что требует его явного указания в квалифицированных именах. Многие примеры опираются на пока не рассмотренную нами конструкцию, позволяющую опускать префикс `std::` для краткости. Такое использование имеет свои недостатки, которые мы рассмотрим далее.

4.15. Введение в форматированный ввод/вывод

Рассмотренные нами в данной главе возможности позволяют формулировать вычислительные алгоритмы из арифметических операций над заданными литералами 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.

Как было сказано ранее, основным средством взаимодействия с ОС являются файлы, и в наших первых программах мы будем работать с объектами языка C++, соответствующими стандартным файлам ввода, вывода и вывода ошибок. В стандартной библиотеке языка C++ работа с файлами уровня операционной системы происходит через объекты ***потоков ввода-вывода (input/output streams)***. Операции над ними отражают схожую идею: чтение и запись (ввод и вывод) последовательностей символов. В отличие от файлов, некоторые виды потоков ввода-вывода могут быть реализованы полностью внутри самой программы без обращения к операционной системе, или соответствовать другим её абстракциям, отличным от файлов. Мы будем использовать объекты, соответствующие стандартным файлам, поэтому в данном случае соответствие прямое. Также потоки ввода-вывода C++ позволяют выбирать единицу обмена информацией с потоком, которая в большинстве случаев, включая рассматриваемый нами, является байтом, представимым типом `char` в системе типов языка.

Также как стандартные файлы уже являются открытыми в момент запуска программы и доступны на протяжении всей её работы, объекты соответствующих поток в ввода-вывода существуют всё время выполнения программы, и их время хранения не привязано к какому-либо блоку. Определения этих объектов даны в стандартной библиотеке языка C++ и для их использования в программе достаточно иметь их описания.

Вернёмся к одной из программ, приведённых 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      if(!(std::cin >> a >> b)){
8          std::cerr << "Input error!\n";
9          return 1;
10     }
11     c = a*b;
12     std::cout << a << " * " << b << " = " << c << '\n';
13 }
```

Данный исходный текст начинается с директивы предварительной обработки, включающей в текст единицы трансляции заголовочный файл `iostream`, содержащий 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. Рассмотрим содержащиеся в нём операции ввода-вывода. Обратите внимание, что значительная часть из деталей поведения этих операций, которая будет изложена ниже, является следствием стандартных настроек объектов потоков ввода-вывода, и может быть изменена средствами, которые будут изучены позднее.

4.15.1. Форматированный вывод

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

Описание квалифицированного имени `std::cout` было включено в эту единицу трансляции директивой из её первой строки и является именем объекта, соответствующего стандартному потоку вывода. 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.

Рассматриваемые потоки ввода-вывода осуществляют обмен с файлами информацией в виде последовательности значений типа `char`. В языке C++ это тип имеет несколько смыслов, в том числе «байт» и «узкий символ». Файлы называют ***текстовыми (text)***, если они состоят из последовательности байтов, соответствующих кодам символов в некоторой кодировке. Обычно она соответствует набору символов среды выполнения в терминах языка C++ и некоторому выбранному способу их представления. Такие файлы могут быть прочитаны или созданы обычными текстовыми редакторами. Файлы, содержащие информацию в иных представлениях, называют ***бинарными (binary)***, и для работы с ними потребуется соответствующее ПО.

Потоки ввода вывода имеют два режима работы с файлами, соответствующие этой классификации: при работе с потоком ввода-вывода в двоичном режиме считываемая и записываемая информация самим потоком ввода-вывода не изменяется, за содержимое файла полностью отвечает программа. Такой файл может содержать информацию в наиболее удобном для обработки программой виде, например, таком же, как и в памяти: число 42 может быть записано как один байт, видимый как `2a` в шестнадцатеричном представлении.

В текстовом режиме то же значение будет записано в виде двух символов `'4'` и `'2'`, и, возможно, отделено пробельными символами от другой информации. С точки зрения языка C++ для разделения строк в текстовом файле используется управляющий символ «перевод строки», обозначаемый в литералах escape-последовательностью `\n`. В этом режиме поток ввода-вывода преобразует считываемую и записываемую информацию так, чтобы программа могла одинаково работать с текстовыми файлами в любых операционных системах, имеющих свои отличия в форматах текстовых файлов, которые формально относятся к зависящему от реализации поведению. Основным таким изменением в настоящее время является использование в Windows, в отличие от большинства других ОС, пары символов `"\r\n"` для представления границы между строками в текстовых файлах. С точки зрения программ на C++ при работе с текстовыми потоками вводится и выводится всегда ровно один символ «перевод строки» (`\n`), а символ «возврат каретки» (`\r`) добавляется и убирается при записи и чтении соответственно самой реализацией потока ввода вывода. В других ОС такого преобразования не требуется, т.к. представление текстовых файлов совпадает с правилами, принятыми в самом языке C++. Наиболее частое проявление этого различия — «склейка» всего текстового файла, подготовленного в Unix-производной ОС, в одну строку при открытии простым текстовым редактором в ОС Windows, который понимает только принятые в этой ОС разделители строк. Для конверсии между двумя форматами можно использовать программы `dos2unix` и `unix2dos`.

Стандартный режим работы потоков ввода-вывода — текстовый, и именно его мы будем использовать в начале изучения языка, поскольку текстовые файлы легко могут быть обработаны другими программами.

С терминалом также следует работать, как с текстовым файлом. Текст будет выводится начиная с текущего положения курсора слева направо, сдвигая сам курсор по ходу вывода. При достижении края экрана курсор перейдёт на начало следующей строки, того же можно добиться в произвольный момент времени выводом управляющего символа «перевод строки». Если курсор находился на последней строке, то при переходе на новую весь экран сдвигается на строку вверх, освобождая нижнюю строку, самая верхняя при этом теряется или, на современных эмуляторах терминала, попадает в его историю, доступную через полосу вертикальной прокрутки. Это, разумеется, не 100% возможностей вывода текста на терминал, но такой строго последовательный вывод текст поддерживается одинаково в любой ОС и соответствует работе с последовательным файлов на диске, если, например, стандартный файл вывода программы перенаправлен. Если стандартный файл вывода соответствует обычному файлу, информация записывается в него как текст, как было указано выше.

При работе с потоками ввода-вывода в текстом режиме, к ним применяют операции ***форматированного (formatted)*** ввода-вывода, обеспечивающие преобразование информации между их представлением в памяти, принятом в данной реализации языка C++, и последовательностью символов, представляющих это же значение в виде текста. Бинарная инфиксная операция `<<`, левым операндом которой является объект, именованный `std::cout`, осуществляет в качестве побочного эффекта форматированный вывод значения, заданного её правым операндом категории prvalue. В качестве правого операнда могут использоваться значения всех известных нам типов, которые будут выведены в виде, сходным с записью соответствующих литералов, например значение 123 типа `int` будет преобразовано и выведено как последовательность символов `1`, `2` и `3`. Исключением является тип `bool`, значения которого будут выведены как 0 или 1. Для символьных типов, в отличие от прочих целочисленных, выводится один символ, соответствующий числовому коду, хранящемуся в них. Отметим, что указанные соглашения есть лишь частный случай настроек потока, касающихся форматированного вывода, которые активны по умолчанию.

```
1  std::cout << 'x';    // Выводит символ "x".
2  std::cout << '\n';   // Выводит перевод строки.
3  std::cout << true;   // Выводит "1".
4  std::cout << 12;     // Выводит "12" (преобразовав значение типа int
5                       // в последовательность символов "1" и "2").
6  std::cout << 12.345; // Выводит "12.345" (преобразовав значение типа
7                       // double в соответствующую последовательность символов).
```

Результат вычисления этой операции — значение, вновь идентифицирующее объект `std::cout` (по прежнему lvalue), это свойство вместе с ассоциативностью операции `<<` слева направо может использоваться для вывода нескольких значений по цепочке, как показано на строке 12. Точнее:

```
1  std::cout << 12 << ' ' << 3.45 << 'n';
2  // эквивалентно
3  (((std::cout << 12) << ' ') << 3.45) << 'n';
4  // где каждая операция имеет побочный эффекты вывода,
5  // после чего возвращает левый операнд std::cout,
6  // для которого нам известно поведение при подстановке
7  // в левый операнд следующей операции.
```

В примере выше также встречаются 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. Экранирование буквального символа двойной кавычки внутри строкового литерала escape-последовательностью производится аналогично одиночным кавычкам в символьном. Не путайте символьные литералы со строковыми, выражения `'a'` и `"a"` имеют принципиально разные типы и значения! Для нас в данный момент является принципиальным возможность неявного преобразования значения строкового литерала к типу, который может использоваться в качестве правого операнда для операции `<<`, когда левым являются потоки вывода. Такое использование имеет ожидаемый побочный эффект — вывод в указанный поток всех символов, записанных в литерале. В случае, когда требуется вывести один символ, следует предпочесть символьный литерал строковому, т.к. такая операция эффективнее. Иначе, напротив, эффективнее строковый литерал, который также позволяет избежать записи множество операций вывода по одному символу.

```
1  std::cout << 's' << 't' << 'r' << '\n' ; // Выводит "str" и перевод строки.
2  std::cout << "str\n";                    // Тот же эффект, но короче.
```

Выводимые данные часто формируются посимвольно, но обращаться к операционной системе для записи по одному символу за раз слишком накладно, особенно учитывая частое наличие множества промежуточных уровней по пути от программы к реальному устройству вывода. Многие из этих уровней, включая реализацию стандартной библиотеки языка C++ в лице потоков вывода осуществляют ***буферизацию (buffering)*** — накопление выводимых данных и фактический их вывод порциями. При работе с текстовыми файлами вывод по умолчанию ***буферизован построчно (line-buffered)***: вывод накопленных символов осуществляется не только по достижении фиксированного размера буфера, а также при каждом выводе символа перевода строки. Остатки во всех буферах также выводятся при закрытии соответствующих файлов. С практической точки зрения это означает, что не стоит удивляться, если вы остановили отладчиком программу, но её фактический вывод не соответствует месту останова — он может быть буферизован.

На строке 8 в аналогичной конструкции используется объект, именованный `std::cerr`, — это имя объекта стандартного потока вывода. По операциям, которые могут быть к нему применены, он аналогичен стандартному потоку вывода, использовать его следует для вывода сообщений об ошибках, что и показано в этом примере. Поскольку сообщения об ошибках должны по смыслу быть показаны незамедлительно, этот поток по умолчанию буферизацию не использует.

4.15.2. Форматированный ввод

Парным к объектам стандартных потоков вывода является объект `std::cin`, соответствующий стандартному потоку ввода, позволяющий осуществить ввод данных в программу с терминала или из файла. Для этого он используется в качестве левого операнда бинарной инфиксной операции форматированного ввода `>>`, правый операнд которой должен быть леводопустимым выражением, идентифицирующим объект, в который будет записано считанное значение после его преобразования 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 при стандартных настройках поток ввода практически совпадает с синтаксисом соответствующих литералов за следующими исключениями:

- Префиксы систем счисления, суффиксы типов и одиночные кавычки-разделители разрядов не распознаются, ввод осуществляется всегда в десятичной системе.
- Ввод, соответствующий целочисленному литералу, принимается и при вводе значений с плавающей точкой.
- Перед вводимыми числами допускаются символы `+` и `-` для задания знака значения.
- При вводе значений символьных типов считывается один символ в виде соответствующего ему кода.

Ввод может завершиться неудачно по разным причинам, начиная от несоответствия пользовательского ввода требуемому формату, достижения конца файла, и заканчивая аппаратными ошибками при доступе к файлу. Для отслеживания этих ситуаций объект потока ввода хранит в себе флаги, соответствующие различным видам этих ошибок.

Сам процесс форматированного ввода состоит из четырёх этапов:

1. Проверяются флаги ошибок. Если одна из предыдущих операций ввода была неудачна по какой-либо причине, дальнейших действий не предпринимается.
2. Вводятся и пропускаются все ***пробельные символы (whitespace)***. Пробельные символы — обычные и управляющие символы текста, относящиеся к его содержимому, не имеющие графического представления. Чаще всего в их качестве используются пробел, табуляция и перевод строки.
3. Вводятся и накапливаются символы, пока накопленная строка соответствует формату представления требуемого типа данных, как описано выше. Первый символ, нарушающий этот формат возвращается и остаётся не считанным — он доступен для следующих операций ввода.
4. Просматривается сохранённая строка. Если она не пуста, то по построению она содержит допустимое по форме представление требуемого типа данных. В таком случае осуществляется преобразование строки в соответствующее значение и, как побочный эффект, запись его в объект идентифицированный правым операндом операции `>>`. Если значение корректно по форме, но не попадает в диапазон значений требуемого типа, записывается минимальное или максимальное значение, которое представимо, и выставляется один из флагов ошибки. Если строка пуста, то это означает, что первый встреченный непробельный символ не является корректным представлением значения требуемого типа, так что ввод также неудачен: в объект, идентифицированный правым операндом, записывается значение 0, приведённое к его типу, и выставляется флаг ошибки.

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. Например, при запросе рассматриваемой нами программой двух чисел типа `double` ввод строки `1 23` и нажатие `Enter` на терминале обрабатывается следующим образом:

1. Первая операция чтения:
   (a) Пропуск пробельных символов не считывает ничего (`1` — не пробельный).
   (b) В качестве представления значения считывается символ `1`, следующий символ «пробел» считывается, но единица с пробелом месте не является представлением величины типа `double`, в которую пробелы входить не могут. В качестве представления остаётся один символ единицы, пробел возвращается для последующего считывания.
   (c) Считанная последовательность не пуста, строка `1` преобразуется в значение 1 типа `double` и записывается в объект `a` (при форматированном вводе целочисленные по формату литералов C++ значения допускаются).
2. Вторая операция чтения:
   (a) Пропуск пробельных символов пропускает пробел, оставленный прошлой операцией не считанным.
   (b) В качестве представления значения считывается символы `23`, а символ перевода строки остаётся не считанным, т.к. не может входить в представление `double`.
   (c) Строка `23` преобразуется в соответствующее значение, которое заносится в объект `b`.

3. Символ перевода строки остаётся не считанным, и доступен последующим операциям ввода. Если они будут проходить по тому же алгоритму, он будет считан и отброшен на первом шаге.

С тем же успехом можно было вводить значения по одному на строку, любые пробельные символы в данном алгоритме будут являться и ограничителями, обозначающими конец вводимых значений, и пропускаться в начале ввода следующих.

Поток ввода, как и потоки вывода, обладает буфером. В случае с вводом с терминала также будет наблюдаться построчная буферизация — 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. В данном случае это результат буферизации не в реализации потоков ввода-вывода, а в эмуляторе терминала. Это обычно не является проблемой для программ с диалоговым интерфейсом, и в то же время позволяет использовать клавишу `Backspace` для исправления ошибок ввода, прежде чем они будут введены в программу.

Несмотря на описанные выше правила буферизации, в показанном примере используется запрос к вводу чисел, не содержащий символ перевода строки. Исходя только из этих правил гарантии того, что он отобразиться перед началом чтения из потока стандартного ввода нет. Стандарт языка C++ относит к видимому поведению программы, обязательного к выполнению реализациям следующее: «динамика интерактивных операций ввода-вывода должна быть такова, что тексты запросов к вводу выводятся перед началом соответствующих операций ввода». Стандарт относит к «интерактивным устройствам» то, что определяется самой реализацией, в данном случае это правило касается устройств ввода-вывода, связанных с (виртуальными) терминалом. Для обеспечения этой гарантии на уровне реализации потоков ввода-вывода стандартной библиотекой языка C++, вводиться понятие ***связанного (tied)*** потока: если у потока имеется связанный, то перед каждой операцией ввода-вывода на данном потоке, данные из буфера связанного потока принудительно записываются. Для `std::cin` и std::cerr связанным по умолчанию является `std::cout`, что и обеспечивает то, что приглашения к вводу выводятся перед самим вводом. Это позволяет размещать приглашения к вводу и сам пользовательский ввод на одной строке, чего мы в данном случае и добивались.

Особо отметим, что не всякий ввод является удачным. Например, если на запрос числа пользователь введёт букву как первый непробельный символ, операция ввода завершится неудачно. Последующие операции ввода на потоке, который находится в ошибочном состоянии предприниматься не будут. Даже если это было бы не так, этот символ остался не считанным, и препятствовал бы успеху последующих операций ввода. Хотя в некоторых программах может иметь смысл предоставлять пользователю дополнительные попытки ввода в интерактивном режиме, соответствующие средства потоков ввода-вывода, позволяющие корректно обрабатывать подобные ситуации, нами пока изучены быть не могут, т.к. опираются на сложные и пока не известные нам языковые конструкции. По этой причине автор настоятельно рекомендует до изучения этого вопроса считать любые ошибки ввода фатальными, после чего не предпринимать дальнейших попыток ввода. Во многих программах после возникновения такой ситуации не останется ничего другого, кроме как сообщить об ошибке ввода и завершить выполнение. Хотя это не самый дружелюбный к интерактивному пользователю сценарий, его придерживаются многие программы, поскольку он часто единственно корректен, если принимать возможность работы программы с не интерактивными источниками данных в пакетном режиме.

Главной причиной, по которой критически важно отслеживать подобные ошибки ввода — побочного эффекта записи в объект не происходит, если ввод осуществляется по цепочке и некоторые операции чтения не меняли содержимого своих объектов из-за неудачи предыдущих вводов. То же относится даже к одиночным вводам, если вы используете стандартную библиотеку стандарта C++03, т.к. зануление объектов при ошибках чтения гарантируется только начиная с C++11. Вероятно, соответствующий объект ещё не получал значения, т.к. это и предполагалось сделать в этой операции, и, следовательно, первое его чтение в программе далее станет неопределённым поведением.

Таким образом, проверять поток ввода на случившиеся ошибки необходимо после каждой операции чтения. Откладывать проверку на конец последовательности чтений, включая записанные по цепочке, можно только в случае, если между чтениями не производится попыток использования ранее считанных значений.

Как было указано ранее, объект потока ввода хранит в себе информацию о том, были ли ошибки ввода с момента запуска программы. Эту информацию можно извлечь из него

за счёт того, что он допускает ***контекстное преобразование (contextual conversion)*** к типу `bool`. Результат преобразования — истина, пока ни одной ошибки не происходило, после первой он обращается в ложь. Таким образом, хорошей привычкой можно назвать непосредственную проверку результата операции ввода или их цепочки, путём помещения их в контролирующее выражение условного оператора, т.к. результат таких выражений и есть требуемый к проверке объект `std::cin`. Контекстные преобразования к указанному типу являются подвидом неявных преобразований, допускаемых только в случае, если в точке преобразования явно требуется указанный тип. В данном случае это справедливо для контекстов, требующих именно `bool`, таких как контролирующее выражение `if` или операнд логической операции, но не в других случаях, например, недопустимо `std::cin+3` за счёт преобразования «тип `std::cin`» — `bool` — `int` (обычные арифметические преобразования после преобразования значения объекта стандартного потока в `bool`).

В рассматриваемом нами примере именно такая конструкция (с отрицанием преобразованного значения `std::cin`) используется в строке 7 в контролирующем выражении `if` — оно истинно в случае ошибки ввода (отрицания успеха). Строки 8–9 выводят сообщение об ошибке в стандартный поток вывода ошибок и завершают выполнение программы с кодом возврата 1.

Смысл остальных строк программы теперь должен быть приблизительно ясен: строка 6 определяет 3 объекта типа `double`, строка 11 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 `<<`.

К сожалению, обычное достижение конца файла (обычного, или нажатие соответствующей комбинации на терминале), нам пока не отличить от ошибки ввода по требуемому формату, например.

4.16. Циклы

Для записи произвольных алгоритмов осталось рассмотреть последнюю конструкцию структурного программирования — циклы.

4.16.1. 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  double x;
2  while(std::cin >> x)
3      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 по формату 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.

Когда среди значений в типе, в котором осуществляется ввод, есть недопустимые по смыслу программы, потребуется дополнительная проверка после успешности ввода на допустимость введённого значения. В таком случае для завершения ввода можно использовать и значение, корректное по формату, но неподходящее по смыслу. Однако, поскольку проверка на успешность ввода обязательна и присутствует всегда, автор не рекомендует использовать некорректные значения для завершения работы подобных программ — это неестественное решение при вводе из обычного файла, а не клавиатуры, и даже с клавиатуры обозначить конец ввода специально предназначенной для этого комбинацией клавиш труда не составляет.

> Чтобы упростить анализ циклов оптимизатором, в C++ требуется, чтобы любой цикл выполнял действие, включающее видимое поведение программы, взаимодействие с другими потоками выполнения или завершение за конечное число шагов. Таким образом, пустые вечные циклы вроде `while(true);` имеют неопределённое поведение.

4.16.2. Операция запятая

> Приведённый выше пример цикла не выводил приглашения к вводу каждого нового значения. Попытка сделать это без дублирования кода, вероятно, закончится неудачно. В данном случае приглашение к вводу логически является частью ввода, который выражается в побочном эффекте контролирующего выражения.
>
> Аналогично блоку, представляющему несколько операторов в виде одного, операцию «запятая» можно использовать для группировки нескольких независимых выражений в одно формальное. Это бинарная инфиксная операция, которая вычисляет детерминированно сначала левый операнд, отбрасывает результат, вычисляет правый и объявляет его результатом. С её использованием можно добавить в пример выше приглашение к вводу каждой величины без дублирования кода:

```
1  double x;
2  while(std::cout << "Input a number: ",
3        std::cin >> x)
4      std::cout << x*x << '\n';
```

> Не следует путать операцию «запятая» с другим использованием запятой как пунктуатора.

4.16.3. 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:

```
1  #include <iostream>
2
3  int main()
4  {
5      double sum,term,x;
6      sum = x = 0.;
7      // Для накопления суммы потребуется никак не меньше
8      // одного члена ряда, проверка в конце.
9      do{
10         ++x;
11         term = 6./(x*x);
12         sum += term;
13     }while(sum/term<1e10);
14     std::cout << "pi^2 = " << sum << " (" << it << " iterations used)\n";
15 }
```

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

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

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.

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

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

```
int i;
for(i=1;i<=100;++i)
        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. 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`:

```
int i,j;

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

// Степени числа 2 до 1024
for(i=1;i<=1024;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,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.
// Этот вариант предпочтительнее, т.к. число итераций - 10
// в нём записано явно. Начальные/конечные значения в виде 0
// вместо 1 часто математически и архитектурно удобнее.
```

```
11  // Удобство значения 0 ещё не раз встретится в C++.
12  for(i=0;i<10;++i)
13          std::cout << i << '\n';
14
15  // [0;10] - другой интервал,
16  // в который также входит число 10:
17  for(i=0;i<10;++i)
18          std::cout << i << '\n';
19
20  // От 10 до 0:
21  for(i=10;i>=0;--i)
22          std::cout << i << '\n';
23
24  // Особое внимание при счёте вниз и
25  // беззнаковых типах - следующий цикл
26  // вечный, потому что unsigned всегда >=0 !
27  unsigned j;
28  for(j=10;j>=0;--j)
29          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.

4.17. 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.

4.17.1. 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.

```
1   #include <iostream>
2
3   int main()
4   {
5           // Запросим у пользователя, сколько
6           // он будет вводить чисел.
7           unsigned n;
8           std::cout << "How many numbers? ";
9           if(!(std::cin >> n)){
10                  std::cerr << "Input failed or invalid count.\n";
11                  return 1;
12          }
13          // Посчитаем их сумму.
14          double sum = 0;
15          while(n){
16                  std::cout << "Input a number: ";
17                  double x;
18                  if(!(std::cin >> x))
19                          // Пользователь не ввёл очередное число, выходим из цикла досрочно
20                          break;
21                  sum += x;
22                  --n;
23          }
```

```
24          // Выведем сумму и сколько ещё чисел пользователь обещал ввести, но не ввёл.
25          std::cout << "Sum of numbers: " << sum << "\n"
26                    "You failed to input " << n << " numbers.\n";
27  }
```

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:

```
1   // !!! Умышленно плохой пример !!!
2
3   int i = 0;
4   while(i<10){
5           // ...
6           if(condition){
7                   // Условие истинно,
8                   // выполняем требуемые действия.
9           }else{
10                  // Условие ложно, цикл нужно
11                  // завершить заранее.
12                  // Присвоим i значение,
13                  // которое к этому приведёт.
14                  i = 10;
15          }
16  }
17  // А что делать, если значение i, на котором
18  // цикл прерван досрочно потребуется далее?
```

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

```
1   int i = 0;
2   while(i<10){
3           // ...
4           if(!condition)
5                   // Условие ложно, цикл нужно
6                   // завершить заранее.
7                   break;
8           // Условие истинно,
9           // выполняем требуемые действия.
10          // Не нужно перестраивать структуру программы, чтобы действия,
11          // выполняемые после проверки дополнительного условия продолжения
12          // цикла шли в блоке, зависящем от него.
13          // ...
14  }
15  // 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:

```
1   unsigned i;
2   for(i=10;;--i){
3           std::cout << i << '\n';
4           // Условие завершения цикла вынесено из контролирующего в само тело
5           // цикла и проверяется до декремента счётчика.
6           if(!i)
7           break;
8   }
```

4.17.2. Оператор continue

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

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

```
3  int main()
4  {
5          double sum = 0,x;
6          std::cout << "Input positive numbers:\n";
7          while(std::cin >> x){
8                  if(x<=0){
9                          std::cerr << x << " is non-positive, try again: ";
10                         if(!(std::cin>>x)||x<=0){
11                                 std::cerr << "Failed again, try with next index.\n";
12                                 continue;
13                         }
14                 }
15                 sum += x;
16                 std::cout << "Sum = " << sum << '\n';
17         }
18 }
```

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
3.1↵
Sum = 5.1
0↵
0 is non-positive, try again: 4↵
Sum = 9.1
-1↵
-1 is non-positive, try again: -5↵
Failed again, try with next position.
6.5↵
Sum = 15.6
x↵
```

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

```
1  while(loop_condition){
2          // ...
3          if(condition1){
4                  // ...
5                  if(condition2){
6                          // ...
7                          if(condition3){
8                                  // ...
9                          }
10                 }
11         }
12 }
```

можно записать

```
1  while(loop_condition){
2          // ...
3          if(!condition1)
4          continue;
5          // ...
6          if(!condition2)
7          continue;
8          // ...
9          if(!condition3)
10         continue;
11         // ...
12 }
```

4.17.3. 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:

```
while(condition1){
        // ...
        while(condition2){
                // ...
                while(condition3){
                        // ...
                        if(condition4){
                                // Здесь обнаружилось,
                                // что нужно выйти из
                                // всех трёх циклов -->
                        }
                        // ...
                }
                // ...
        }
        // ...
}
// <-- т.е. вот сюда.
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. Помеченный оператор предписывает выполнить содержащийся в нём, а также является определением метки с областью видимости «функция», поэтому все метки в пределах одной функции должны иметь уникальные имена. Оператор `goto` состоит из ключевого слова `goto`, имени метки, к которой осуществляется переход, и точки с запятой. Поиск имени метки в операторе `goto` осуществляется по всей функции как до, так и после точки использования, чтобы можно было прыгать вперёд по тексту программы.

С помощью этого оператора вышеприведённая задача легко решается:

```
while(condition1){
        // ...
        while(condition2){
                // ...
                while(condition3){
                        // ...
                        if(condition4){
                                // Здесь обнаружилось,
                                // что нужно выйти из
                                // всех трёх циклов -
                                // поможет goto:
                                goto end_of_loops;
                        }
                        // ...
                }
                // ...
        }
        // ...
```

```
}
end_of_loops:
std::cout << "Loops done\n";
// ...
```

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

```
#include <iostream>

int main()
{
    // Ввести целое число больше единицы.
    std::cout << "Input a number >1: ";
        unsigned a;
        if(!(std::cin>>a)||a<2){
                std::cerr << "Invalid input.\n";
                return 1;
        }
        // Попробовать найти такие числа x и y,
        // из [2;9], что введённое число делится
        // и на (x+y-1), и на (x+1)*(y-1).
        // Разные промежуточные ситуации отразить
        // знаками в таблице символов, где
        // значения x и y являются строками и
        // столбцами соответственно.
        std::cout << "x/y:23456789\n";
        unsigned x,y;
        for(x=2;x<=9;++x){
                std::cout << x << "  :";
                for(y=2;y<=9;++y){
                        // Проверить требования
                        bool cond1,cond2;
                        cond1 = a%(x+y-1)==0;
                        cond2 = a%((x+1)*(y-1))==0;
                        if(cond1)
                                if(cond2){
                                        // Выполнены оба.
                                        std::cout << '3';
                                        // Выйти из обоих циклов.
                                        goto done;
                                }else
                                        // Только cond1.
                                std::cout << '1';
                        else
                                if(cond2)
                                        // Только cond2.
                                        std::cout << '2';
                                else
                                        // Ни то, ни другое.
                                        std::cout << ' ';
                }
                std::cout << '\n';
        }
done:
        if(x==10&&y==10)
                // Циклы дошли до конца, нужное
                // значение не найдено.
                std::cout << "Not found.\n";
        else
                // Нашли. Выведем конец строки
                // для последней неполной
                // и найденные значения.
                std::cout << "\nFound: x=" << x << ", y=" << y << '\n';
}
```

Пример сеанса работы:

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

К счастью, большинство компиляторов выдают в таких случаях предупреждения.

Операторы перехода не отменяют правил автоматического времени хранения: преждевременный выход из блока оператором перехода до достижения его конца так же уничтожает автоматические объекты и освобождает занимаемую ими память. При прыжке оператором `goto` назад в пределах одного блока через определения объектов они также уничтожаются. Прыгать на метку вперёд через определение объекта с нетривиальной инициализацией запрещено, т.к. это противоречит требованию инициализации объекта, данному в этом определении.

Итак, наличие в языке операторов перехода никак не отменяет его поддержку структурной парадигмы, а, напротив, предоставляет больше выразительных возможностей программисту для реализации собственных намерений. Также следует отметить, что неопытность начинающих программистов часто приводит к избыточному использованию «флагов» — булевских по типу или просто смыслу объектов, которые добавляются в условия ветвлений и циклов, и модифицируются по необходимости с целью придать имеющимся в языке конструкциям дополнительную функциональность. В большинстве случаев подобные реализации алгоритмов можно переписать в форме без флагов, используя другие явные конструкции передачи управления. В императивном языке следует использовать явные операторы для описания логики работы программы, прибегая к флагам только в крайнем случае, опять же, с точки зрения читаемости. Использование флагов усложняет чтение программы и увеличивает количество её возможных состояний, затрудняя отладку.

4.18. Определения в операторах

Для минимизации видимости и времени хранения объектов, многие операторы позволяют использовать в них вместо выражений определения, чтобы избежать необходимости окружать оператор явным блоком. В отличие от некоторых других языков, C++ позволяет произвольно смешивать операторы и определения в теле функции, что вместе с детерминированным контролем выделения и освобождения памяти позволяет в точности контролировать время хранения объектов. Данная возможность позволяет минимизировать проблемы скрытия имён, а после введения объектов с нетривиальным созданием и уничтожением позволит точнее передавать семантику их использования.

Определением можно заменить контролирующие выражения операторов `if` и `while`, если они содержат только один описатель, точка с запятой в конце такого определения не ставится. Контролирующее услвоие в таком случае — начальное значение объекта. Если требуется нечто иное, его можно записать через точку с запятой после определения. Для оператора `if` роль блока, с которым связана видимость и время хранения такого объекта, выполняет сам оператор — объект доступен и в положительной и отрицательной ветвях, но не после выполнения всего оператора. Для оператора `while` объект будет связан с итерацией и пересоздаваться при каждом новом вычислении контролирующего выражения. У оператора `for` таким образом заменяется первое выражение, оно может иметь любое количество описателей, и соответствующие объекты хранятся всё время выполнения цикла:

```
1  // Блок ниже
2  {
3      unsigned n = get_n();
4      if(n){
5          // ...
6      }
7  }
8
9  // может быть переписан в виде:
10 if(unsigned n = get_n()){
11     // ...
12 }
13
14 // Произвольное контролирующее выражение может быть записано после
15 // точки с запятой:
16
17 if(unsigned n = get_n();n%3==1){
18     // Только если n, полученное вызовом функции,
19     // даёт 1 в остатке от деления на 3, сделать
20     // что-то с этим значением.
21 }
22 // После этого это значения больше не нужно.
23
24 // Чаще всего определение содержится в цикле for,
25 // теперь само создание счётчика тоже в его заголовке:
26 for(int i=0;i<100;++i)
27     std::cout << i << '\n';
28
29 // Это позволяет использовать то же имя для счётчиков
30 // циклов в той же области видимости и определять их
31 // однообразно, независимо от порядка...
32 for(int i=0;i<200;++i)
33     std::cout << i*2 << '\n';
34 // ... хотя это не подходит, когда после цикла нужно
35 // знать значение счётчика, на котором он остановился -
36 // он больше не существует.
37
38 // Работает также для while.
39 // Пока не принципиально, что объект уничтожается и
40 // создаётся заново на каждой итерации, т.к. это тривиально
41 // и значение всё равно задаётся заново каждый раз.
42 while(double f;
43       std::cout << "Input a number: ",
44       std::cin >> f)
```

```
45      std::cout << f << "^2 = " << f*f << '\n';
```

По очевидным причинам заголовок цикла `do` определений содержать не может.

4.19. Тип void

Формально, функции в C++ всегда имеют ровно одно возвращаемое значение, однако имеются случаи, когда оно не требуется. Это функции, полезное действие которых заключается в побочных эффектах. Придумывать фиктивные возвращаемые значения таким функциям, разумеется, не следует.

Ключевое слово `void` является именем фундаментального типа, имеющего пустое множество значений. Несмотря на это, выражения такого типа в языке существуют, и обозначают отсутствие осмысленного результата. Любое значение может быть приведено к типу `void`, в том числе явно, чтобы показать, что оно не требуется. Никаких других операций над значениями этого типа нет.

Этот тип чаще всего используется в качестве возвращаемого значения функций, которым по смыслу возвращать нечего:

```
1  // Вывести линию из n символов c.
2  void line(int n,char c)
3  {
4      for(int i=0;i<n;++i)
5          std::cout << c;
6  }
```

Показанная функция полезна своим побочным эффектом, а возвращать ей нечего. Возврат из функций, возвращающих `void`, осуществляется одним из трёх способов.

1. По достижению конца их тела, как в примере выше. Функции, возвращающие `void` не обязаны завершаться вызовом `return`.
2. Из любого места такой функции можно сразу выйти оператором `return` в специальной форме без выражения:

```
1  void f(int i)
2  {
3    // Если i==0, сразу выйти.
4    if(i==0)
5      return;
6    // ...
7  }
```

3. При наличии выражения типа `void`, можно воспользоваться стандартной формой `return` с выражением. Такой вариант используется нечасто. Пока только результаты операций вызова функций с возвращаемым значением типа `void` имеют такой тип, так что учитывая функцию `line` из примера выше, можно записать:

```
1  void f()
2  {
3      while(double x;std::cin>>x){
4          // Если очередное значение отрицательное,
5          // вывести отделяющую линию и выйти из функции.
6          if(x<0)
7              return line(80,'=');
8          // Этот пример несколько экзотичен, большинство
9          // программистов напишет:
10         if(x<0){
11             line(80,'=');
12             return;
13         }
14         // ...
15     }
16 }
```

Подобные функции чаще всего вызываются в операторе-выражении, которому не составит труда отбросить и так бессмысленное значение.

4.20. Перегрузка функций

Многие задачи с одинаковым итоговым результатом могут иметь несколько вариантов решения в зависимости от наборов входных данных. Например, геометрическая задача решения треугольника может быть поставлена по трём сторонам, стороне и двум углам, или двум углам и стороне, а в конечном итоге выдаёт все шесть характеристик треугольника.

Функции в C++ являются поименованными частями алгоритма и также допускают наличие нескольких вариантов своего определения. Свойство элемента программы вести себя по разному в зависимости от контекста использования называют ***называют (polymorphism)***. Версии определения функции называют ***перегрузками (overload)***, и различают по набору типов входных параметров. Поскольку C++ имеет статическую систему типов, такой вид полиморфизма, когда выбор конкретного поведения осуществляется на этапе трансляции, также называется статическим (в противоположность динамическому).

Таким образом, для функций допускается несколько несвязанных определений с одним именем в одной области видимости, образующие ***множество перегрузок (overload set)***. После нахождения всего этого множества алгоритмом поиска имён требуется произвести ***разрешение перегрузок (overload resolution)*** — выбор из них одного конкретного требуемого определения в контексте использования имени функции. Выбор осуществляется по требуемым типам аргументов, которые в известном нам случае задаются по составу аргументов в операции вызова функции. Разрешение перегрузок выполняется следующим образом:

1. Из множества перегрузок оставляют только ***годные (viable)*** функции — совпадающие по числу параметров с числом аргументов там, что из каждого типа аргумента существует неявное преобразование в тип соответстующего параметра, если эти типы разные.
2. Если годных функций нет, разрешение перегрузок неудачно.
3. Если годная функция ровно одна, она и есть результат разрешения перегрузок.
4. Иначе годных функций несколько. Требуется дополнительный шаг алгоритма —- выбор ***наилучшей годной (best viable)*** функции:

   (a) Чтобы быть наилучшей, функция должна быть попарно лучше всех остальных. Если такой нет, алгоритм разрешения перегрузок неудачен. По построению алгоритма, нескольких функций с таким свойством быть не может.

   (b) Чтобы из двух функций одна считалась лучше другой, она должна быть хотя бы по одному параметру лучше, а по остальным — не хуже другой.

   (c) Таким образом, отношения сводятся к выбору для конкретного по счёту аргумента, в какой из двух соответствующих параметров обоих функций его лучше приводить. Все имеющиеся в языке преобразования классифицируются в один из трёх рангов:

   - ***Точное совпадение (exact match)*** — преобразование не требуется, типы идентичны, или только преобразование леводопустимого выражения.
   - ***Повышения (promotions)*** — integral/floating promotions.
   - ***Преобразования (conversions)*** — все прочие преобразования между арифметическими типами.

   Если два рассматриваемых преобразования находятся в разных рангах, то лучшим считается точное совпадение, затем повышение, а преобразование — хуже всех остальных, иначе по данному параметру перегрузки не сравнимы.

Изложенный здесь алгоритм — лишь часть реального, соответствующего известным нам преобразованиям и типам. В любом случае, его смысл — выбрать наиболее близкую, с минимальным количеством неявных преобразований, которые возможны, перегрузку.

Рассмотрим перегрузку функций на примере перегрузки функции `std::sin` из стандартной библиотеки. Заголовочный файл `cmath` содержит следующие описания:

```
1  namespace std
2  {
3      float sin(float x);
4      double sin(double x);
5      long double sin(long double x);
6      double sin(Integral x); // см. ниже
7  }
```

Видно, что у функции взятия синуса (в радианах) имеются перегрузки для всех типов с плавающей точкой, при этом результат и вычисления производятся в том же типе, что позволяет обеспечить в точности требуемую производительность вычислений. Последняя фиктивная перегрузка соответствует явно не записанному множеству перегрузок для всех целых типов в языке (или эквивалентной по семантике конструкции). Для вызова `sin(0.5)` разрешение перегрузок выполняется следующим образом:

1. Годны все функции, т.к. один аргумент в точке вызова, и все перегрузки с одним параметром.
2. Годных перегрузок больше одной, требуется разрешение. Параметр всего один, поэтому он и решает отношения между функциями в целом.
3. Для перегрузки с типом параметра `double` ранг преобразования — точное соответствие, т.к. у аргумента тот же тип и преобразования не требуется. Для остальных перегрузок это floating или integral-floating conversion, ранг conversion, что хуже. Побеждает вторая перегрузка.

Наличие перегрузок, заданных последней строкой, позволяет вызывать синус от целочисленных операндов без явного приведения к одному из типов с плавающей точкой — без него из перегрузок нельзя было бы выбрать лучшую. (Синус от целого числа радиан — странный случай, но подобная схема применяется в стандартной библиотеки для всех библиотечных функций, где полезна, а не только для тригонометрии.)

> В языке C отсутствуют как пространства имён (в смысле структурирования области видимости единицы трансляции), так и перегрузки. Вместо перегрузок для математических функций в стандартной библиотеке C используются разные имена функций с суффикасми по той же схеме, что и у литералов с плавающей точкой. Использование математических функций без квалификации `std::` будет находить в глобальном пространстве имён не перегруженные версии только для `double`, что, внешне работая, будет искажать точность или негативно влиять на производительность.
>
> Больше всего в этом плане пострадала функция `std::abs` — поиск абсолютной величины числа. Её разновидности для целых типов и типов с плавающей точкой исторически описывались в разных заговочных файлах стандартной библиотеки языка C, которая потом унаследовалась C++ с добавлением пространств имён и перегрузок. Только начиная с C++17 все версии перегрузок доступны как в `cmath`, так и в `cstdlib`, а забытые программистами квалификации приводят к тому что старые программы перестают компилироваться с современными версиями стандартных библиотек.

Читатель, вероятно, заметил, что в некоторых ситуациях требуются несколько неявных преобразований в одном месте:

```
1  int x = 12;
2  double y = x; // lvalue-to-rvalue, затем intergal-floating conversion.
```

Максимальное количество встроенных в язык неявных преобразований, которые могут быть неявно выполнены в одном месте ограничено ***стандартной последовательностью преобразований (standard conversion sequence)***, которая может включать в себя максимум по одному пункту из нижеперечисленных:

1. ***Разлагающие преобразования (decay)***: разложение lvalue в rvalue и некоторые пока нам не известные.
2. Одно из основных преобразований, к которым относятся все рассмотренные нами преобразования между арифметическими типами.
3. Преобразования указателей функций и
4. преобразования квалификаторов нами пока не рассмотрены.

Ранжирование цепочек стандартных преобразований с точки зрения разрешения перегрузок осуществляется по наихудшему из всех преобразований цепочки. Это не внесёт изменений в дальнейшее рассмотрение, поскольку все неизвестные нам преобразований имеют ранг «точное соответствие» и не ухудшают ранга своим присутствием. Для различия преобразований, включающих их, вводятся дополнительные правила сверх системы рангов, которые мы рассмотрим вместе с соответствующими преобразованиями.

4.21. Псевдонимы типов. Типы фиксированной ширины.

Пока мы имеем только две гарантии относительно ширин целых типов: узкие символьные занимают 1 байт, а типы рангов начиная с `int` не имеют отрицательных эффектов по скорости обработки. В реальных задачах почти всегда имеются интервалы обрабатываемых значений или явные указания на ширины типов и их представления. Чтобы свести разнообразие ширин фундаментальных типов, отражающие аппаратные характеристики, в общую точную систему, введём понятие псевдонима типа.

Определением ***псевдонима типа (type alias, typedef)*** называют определение нестандартного вида, вводящее в программу имя типа, соответствующего другому, уже имеющемуся. Оно имеет синтаксис

`using` *identifier* `=` *type* `;`

Это определение может быть дано в любой области видимости. Оно вводит в программу имя, заданное идентификатором, которое можно использовать в качестве имени типа, заданного после пунктуатора `=`, в том числе в роли спецификатора типа, например:

```
1  using byte = unsigned char;
2
3  bool low_bit(byte b)
4  {
5      return b%2;
6  }
```

В таком простейшем случае псевдоним типа может использоваться для сокращения длинных имён типов, что особенно пригодится нам в будущем для сокращения записей сложных конструкций создания производных типов.

Другая задача псевдонимов типа — введение унифицированных имён для типов, которые могут по разному выражаться через фундаментальные на разных платформах. В заголовочном файле `stdint` в пространстве имён `std` описано множество псевдонимов типов с имена по схеме:

1. Опциональный префикс `u`. Без него речь идёт о знаковом типе, с ним — о беззнаковом.
2. Фиксированные символы `int_`.
3. Опциональный интерфикс точности `_least` или `_fast`.
4. Ширина в битах: `8`, `16`, `32` или `64`.
5. Фиксированные символы `_t`.

Суффикс `_t` в именах производных типов будет нам часто встречаться, чтобы отличать имена типов от имён сущностей другого вида. В зависимости от интерфикса точности, эти псевдноимы соответствуют следующим типам на любой архитектуре:

- Без интерфикса: в точности указанное число бит, которые все используются (без битов заполнения). Для знаковых типов также гарантируется представление в дополнительном коде, в отличии от следующих двух вариантов. Если типа с указанными характеристками в данной среде нет, то нет и соответствующего описания псевдонима в реализации стандартной библиотеки. На практике это встречается редко, и этими вариантами псевдонимов пользуются большинство программистов. Например, `std::uint16_t` — беззнаковый 16-битный тип (если есть).
- Интерфикс `_least`: тип указанной ширины, если таковой имеется, иначе ближайший по ширине более широкий. Имеется в любой реализации.
- Интерфикс `_fast`: тип как минимум указанной ширины, с которым наиболее быстро производить вычисления. Даже если есть тип указанной ширины в битах, быстрые псевдонимы могут соответствовать более широким типам, например, отражая дополнительные расходы на целочисленные повышения. Также имеется в любой реализации.

Таким образом, функцию умножения двух 32-битных беззнаковых чисел можно записать так:

```
1  #include <cstdint>
2
3  // Математически, ширина результата умножения 32 бит на 32 - 64.
```

```
4  std::uint64_t mul32x32(std::uint32_t x,std::uint32_t y)
5  {
6      // Если только на данной архитектуре тип int не 64-битный,
7      // то в умножении std::uint32_t на самого себя целочисленного
8      // повышения не будет, и результат будет того же типа - это
9      // лишь младшая половина результата! Преобразуем явно один
10     // из операндов к 64-битному типу, чтобы вычисления и результат
11     // формально были в нём. В подобных конструкциях компилятор не
12     // делает лишних преобразований или вычислений в излишне широких
13     // типах в машинном коде, а лишь предоставляет доступ к полному результату.
14     return static_cast<std::uint64_t>(x)*y;
15 }
```

4.22. Выбор арифметических типов

Подведём итоги нашему рассмотрению арифметических типов и рассмотрим обычный порядок выбора одного из них в зависимости от ситуации.

- При использовании сторонних интерфейсов, следует придерживаться уже принятых ими решений. Например, если функция возвращает значение некоторого арифметического типа, следует использовать объект того же типа для его хранения, если с нашей стороны дополнительных требований нет.
- Для логических значений следует использовать тип `bool`. Использование для этих значений других типов (например, `int`) не обладает необходимой выразительностью.
- Использовать типы с плавающей точкой следует только при необходимости из-за проблем, связанных с их природой. При отсутствии специальных требований используется тип `double`, при необходимости представления большого количества значений, используется тип `float`, когда его малой точности достаточно. Тип `long double` практически не используется.
- Узкие символьные типы используются для работы с узкими символами и байтами в силу гарантии на размер их представления. При работе с их значениями как числами, предпочитают `unsigned char` в силу его определённой знаковости, иначе используется обычный `char`.
- В простых случаях для целых чисел, не выходящих за рамки 16 бит с учётом знака, гарантированных стандарту для этого типа, используется тип `int` как наиболее простой и эффективный.
- В большинстве практических задач имеются конкретные требования по ширине типов, их характеристикам или диапазонам значений. Для следования им используют типы фиксированной ширины.
- Во всех случаях выше, когда имеются варианты выбора, следует предпочитать по возможности знаковые типы рангом не ниже `int` (для эффективности). Использования беззнаковых типов избегают из-за определённости их переполнения, которое чаще всего не нужно в задачах, но при этом скрывает ошибки переполнения и мешает оптимизациям компилятора.

4.23. Использование драйвера компиляторов clang/gcc

После первичного разбора нашей первой программы перейдём непосредственно к её трансляции и запуску.

В данной книге автор будет придерживаться компилятора языков C/C++ clang, который является частью инфраструктуры LLVM, предназначенной для написания трансляторов, оптимизаторов, сред выполнения и других инструментальных средств путём использования общего промежуточного представления программ, не зависящего от языка исходного текста и целевой среды выполнения.

Как нам уже известно, трансляция программы на C++ формально состоит из нескольких шагов, на практике их также несколько, хотя точного соответствия процессу, описанному в стандарте языка обычно нет. Даже без использования систем сборки, напрямую с компилятором обычно не работают. Вместо этого взаимодействуют с ***драйвером компилятора (compiler driver)***, который берёт на себя вызов всех необходимых инструментальных средств, выполняющих шаги трансляции программы на языке C++. Даже если считать предварительную обработку частью трансляции, потребуется ещё как минимум вызов компоновщика. C

этой точки зрения драйвер компилятора похож на систему сборки, но выполняет все требуемые команды без распараллеливания, и всегда все, независимо от изменений в исходных файлах. Также драйвер компилятора берёт на себя задачу указания компилятору и компоновщику большого числа опций, необходимых для их корректной работы в имеющейся среде.

Драйвер компилятора clang для языка C++, установленный в системные каталоги ОС, обычно называется `clang++`. Это неинтерактивная программа с интерфейсом командной строки, бóльшая часть опций которой совместима с исторически наиболее популярным компилятором gcc. Случаи, специфические для clang, мы будем отмечать отдельно, в остальном излагаемый в данной книге материал применим и к этому компилятору.

Драйверу компилятора в качестве аргументов программы указываются имена входных файлов. Это могут быть имена как файлов с исходным текстом, так и продукты более поздних фаз трансляции, сохранённые в файлы. Содержимое файлов и, соответственно, какой обработки они требуют, определяется драйвером компилятора по расширениям файлов. Если расширение файла нестандартное, можно указать опцией `-x` с соответствующим аргументом тип всех последующих входных файлов, вместо автоматически определённого.

По умолчанию драйвер компилятора компилирует все входные файлы, а затем компонует их, создавая образ программы, то есть доделывает весь процесс трансляции до конца по данным входам. С помощью указания одной из дополнительных опций можно остановить процесс трансляции ранее, получив на выходе соответствующие промежуточные результаты. Результат записывается в файл с именем, зависящим от выбранного этапа остановки, чаще всего с тем же базовым именем, но другим расширением, но оно всегда может быть задано явно опцией `-o`.

Драйверу компилятора можно указывать многие опции вызываемых им инструментальных средств, и он передаст их им напрямую. Для более экзотических опций существуют опции-префиксы, указывающие драйверу компилятора, что следующую опцию необходимо передать соответствующей программе.

В любом случае, когда возникают проблемы с передачей опций, показать команды, выполняемые драйвером компилятора, со всеми параметрами можно указанием ему опции `-v` (опция `-###` только показывает команды, но не выполняет их).

В таблице 4.5 показаны основные представления транслируемой программы на различных этапах и их соответствие режимам работы драйвера компилятора.

Тип файла	Расширение	Тип для опции `-x`	Опция остановки на этом этапе	Префикс опций инструментальн средства
Исходный текст	`.cpp` (`.cxx`, `.cc`,...)	`c++`	`-fsyntax-only`	N/A
Результат предварительной обработки (как текст)	`.ii` (вывод по умолчанию на стандартный поток)	`c++-cpp-output`	`-E`	`-Xpreprocesso` (`-Wp,`)
LLVM IR (не для `gcc`)	`.ll`	?	`-S -emit-llvm`	`-Xclang`
Ассемблер целевой платформы	`.s`	`assembler`	`-S -masm=intel`	`-Xassembler` (`-Wa,`)
Объектный файл	`.o`	не требуется	`-c`	см. выше `-Xlinker`
Образ программы	без расширения (UNIX, a.out по умолчанию)	N/A	—	(`-Wl,`)

Таблица 4.5: Этапы работы драйвера компилятора

Рассмотрим данные режимы подробнее.

При указании опции `-fsyntax-only` производится только синтаксическая и семантическая проверка файлов исходного текста, никакого вывода не производится. Файлы исходного текста на языке C++ принято сохранять в файлах с расширением `.cpp`, также используются `.cxx`, `.cc` и редко некоторые другие. Запишем ранее разбиравшийся пример сложения двух введённых чисел в файл `test.cpp`.

Опция `-E` позволяет остановится на этапе предварительной обработки. Формально, её результат — последовательность токенов, но при записи в выходной файл она преобразуется назад в текст, чтобы быть человекочитаемой. В этом режиме без указания имени выходного файла опцией `-o` вывод осуществляется на стандартный поток вывода. Запустив команду

`clang++ test.cpp -E`, вы получите десятки тысяч строк вывода, последние из которых приведены ниже:

```
// --- начало опущено ---

# 45 "/usr/include/c++/v1/iostream" 3

namespace std {inline namespace __1 {
    extern __attribute__ ((__visibility__("default"))) istream cin;
    extern __attribute__ ((__visibility__("default"))) wistream wcin;

    extern __attribute__ ((__visibility__("default"))) ostream cout;
    extern __attribute__ ((__visibility__("default"))) wostream wcout;

    extern __attribute__ ((__visibility__("default"))) ostream cerr;
    extern __attribute__ ((__visibility__("default"))) wostream wcerr;
    extern __attribute__ ((__visibility__("default"))) ostream clog;
    extern __attribute__ ((__visibility__("default"))) wostream wclog;
} }
# 2 "test.cpp" 2

int main()
{
  std::cout << "Input two real numbers: ";
  double a,b,c;
  if(!(std::cin >> a >> b)){
    std::cerr << "Input error!\n";
    return 1;
  }
  c = a*b;
  std::cout << a << " * " << b << " = " << c << '\n';
}
```

Это похоже на строки нашего файла исходного текста, за исключением пропавшей директивы `#include` — она была выполнена и заменена соответствующим текстом из заголовочного файла. Объём этого текста в примерно 40 тысяч строк объясняется не длиной самого файла `iostream`, а длиной всего дерева включённых заголовочных файлов. Когда в интерфейсе единицы трансляции требуются описания из других единиц трансляции, в самих заголовочных файлах встречаются директивы `#include`, которые также обрабатываются. Полученный текст есть результат последовательной записи многих включений из всех транзитивных зависимостей `iostream`, который использует многие части стандартной библиотеки. Значительная часть этого текста нам пока не ясна, но в приведённом фрагменте видны описания стандартной формы имён `cin`, `cout` и `cerr` в пространстве имён `std`, что нам и требовалось.

Компилятор clang построен на базе библиотек LLVM, предоставляющих инструментарий по создания различных инструментальных средств трансляции, оптимизации, выполнения и отладки программ. В данной инфраструктуре имеется внутреннее представление программ LLVM Intermediate Representation (LLVM IR), которое по уровню представления похоже на машинно-независимый ассемблер. Результат работы clang генерируется именно в этом представлении, а уже его полной оптимизацией и трансляцией в машинной код конкретной архитектуры занимается сама LLVM. Это представление может быть сохранено в файл в человекочитаемом или двоичном форматах, первый из них выбирает комбинаций опций `-S -emit-llvm`. Детали этого представления важны в основном для тех, кто интересуется оптимизациями платформы LLVM, и в этой книге обсуждаться не будут.

Получить ассемблер x86-64 позволяют опции `-S -masm=intel`, вторая требуется, поскольку для UNIX-производных инструментальных средств по умолчанию используется диалект AT&T. В полученном файле с расширением `.s` имеется функция `main`, приведём здесь её начало, в котором видна стандартная форма пролога (строки 5–10, не учитывая директив `.cfi`):

```
; --- фрагмент ---
main:                                   # @main
.cfi_startproc
# BB#0:
  push    rbp
.cfi_def_cfa_offset 16
.cfi_offset rbp, -16
```

```
 8   mov     rbp, rsp
 9 .cfi_def_cfa_register rbp
10   sub     rsp, 80
11   movabs  rdi, offset _ZNSt3__14coutE
12   movabs  rsi, offset .L.str
13   mov     dword ptr [rbp - 28], 0
14   call    _ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc
15 ; --- фрагмент ---
```

В режимах работы, подразумевающих трансляцию, драйвер компилятора принимает опции оптимизации в формате `-O`уровень:

- `-O0` — без оптимизаций, используется по умолчанию. Транслятор делает всё в точности формально, как предписывает язык. Зачастую, очень неэффективно. Тем не менее, это почти всегда самый быстрый режим, используемый при отладке программ, где важна скорость компиляции.
- `-O1` — минимальный режим оптимизации, только простые и однозначно выгодные преобразования. Используется редко.
- `-Og` — уровень с минимальными оптимизациями, которые пригодны для отладки программ, т.к. не нарушают их структуру относительно формально предписываемой языком, т.е. отношение между операторами программы и машинным кодом всё ещё простое. Введён относительно недавно, применяется редко, для clang'а в настоящее время эквивалентен `-O1`.
- `-O2` — стандартный уровень оптимизаций. Включает большинство оптимизаций, полезных для широкого круга программ. Очень часто используется при компиляции рабочей версии программ, включая сборку дистрибутивов операционных систем из пакетов с открытым исходным кодом.
- `-O3` — максимальный уровень оптимизаций. Сверх второго уровня выполняет дополнительные оптимизации, которые затратны по времени, увеличивают объём кода, и не обязательно повышают производительность. Выгодно ли его использовать вместо `-O2`, определяется характером транслируемой программы, в основном положительный эффект проявляется на математическом ПО. Среди всех уровней оптимизации, пожалуй, наиболее ярко проявляет неопределённое поведение в программах, его имеющих, но не демонстрирующих на более низких уровнях оптимизации, по этой причине не применяется для компиляции дистрибутивов ОС целиком, содержащих множество старых программ, не следующих современным уровням соответствия кода стандартам.
- `-Ofast` — сверх `-O3` включает дополнительные оптимизации, жертвующие ради скорости выполнения даже соответствию стандарту. Используется только математическим ПО, пытающимся выжать из транслятора абсолютно всё. За счёт выхода за пределы стандарта хрупок и не рекомендуется.
- `-Os` — оптимизации, схожие с `-O2`, отдающие приоритет размеру кода взамен его производительности. Применяется при сборке программ для систем с ограниченной памятью (микроконтроллеры и подобные). В патологических случаях может быть даже быстрее `-O2` за счёт уменьшения давления на кэш инструкций процессора.
- `-Oz` (только clang) — оптимизации, жертвующие всем в пользу уменьшения размера кода, усиленная версия `-Os`. Применяется редко.

В современной практике в основном используется уровень `-O0` при отладке, а для итоговой версии применяют `-O2` или `-O3`. Эти опции на самом деле переключают большое количество более детальных настроек компилятора, которые можно контролировать и отдельно. Например, за опускание кадра стека отвечает опция `-fomit-frame-pointer`, названия подобных опций начинаются с `f` - Features. При отключении опций негативные формы имеют частицу `no-` перед именем опции: `-fno-omit-frame-pointer`.

Приведём для сравнения соответствующий фрагмент, полученный командой `clang++ test.cpp -S -masm=intel -O3`:

```
1 ; --- фрагмент ---
2 main:                                   # @main
3 .cfi_startproc
4 # BB#0:
5   sub     rsp, 40
6 .cfi_def_cfa_offset 48
```

```
 7    mov     edi, offset _ZNSt3__14coutE
 8    mov     esi, offset .L.str
 9    mov     edx, 24
10    call    _ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamI
11  ; --- фрагмент ---
```

В этом случае пролог в стандартной форме отсутствует, и первая вызываемая из `main` функция оказалась другой, что неудивительно, т.к. в режиме полной оптимизации структура исходного текста часто полностью теряется в угоду производительности.

Опция `-c` транслирует входные файлы в объектные, но не компонует их. Полученные объектные файлы с расширением `.o` не являются текстовыми, и для анализа их содержимого в следующих главах нам потребуются дополнительные инструментальные средства. Это уровень, который часто используется система сборки, чтобы вызвать несколько процессов трансляции параллельно.

Наконец, без указания специальных опций трансляция программы дойдёт до конца с получением образа программы. Перед примером на этот случай перечислим дополнительные опции драйвера компилятора, которые необходимо использовать для компиляции программ по современным стандартам:

- `-std=c++17` — выбор версии используемого стандарта языка. Для совместимости драйверы компилятора не используют самую новую версию из поддерживаемых по умолчанию, поскольку, несмотря на приложенные в этом направлении усилия, версии языков C++ не полностью совместимы в сторону более новых версий. Без этой опции транслятор не будет принимать новые языковые конструкции, даже если они ему известны и корректны.
- `-pedantic-errors` — педантичное следование указанному предыдущей опцией стандарту. Помимо стандартных средств языка, трансляторы включают и дополнительный функционал, являющийся нестандартным. В процессе изучения языка начинать следует со стандартных средств, и, чтобы нестандартный код не принимался компилятором, отклонения от стандарта необходимо приравнять к ошибкам. Это позволяет учиться писать переносимый на другие трансляторы код с самого начала обучения.
- `-Wall` — включить основные предупреждения. Помимо обнаруживаемых транслятором ошибок, доступны и дополнительные диагностические сообщения — ***предупреждения (warning)***. Эта опция включает их наиболее полезный основной набор.

При отладке программы вместе уровнем оптимизации `-O0` или `-Og` указывают также опцию `-g`, которая необходима для генерации отладочной информации, без которой многие другие инструментальные средства не смогут локализовывать ошибки в программе на уровне её исходного текста.

Теперь можно, наконец, собрать нашу первую программу. Поместим её образ в файл с именем `testprog`:

```
% clang++ -std=c++17 -pedantic-errors -Wall -O0 -g test.cpp -o testprog↵
% ./testprog↵
Enter two real numbers: 2.34 5.67↵
3.4 * 5.6 = 19.04
%
```

Как показано в примере, для запуска программы из текущего каталога пришлось указать его явно в виде `"."`. Это необходимо, поскольку имя команды без разделителей компонент путей `/` считается именем внешней программы с поиском её в системных каталогах программ, куда в UNIX-системах текущий каталог традиционно не входит по соображениям безопасности.

> При использовании файлов с нестандартными расширениями необходимо указывать их тип драйверу компилятора опцией `-x` с параметрами, указанными в третьем столбце таблицы 4.5. Например, проверим, что пустой файл исходного текста синтаксически корректен. Чтобы не создавать пустой файл, воспользуемся именем нулевого устройства `/dev/null` в UNIX-системе, которое при чтении сразу возвращает конец файла.
>
> ```
> % clang++ -std=c++17 -pedantic-errors -Wall -fsyntax-only -x c++ /dev/null↵
> % echo $?↵
> 0
> %
> ```
>
> В последнем столбце таблицы 4.5 указаны опции передачи неизвестных драйверу

> компилятору аргументов вызываемым им программам. Использовать из них можно те, что относятся к вызываемым средствам, например, при сборке в стандартном режиме от исходного текста к образу программы можно указывать их все: для препроцессора, транслятора, ассемблера и компоновщика. Первый формат в виде опций с именем `-Хпрограмма` указывает, что следующая опция предназначается соответствующей программе. Вторая форма передаёт сколько угодно опций сразу, разделяя их запятыми как часть самой себя. Например, может потребоваться передать компоновщику опцию `-O1` для оптимизации образа программы. Поскольку одноимённая опция есть у транслятора, драйвер компилятора передаст её ему в первую очередь, а для передачи её компоновщику можно использовать на выбор либо `-Xlinker -O1`, либо `-Wl,-O1`.

4.23.1. Визуализация представления программы в процессе трансляции

Изученные нами возможности C++ не всегда имеют видимое отображение в синтаксисе — например, неявные преобразования. Выработка навыков распознавания таких конструкций важна, и помочь в ней может сравнение своего мнения с правильным.

Современные компиляторы редко работают в режиме «прочёл часть текста — перевёл», достаточно сложные языки — такие, как современный C++ — не позволяют использовать такую ***однопроходную (single pass)*** схему в принципе. Вместо этого компилятор сначала переводит текст программы в некоторое внутреннее представление, с которым удобно работать — ***абстрактное синтаксическое дерево (abstract syntax tree, AST)***. После этого проверка сложных правил сементики, оптимизации и генерация кода осуществляются за счёт работы с этим внутренним предствлением.

Компилятор clang отличает то, что структура его AST для языка C++ во многом повторяет формальные правила языка, в том числе она содержит многие неявные конструкции в явном виде. С помощью опции `-ast-dump`, передаваемой непосредственно компилятору, можно запросить вывод этого предствления. Эта опция не известна драйверу компилятору, поэтому для правильной её передачи необходимо указать перед ней `-Xclang`. Обычно в таком режиме никакого другого вывода не требуется, так что используется и опция `-fsyntax-only`.

Покажем использование этой возможности на примере:

```
1  // ast-test.cpp
2
3  int f(int x)
4  {
5      return x+1.5;
6  }
```

```
~ % clang++ -std=c++17 -fsyntax-only -Xclang -ast-dump ast-test.cpp↵
TranslationUnitDecl 0x1fac8d8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x1face90 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x1facb70 '__int128'
|-TypedefDecl 0x1facf00 <<invalid sloc>> <invalid sloc> implicit __uint128_t
    'unsigned __int128'
| `-BuiltinType 0x1facb90 'unsigned __int128'
|-TypedefDecl 0x1fad248 <<invalid sloc>> <invalid sloc> implicit __NSConstantString
    'struct __NSConstantString_tag'
| `-RecordType 0x1facff0 'struct __NSConstantString_tag'
|   `-CXXRecord 0x1facf58 '__NSConstantString_tag'
|-TypedefDecl 0x1fad2e0 <<invalid sloc>> <invalid sloc> implicit
    __builtin_ms_va_list 'char *'
| `-PointerType 0x1fad2a0 'char *'
|   `-BuiltinType 0x1fac970 'char'
|-TypedefDecl 0x1fe1be0 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list
    'struct __va_list_tag [1]'
| `-ConstantArrayType 0x1fad5c0 'struct __va_list_tag [1]' 1
|   `-RecordType 0x1fad3d0 'struct __va_list_tag'
|     `-CXXRecord 0x1fad338 '__va_list_tag'
`-FunctionDecl 0x1fe1d18 <ast-test.cpp:3:1, line:6:1> line:3:5 f 'int (int)'
  |-ParmVarDecl 0x1fe1c50 <col:7, col:11> col:11 used x 'int'
  `-CompoundStmt 0x1fe1ec8 <line:4:1, line:6:1>
    `-ReturnStmt 0x1fe1eb0 <line:5:5, col:14>
      `-ImplicitCastExpr 0x1fe1e98 <col:12, col:14> 'int' <FloatingToIntegral>
```

```
`-BinaryOperator 0x1fe1e70 <col:12, col:14> 'double' '+'
  |-ImplicitCastExpr 0x1fe1e58 <col:12> 'double' <IntegralToFloating>
  | `-ImplicitCastExpr 0x1fe1e40 <col:12> 'int' <LValueToRValue>
  |   `-DeclRefExpr 0x1fe1df8 <col:12> 'int' lvalue ParmVar 0x1fe1c50 'x' 'int'
  `-FloatingLiteral 0x1fe1e20 <col:14> 'double' 1.500000e+00
```

(В терминале приведённый здесь текст будет раскрашен для читаемости.)

Перед нами дерево разбора нашей единицы трансляции, сама она представлена корневым элементом типа `TranslationUnitDecl`. Нас интересует последний его поэлемент типа `FunctionDecl`, соответствующий написанной нами функции. (Идущие перед этим описания псевдонимов типов даны транслятором неявно для поддержки собственных расширений.) В этом элементе видны его местоположение в файле исходного текста, имя и тип сущности. У него два дочерних элемента: первый типа `ParmVarDecl`, соответствующий описанию параметра функции, и второй типа `CompoundStmt` — составной оператор, являющийся телом функции (так что это описание функции — определение).

В этом составном операторе один дочерний элемент — `ReturnStmt`, т.е. оператор `return`. У этого оператора дочерний элемент присутствует и соответствует выражению, в нём содержащемуся. Разберём его, начиная с конца, т.е. с его минимальных неделимых элементов.

Самый последний элемент в приведённом примере — `FloatingInterval` — литерал с плавающей точкой, соответствующий второму операнду сложения в исходном тексте. Помимо значения указан его тип. При возникновении проблем с выявлением типов литералов, эта информация может пригодиться.

Перед ним на некотором уровне вложенности расположен узел типа `DeclRefExpr` — «выражение, ссылающееся на описание». Таким образом в AST представлены результаты поиска имён — ссылки на описания, которым в исходном тексте соответствуют имена. В конце представления этого узла указана информация о найденном описании. В общем случае все узлы дерева могут быть однозначно идентифицированы номерами в форме шестнадцатеричных литералов, которые указаны сразу после их типа в начале строки. В нашем случае результат поиска имён — определение параметра функции `x`.

Родительским для данного элемента является `ImplicitCastExpr` — явное представление в дереве синтаксического разбора неявного преобразования. Судя по указанному в нём типу `LValueToRValue` — это преобразование леводопустимого выражения.

Далее следует ещё одно неявное преобразование типа `IntegralToFloating` — это предписываемое алгоритмом обычных арифметических преобразований вычисление. Выше расположен узел `BinaryOperator`, который объединяет два дочерних одной бинарной операцией, сложения в нашем случае. Перед использованием в качестве возвращаемого выражения `return`, потребовался ещё один узел неявного преобразования к типу результата.

Таким образом, многие неявные правила семантики языка нашли в AST явный вид. При использовании этого режима для выяснения правил разбора тех или иных конструкций языка следует минимизировать транслируемый фрагмент, поскольку для нетривиальных единиц трансляции вывод будет очень объёмным. Например, при использовании директив включения заголовочных файлов придётся просматривать только самый конец вывода.

4.24. Система сборки CMake

Для любого нетривиального проекта незаменимой оказывается система сборки, обеспечивающая, как нам уже известно, максимальную скорость трансляции. Современные системы сборки выполняют и другие функции, рассмотрим их на примере системы сборки CMake.

Системе сборки требуется информация о форме графа зависимостей исходных, промежуточных и результирующих файлов, часть которой должна быть задана пользователем. CMake различает два основных каталога в своей работе:

- ***Каталог исходных файлов (source directory)*** — каталог с исходными файлами на языке C++ и другими файлами, необходимыми для сборки рабочей программы, включая файл с конфигурацией CMake. Даже для программ из одной единицы трансляции рекомендуется создавать отдельные каталоги.
- ***Каталог сборки (build directory)*** — каталог, где размещаются все промежуточные и итоговые файлы, создаваемые в процессе сборки. Традиционно это подкаталог `build` каталога исходных файлов, или полностью отдельный от каталог вне него. Автор рекомендует второй вариант, поскольку он позволяет иметь несколько каталогов сборки с разной конфигурацией одновременно. Ситуации, где каталог сборки совпадает с каталогом исходных

текстов, CMake формально поддерживаются, но не рекомендуются, т.к. смешивать исходные и результирующие файлы неудобно, хотя бы с точки зрения быстрого удаления последних: в рекомендуемом варианте достаточно удалить один каталог целиком, а не выискивать продукты среди исходных текстов.

Файл конфигурации CMake должен называться `CMakeLists.txt` и содержать текст на языке CMake. Этот, по факту, императивный язык следует применять в декларативном стиле для описания ***целей (target)*** — итоговых продуктов сборки и их характеристик, включая зависимости. Команды языка CMake состоят из имени команды, за которым следует список аргументов, разделённых пробелами в круглых скобках. Команды разрешается разбивать на несколько строк. Система типов языка CMake примитивна — есть только строки в виде последовательностей символов. Для задания последовательностей символов с пробелами внутри можно заключать их в двойные кавычки, а также пользоваться escape-последовательностями, аналогичными C++. В требуемых контекстах числа, записанные строками, трактуются как числа. В качестве логических значений строки `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, а также заканчивающиеся на `-NOTFOUND` (все в любом регистре) трактуются как ложь, а остальные — как истина. Списки представляются в виде последовательностей элементов, разделённых пробелами или точками с запятой.

В процессе обработки файла конфигурации, CMake отслеживает значения используемых им переменных. Многие переменные создаются самой системой сборки, файл конфигурации может изменять их значения и создавать собственные. Создание собственных переменных в современном стиле использования CMake стараются свести к минимуму. Помимо обычных переменных, теряющих свои значения после каждого выполнения системы сборки, имеются также ***переменные кэша (cache variables)***, сохраняющие значения между запусками CMake в файле `CMakeCache.txt` в каталоге сборки. Они предназначены для конфигурации пользователем процесса сборки, и изменяться самой системой сборки в большинстве прикладных случаев не должны.

В отличие от C++, но схоже с языком командной оболочки UNIX, аналога преобразования lvalue-to-rvalue в языке CMake нет — чтобы заменить имя переменной её текущим значением, необходимо заключить его в фигурные скобки и поставить знак `$` перед ними. Описывать переменные не нужно — они создаются по факту первого присвоения им значения. Для задания значений обычным переменным используется команда `set`, например:

```
# Однострочные комментарии начинаются со знака #.
# Записать истину в переменную USE_SOMETHING.
set(USE_SOMETHING ON)
# Записать значение переменной USE_SOMETHING в другую.
set(USE_OTHER_THING ${USE_SOMETHING})
```

Рассмотрим минимальный файл конфигурации CMake для нашей тестовой программы:

```
cmake_minimum_required(VERSION 3.8 FATAL_ERROR)

project(test_program LANGUAGES CXX)

add_executable(${PROJECT_NAME} test.cpp)

target_compile_features(${PROJECT_NAME}
                        PRIVATE cxx_std_17)
set_target_properties(${PROJECT_NAME} PROPERTIES
                      CXX_EXTENSIONS OFF)
```

Первая команда, проверяет наличие необходимой версии CMake, что сразу останавливает сборку, если используемая версия CMake ниже требуемой. Нам понадобится как минимум 3.8, т.к. в ней появилась поддержка C++17. Аргумент `FATAL_ERROR` неизвестен ранним версиям CMake, в которых эта команда лишь выдавала предупреждения при несоответствии версий, вызывая ошибку и в них (новые версии его игнорируют). С этой команды должен начинаться любой файл конфигурации CMake, т.к. выставляет настройки совместимости CMake в указанную версию. Это позволяет CMake сохранять совместимость с файлами конфигурации, написанными для предыдущих версий, даже при изменении в новых поведения старых команд: увидев требование более ранней версии, CMake возвращается к старому поведению.

Также должна присутствовать команда `project`, задающая новый проект — группу целей (одну в нашем случае). Она задаёт значение некоторых переменных, в первую очередь `PROJECT_NAME` в указанное имя, это позволит не дублировать его далее. Из её опциональных

частей мы используем указание языков, используемых проектом — в нашем случае, только C++ (по умолчанию также активируется C, но он нам пока не нужен).

Команда `add_executable` добавляет к проекту цель типа «исполняемый файл» и позволяет сразу задать список исходных файлов, от которых она зависит. В качестве имени цели, которая станет именем образа программы, мы используем имя проекта, в качестве списка имён файлов исходного текста — наш единственный `test.cpp`.

Команда `target_compile_features` указывает необходимые для сборки цели возможности компилятора, в нашем случае — поддержку стандарта C++17. Область действия этих требований — только сама цель (`PRIVATE`).

Наконец, команда `set_target_properties` позволяет задать произвольные характеристики цели. Список файлов исходного текста и возможности транслятора тоже можно задать ей, но для этого предпочитают применённые нами более специализированные команды, что мы и сделали выше. Для свойства цели `CXX_EXTENSIONS` таких команд нет, и приходится пользоваться общей формой. Это свойство по умолчанию содержит истину, что приводит к передаче драйверу компилятора опции `-std=gnu++17` вместо требуемой нами `-std=c++17`, что активирует расширения транслятора, которые мы договорились не использовать. Остальные рекомендованные нами опции (`-pedantic-errors -Wall`) желательны, но не семантически необходимы для корректной трансляции, и поэтому в описание процесса сборки не входят.

> Современный стиль использования CMake подразумевает отказ от многих традиционных паттернов написания файлов конфигурации. В первую очередь не стоит менять переменные кэша, с именами, начинающимися на `CMAKE_`, отвечающими за настройки по умолчанию — это настройки для пользователя системы сборки, и задаются им, заменять их не следует. Не следует применять команды задания свойств на уровне каталогов, вместо этого следует использовать аналоги для целей (например, вместо `add_compile_options` следует использовать `target_compile_options`). При задании конкретных опций транслятора не следует добавлять их без проверок, или проверяя сам вид транслятора, следует проверять их на фактическое функционирование — как это делать правильно, будет рассказано позднее.

Перед сборкой проекта его необходимо ***сконфигурировать (configure)*** в новом каталоге сборки, что выполняет две функции. Во-первых, CMake анализирует среду, в которой производится трансляция: ищет компилятор C++, выясняет его поддерживаемые опции, наличие внешних библиотек, инструментальных средств и др. Этот этап присутствует у большинства таких систем и позволяет настроить процесс сборки с учётом имеющегося окружения. Во-вторых, CMake, на самом деле, является не самостоятельной системой сборки, а генератором конфигураций для различных настоящих систем сборки и интегрированных сред. Это позволяет ему ближе интегрироваться с родными для ОС системами сборки проектов и предоставляет программисту дополнительную гибкость. В конце данного этапа создаются файлы конфигурации для используемой настоящей системы сборки, которая будет выполнять фактическую работу.

При вызове CMake необходимо последним аргументом указать каталог исходных файлов или каталог сборки. В первом случае каталогом сборки станет текущий и произойдёт начальная конфигурация, если он был пуст, иначе будут лишь внесены требуемые изменения. Во втором случае каталог исходных файлов будет считан из файла кэша CMake.

При первом вызове в новом каталоге сборки потребуется указать опции `-G Ninja` — это выбор используемого генератора для настоящей системы сборки, в нашем случае предпочтём использование ninja вместо make, используемого на большинстве систем по умолчанию. ninja разрабатывался именно с целью быть максимально быстрой системой сборки, в чём он выигрывает у make, при этом синтаксис его файлов конфигурации скуден и предназначен не столько для написания человеком, сколько для машинной генерации — эту роль и выполняет CMake. Помимо этого, опциями `-D` можно задать или заменить имеющиеся значения переменных кэша, имена переменных и значения указываются через знак `=` слитно с этой опцией. Нам потребуются следующие значения переменных:

- `CMAKE_CXX_COMPILER=clang++` — используемый компилятор C++. Без его указания CMake выберет более традиционный gcc.
- `CMAKE_BUILD_TYPE=Debug` — выбор одной из стандартной конфигураций сборки, отладочной в нашем случае. Это укажет CMake настроить транслятор для отладки нашей программы, прибавив для clang опцию `-g`.
- `CMAKE_CXX_FLAGS="-pedantic-errors -Wall"` — дополнительные опции для компиляции, которые мы желаем использовать.

- `CMAKE_VERBOSE_MAKEFILE=ON` — отображать команды сборки в явном виде. Полезно видеть команды, выполняемые системой сборки, чтобы понять, есть ли в них ошибки, которые не приводят к сбоям (неудачные команды сборки отображаются всегда).

Большинство значений кэша можно поменять и применить изменения к системе сборки в имеющемся каталоге путём вызова CMake с опциями `-D` для изменившихся значений, или редактированием файла кэша (он текстовый в формате `имя=значение`), с последующим вызовом CMake. Создавать новый каталог сборки придётся только при смене генератора или компилятора.

Запустить CMake в режиме, собственно, сборки опцией `--build` с указанием каталога сборки. При этом по умолчанию собирается особая цель по умолчанию, включающая все цели проекта. Можно собирать и отдельные цели по их именам, путём их задания дополнительной опцией `--target` в этом режиме. Особая цель `clean` удаляет продукты сборки, что может быть полезно, чтобы заставить систему сборки собрать всё полностью заново даже при отсутствии изменений. В большинстве случаев это не требуется.

Дальнейшее общение с CMake покажем на примере.

```
~ % # начинает однострочный комментарий и в языке оболочки.↵
~ % # Переместим test.cpp в отдельный каталог.↵
~ % # && используется аналогично C++ для выполнения следующих команд↵
~ % # только, если прошлые удачны.↵
~ % # mkdir - создание каталога.↵
~ % # mv - перенос указанных файлов в другое место.↵
~ % mkdir cmake-test && cd cmake_test && mv ../test.cpp .↵
~/cmake_test % # Создадим текстовый файл конфигурации CMake. ↵
~/cmake_test % # cat выводит все указанные файлы на стандартный вывод,↵
~/cmake_test % # если аргументов нет - читает стандартный ввод.↵
~/cmake_test % # ^D обозначает нажатие Control+D.↵
~/cmake_test % cat > CMakeLists.txt↵
cmake_minimum_required(VERSION 3.8 FATAL_ERROR)

project(test_program LANGUAGES CXX)

add_executable(${PROJECT_NAME} test.cpp)

target_compile_features(${PROJECT_NAME}
                        PRIVATE cxx_std_17)
set_target_properties(${PROJECT_NAME} PROPERTIES
                      CXX_EXTENSIONS OFF)
^D
~/cmake_test % # Создадим и перейдём в каталог сборки.↵
~/cmake_test % cd .. && mkdir build-cmake-test && cd build-cmake-test↵
~/build-cmake_test % # Сконфигурируем проект.↵
~/build-cmake_test % # Символ в качестве последнего символа строки экранирует его,↵
~/build-cmake_test % # позволяя продолжить команду на следующей строки.↵
~/build-cmake_test % # Здесь использовано, чтобы уместиться на страницу.↵
~/build-cmake_test % cmake -G Ninja -DCMAKE_CXX_COMPILER=clang++ \↵
~/build-cmake_test %                -DCMAKE_BUILD_TYPE=Debug \↵
~/build-cmake_test %                -DCMAKE_CXX_FLAGS="-pedantic-errors -Wall" \↵
~/build-cmake_test %                -DCMAKE_VERBOSE_MAKEFILE=ON \↵
~/build-cmake_test %                ../cmake-test↵
-- The CXX compiler identification is Clang 6.0.0
-- Check for working CXX compiler: /usr/lib/llvm/6/bin/clang++
-- Check for working CXX compiler: /usr/lib/llvm/6/bin/clang++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/build-cmake-test
~/build-cmake_test % # Результат конфигурации:↵
~/build-cmake_test % ls↵
build.ninja  CMakeCache.txt  CMakeFiles  cmake_install.cmake  rules.ninja
~/build-cmake_test % # *.ninja и есть конфигурация для настоящей системы сборки,.↵
~/build-cmake_test % # также виден кэш CMake.↵
```

```
~/build-cmake_test % # Соберём проект:↵
~/build-cmake_test % cmake --build .↵
[1/2] /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g -std=c++1z -MD
                                  -MT CMakeFiles/test_program.dir/test.cpp.o
                                  -MF CMakeFiles/test_program.dir/test.cpp.o.d
                                  -o CMakeFiles/test_program.dir/test.cpp.o
                                  -c /home/user/cmake-test/test.cpp
[2/2] : && /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g
                                       CMakeFiles/test_program.dir/test.cpp.o
                                       -o test_program && :
~/build-cmake_test % # CMake использует более старое наименование стандарта,↵
~/build-cmake_test % # а также указывает опции для отслеживания неявных зависимостей↵
~/build-cmake_test % # файлов исходного текста от включённых в них заголовочных.↵
~/build-cmake_test % # Образ программы теперь готов:↵
~/build-cmake_test % ls↵
build.ninja  CMakeCache.txt  CMakeFiles  cmake_install.cmake  rules.ninja  test_program
~/build-cmake_test % # Объектный файл в подкаталоге:↵
~/build-cmake_test % ls CMakeFiles/test_program.dir↵
test.cpp.o
~/build-cmake_test % # Повторная сборка ожидаемо ничего не делает - нет изменений.↵
~/build-cmake_test % cmake --build .↵
ninja: no work to do.
~/build-cmake_test % # Удалив только образ программы при сборке будет только компоновка -
~/build-cmake_test % # объектный файл не старее неизменного файла исходного текста:↵
~/build-cmake_test % rm test_program↵
~/build-cmake_test % cmake --build .↵
[1/1] : && /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g
                                       CMakeFiles/test_program.dir/test.cpp.o
                                       -o test_program && :
~/build-cmake_test % # Удалим продукты сборки:↵
~/build-cmake_test % cmake --build . --target clean↵
Cleaning...
Remove CMakeFiles/test_program.dir/test.cpp.o
Remove test_program
2 files.
~/build-cmake_test % # Изменим опции кэша:↵
~/build-cmake_test % cmake -DCMAKE_VERBOSE_MAKEFILE=OFF .↵
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/build-cmake-test
~/build-cmake_test % # Вместо полной конфигурации только применились изменения.↵
~/build-cmake_test % # Теперь при сборке только общая информация, без команд:↵
~/build-cmake_test % cmake --build .↵
[1/2] Building CXX object CMakeFiles/test_program.dir/test.cpp.o
[2/2] Linking CXX executable test_program
~/build-cmake_test % # (на терминале будет видна только последняя строка, т.к.↵
~/build-cmake_test % # успешные затираются последующими.)↵
```

> Если CMake выдаёт не слишком информативные ошибки, изучите файлы `CMakeFiles/*.log` в каталоге сборки — в них содержится дополнительная информация.

4.25. Диагностические сообщения

Из всех инструментальных средств обычно компилятор обладает самым богатым репертуаром диагностических сообщений, которые делятся на следующие группы:

- ***Ошибки (error)*** — нарушения синтаксиса или семантики языка. Не все подобные нарушения диагностируемы, например, неопределённое поведение. Обнаружив ошибку, транслятор пытается восстановиться и продолжить просмотр оставшейся части исходного текста, чтобы показать все имеющиеся в нём ошибки за один вызов. Это не всегда успешно — он может запутаться и начать выдывать множество несуществующих ошибок. При возникновении множества похожих на ложные ошибок, следует исправить первую по порядку настоящую ошибку — возможно, она и есть причина непонимания со стороны транслятора и ложные ошибки пропадут. Кроме того, не всегда ошибки локализуются корректно — указанное

компилятором место и ошибка могут соответствовать другой ошибки в другом месте. Это особенно часто случается при пропуске парных конструкций — скобок, кавычек и др.

- ***Фатальные ошибки (fatal error)*** — ошибки, после которых компилятор не пытается восстановиться, поскольку считает их достаточно серьёзными, чтобы был смысл просматривать входной файл далее. Например, достижение слишком большого числа обычных ошибок становится фатальной.
- ***Предупреждения (warning)*** — сообщения о подозрительных, опасных конструкциях или обнаруженных ошибках, формально не являющихся таковыми по стандарту языка. Это дополнительные диагностические сообщения, которые, хотя не препятствуют успешному завершению трансляции, вероятно являются признаками настоящих ошибок или других проблем. ***Следует выработать привычку не игнорировать предупреждения, даже если программа оттранслировалась, запустилась и «работает»!*** Необходимо выявить их причину, устранить её, а если проблемы нет, объяснить это транслятору — в сомнительных ситуациях он подсказывает, как это сделать. Подобные исправления покажут и ему, и другим программистам, что подозрительные конструкции на самом деле осознанно умышленны, а не случано шибочны.
- ***Заметки (note)*** — дополнительные уточнения к предыдущим диагностическим сообщениям. Стандартный формат диагностические сообщения транслятора содержит имя файла исходного текста, строку и столбец места ошибки, вид сообщения, пояснительный текст, цитату исходного кода и место ошибки, возможно, с предлагаемыми в нём исправлениями. Для некоторых ошибок другие места в программе являются важной информацией о причине их возникновения — они и отображаются в виде дополнительных заметок после основных диагностических сообщений. Например, для ошибки повтороного описания другой сущности в той же области видимости полезно показать первое описание. Сами по себе сообщения такого вида не выдаются.

Приведём примеры этих сообщений:

```
1  int f()
2  {
3      return 0;
```

В этом простом случае сообщение об ошибке напрямую указывает на проблему:

```
/home/user/test.cpp:3:14: error: expected '}'
return 0;
^
/home/user/test.cpp:2:1: note: to match this '{'
{
  ^
  1 error generated.
```

Заметкой в этом примере указано местонахождение открывающей непарной скобки.

```
1  int f()
2  {
3      int x = 5;
4      return 0;
5  }
```

Здесь будет выдано предупреждение:

```
/home/user/test.cpp:3:9: warning: unused variable 'x' [-Wunused-variable]
int x = 5;
^
1 warning generated.
```

Безобидное на первый взгляд предупреждение о неиспользуемом объекте может скрывать за собой реальную ошибку: это просто забытый и более не нужный из-за изменений дальнешего алгоритма объект? Или он всё-таки нужен, а не используется из-за дальнейшей опечатки в алгоритме, являющейся логической ошибкой?

В сообщениях о предупреждениях в конце обычно указана опция, которая привела к выводу этого сообщения, чтобы весь класс таких предупреждений можно было отключить,

если они не интересуют программиста. Мы не указывали такой опции, это предупреждение было включено группой `-Wall`. Отключить его можно, воспользовавшись стандартным видом отрицательных форм опций драйвера компилятора: `-Wno-unused-variable` (и для этого сообщения конкретно, не следует, оно полезно, как было показано выше).

Вот пример более серьёзной ошибки, нередко совершаемой изучавшими до C++ другие языки программистами:

```
1  #include <iostream>
2
3  int f(int x)
4  {
5      if(x=0)
6          std::cout << "x is zero!\n";
7      return x/2;
8  }
```

Судя по выводимому тексту, подразумевалось не присваивание, а проверка на равенство, но вместо `==` программист написал `=`. К несчастью, такое выражение оказалось и синтаксически и семантически корректным: вместо проверки на ноль, в `x` записывается (!) ноль, тут же считывается назад, и, после приведения к `bool` приводит к тому, что контролируемый `if` оператор не выполянется никогда. Тем не менее, компилятор бдителен:

```
/home/user/cmake-test/test.cpp:5:9: warning: using the result of an assignment
as a condition without parentheses [-Wparentheses]
if(x=0)
~^~
/home/user/cmake-test/test.cpp:5:9: note: place parentheses around the
assignment to silence this warning
if(x=0)
^
(  )
/home/user/cmake-test/test.cpp:5:9: note: use '==' to turn this assignment
into an equality comparison
if(x=0)
^
==
1 warning generated.
```

Несколько странно звучащее предупреждение раскрывается своими заметками: транслятор считает подозрительным использование присваивание в качестве последней операции в контролирующем выражении. Если вдруг это именно то, что хотел программист, это можно явно указать лишней незначащей парой круглых скобок вокруг присваивания — это стандартный приём против такого предупреждения для современных компиляторов. Если всё же имелось в виду сравнение на равенство, то один знак «равно» следует заменить на два, как подсказывает последняя заметка.

> Практически невозможно писать нетривиальный код, который не будет выдавать предупреждений с любыми компиляторами, любых версий, для любых их наборов настроек. Многие предупреждения соответствуют совсем тонким, временами чисто стилистическим деталям, к счастью, они не входят в стандартные наборы предупреждений. В то же время к чистоте кода на уровне `-Wall` стремится однозначно следует. Какие ещё сверх него предупреждения действительно полезны, мы будем обсуждать по мере изучения языка.
>
> Некоторые проекты используют опцию `-Werror`, превращающую все предупреждения в ошибки, что не позволяет успешно транслировать программы, их содержащие. В некоторых замкнутых экосистемах это может быть полезно, чтобы заставлять разработчиков исправлять все предупреждения на выбранном уровне, но для использования в переносимом коде, который может собираться другими людьми, это опция не приемлема: стоит выйти новой версии транслятора с новым предупреждением, включённым в стандартную группу `-Wall`, и с ним ваша программа перестанет собираться, хотя хуже не стала. По этой причине автор не рекомендует использование этой опции кроме редких, ограниченных ситуаций. Привычка исправлять осмысленные предупреждения должна вырабатываться независимо от такой настройки компилятора.

Прочие инструментальные средства имеют свои диагностические сообщение, в основном ошибки и предупреждения. При ручном их запуске видно, кто из них что выдаёт. Настрой-

ка `CMAKE_VERBOSE_MAKEFILES=ON` позволяет видеть конкретные команды, которые выдают предупреждения.

Приведём пример ошибки компоновки:

```
1  int Main()
2  {}
```

Это корректная единица трансляции с точки зрения языка C++, однако если она единственная, то в нашей программе нет определения функции `::main`, с которой начинается её выполнение. Сборка с помощью CMake/ninja покажет следующее:

```
~/build-cmake-test % cmake --build .↵
[1/2] /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g -std=c++1z -MD
                                  -MT CMakeFiles/test_program.dir/test.cpp.o
                                  -MF CMakeFiles/test_program.dir/test.cpp.o.d
                                  -o CMakeFiles/test_program.dir/test.cpp.o
                                  -c /home/user/cmake-test/test.cpp
[2/2] : && /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g
                                       CMakeFiles/test_program.dir/test.cpp.o
                                       -o test_program && :
FAILED: test_program
: && /usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall -g
                                 CMakeFiles/test_program.dir/test.cpp.o
                                 -o test_program && :
/usr/bin/ld.lld: error: undefined symbol: main
>>> referenced by /usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/../../../../lib64
/crt1.o:(_start)
/usr/bin/ld.lld: error: symbol 'main' defined in /usr/lib/gcc/x86_64-pc-linux-gnu
/7.2.0/../../../../lib64/crt1.o has no type
clang-6.0: error: linker command failed with exit code 1 (use -v to see invocation)
ninja: build stopped: subcommand failed.
```

Здесь наблюдается следующее:

- Первая команда — компиляция — успешна.
- Далее вызывается компоновка, и в ней возникают ошибки. Первая — неопределённый символ `main`, что говорит о том, что требуемое определение функции `main` не найдено. Выполнение программы начинается с `main` только формально по стандарту C++ (и наших текущих знаний о нём), на самом деле вначале управление в созданном ОС процессе получает код запуска, отвечающий за инициализацию стандартной библиотеки, а уже он вызывает `main`. Ему и требуется не найденное определение. (Термин «символ» относится к структуре объектных файлов и будет рассмотрен позднее.)
- Вторая ошибка фантомна и является следствием первой. Это техническая особенность используемого компоновщика, она пропадёт вместе с первой, когда она будет исправлена.

На двух последних строчках видно, что о проблеме компоновщика рапортует драйвер компилятора, вызывавшей его, а за ним ninja, обнаруживший ошибку вызова драйвера компилятора.

Наконец, ошибки может выдавать сам CMake. Попробуем указать дополнительные опции компиляции, которые используемому компилятору не известны, при первончальной конфигурации каталога сборки, когда CMake сам вызывает компилятор для проверки его возможностей:

```
~/build-cmake-test % cmake -G Ninja -DCMAKE_CXX_COMPILER=clang++ \↵
~/build-cmake-test %                -DCMAKE_BUILD_TYPE=Debug \↵
~/build-cmake-test %                -DCMAKE_CXX_FLAGS=--bogus-flag \↵
~/build-cmake-test %                -DCMAKE_VERBOSE_MAKEFILE=ON \↵
~/build-cmake-test %                ../cmake-test↵
-- The CXX compiler identification is Clang 6.0.0
-- Check for working CXX compiler: /usr/lib/llvm/6/bin/clang++
-- Check for working CXX compiler: /usr/lib/llvm/6/bin/clang++ -- broken
CMake Error at /usr/share/cmake/Modules/CMakeTestCXXCompiler.cmake:44 (message):
The C++ compiler "/usr/lib/llvm/6/bin/clang++" is not able to compile a
simple test program.
It fails with the following output:
Change Dir: /home/user/build-cmake-test/CMakeFiles/CMakeTmp
```

```
Run Build Command:"/usr/bin/ninja" "cmTC_5de6a"
[1/2] Building CXX object CMakeFiles/cmTC_5de6a.dir/testCXXCompiler.cxx.o
FAILED: CMakeFiles/cmTC_5de6a.dir/testCXXCompiler.cxx.o
/usr/lib/llvm/6/bin/clang++ -pedantic-errors -Wall --bogus-flag -o
CMakeFiles/cmTC_5de6a.dir/testCXXCompiler.cxx.o -c testCXXCompiler.cxx
clang-6.0: error: unsupported option '--bogus-flag'
ninja: build stopped: subcommand failed.
CMake will not be able to correctly generate this project.
Call Stack (most recent call first):
CMakeLists.txt:3 (project)
-- Configuring incomplete, errors occurred!
See also "/home/user/build-cmake-test/CMakeFiles/CMakeOutput.log".
See also "/home/user/build-cmake-test/CMakeFiles/CMakeError.log".
```

В данном случае CMake представляется в сообщении об ошибке и приводит исчерпывающую информацию о проблеме.

4.26. Статический анализатор clang-tidy

Компилятор выдаёт ошибки не столько потому, что стандарт предписывает это, сколько по тому, что концептуально не способен завершить трансляции успешно при наличии синтаксических или семантических ошибок в программе. А вот предупреждения он выдывать не обязан, их наличие — результат того, что они могут быть легко распознаны по ходу основной работы транслятора — трансляции. На отдельные дополнительные проверки транслятор тратить время не будет, поскольку это не его прямая обязанность:

```
1  // Безопасное деление: результат деления на ноль определён и равен нулю.
2  int safe_divide(int x,int y)
3  {
4      if(y==0)
5          return x/y;
6      return 0;
7  }
```

Перед нами пример классической невнимательности: контролирующее условие неверно с точностью до наоборот, вероятно, это результат изменения структуры программы, где раньше, например, вначале проверяли на ноль, и возвращали его без деления. Обычный компилятор, даже при использовании `-Wall` и других опций предупреждений не выдаёт. Однако можно доказать формально, что строка 5 не может быть частью корректного алгоритма: она либо не выполняется, либо гарантировано вызывает неопределённое поведение.

Инструментальное средство, которое производит максимально детальный анализ исходного текста программа на предмет наличия ошибок, называется ***статическим анализатором (static analyzer)***. Поскольку такому анализатору нужно разбираться в структуре программы на C++ не хуже самого транслятора, он часто является его частью либо дополнением. В компилятор clang такой входит, и называется clang static analyzer. Мы рассмотрим более мощное средство: clang-tidy. Также построенное на базе инструментария LLVM, оно создавалось изначально для особого вида проверок, но впоследствие научилось интегрироваться и с самим clang static analyzer, теперь предоставляя надмножество доступных ему возможностей.

Для корректного разбора текста программы, ему необходимо знать все опции, с которыми система сборки собирается вызывать настоящий транслятор. Эту информацию CMake может предоставить, если установить переменную кэша `CMAKE_EXPORT_COMPILE_COMMANDS` в истину. После пересборки проекта в каталоге сборки появится новый файл `compile_commands.json`. Это текстовый файл в формате JSON, который содержит полные командные строки, предназначенные для выполнения системой сборки.

Также как при настройке предупреждений компилятора, у clang-tidy есть множество опций по выбору проводимых проверок, которые он считывает из файла с именем `.clang-tidy` из каталога проверяемого файла или ближайшего надкаталога. Это текстовый файл в формате YAML, формат которог описан в документации. Автор предлагает настраивать ваши проекты размещением в их корне файла следующего содержания:

```
---
Checks: >
  boost-*,bugprone-*,cert-*,cppcoreguidelines-interfaces-global-init,
```

```
  cppcoreguidelines-no-malloc,cppcoreguidelines-pro-type-cstyle-cast,
  cppcoreguidelines-slicing,google-default-arguments,
  google-global-names-in-header,google-runtime-int,llvm-include-order,
  misc-*,modernize-*,performance-*,readability-avoid-const-params-in-decls,
  readability-container-size-empty,readability-*,
  -readability-braces-around-statements,-readability-function-size,
  -readability-identifier-naming,-readability-implicit-bool-conversion,
  -readability-named-parameter
AnalyzeTemporaryDtors: true
HeaderFilterRegex: '.*'
CheckOptions:
- key:   cppcoreguidelines-no-malloc.Allocations
value: '::malloc;::calloc;::posix_memalign;::_aligned_malloc;::std::aligned_alloc'
- key:   misc-assert-side-effect.AssertMacros
value: 'assert,Q_ASSERT'
- key:   modernize-use-auto.RemoveStars
value: '1'
- key:   modernize-use-default-member-init.UseAssignment
value: '1'
- key:   modernize-use-emplace.ContainersWithPushBack
value: '::std::vector;::std::list;::std::deque'
- key:   modernize-use-emplace.SmartPointers
value: '::std::shared_ptr;::std::unique_ptr;::std::weak_ptr'
- key:   modernize-use-nullptr.NullMacros
value: 'NULL;Q_NULLPTR'
- key:   performance-faster-string-find.StringLikeClasses
value: 'std::basic_string;std::basic_string_view'
- key:   readability-simplify-boolean-expr.ChainedConditionalAssignment
value: '1'
- key:   readability-simplify-boolean-expr.ChainedConditionalReturn
value: '1'
...
```

При вызове clang-tidy потребуется указать опцией `-p` имя каталога сборки с этим файлом, а также указать имена проверяемых файлов исходного текста. Чтобы автоматизировать процесс указания этих имён для крупных проектов, можно последовательность требуемых команд записать в файл-***сценарий (script)*** оболочки:

```
~/build-cmake-test % # Записать в каталог исходных файлов сценарий:↵
~/build-cmake-test % cat > ../cmake-test/run-static-analysis↵
#!/bin/bash
# Первая строка файла, начинающаяся с #! указывает ОС,
# какую программу вызвать при попытке запустить такой файл.
# В нашем случае - оболочку.

# Вычислим каталог исходных файлов, взяв каталог, в котором
# расположен этот файл сценария.
SOURCE_DIR="$(dirname $BASH_SOURCE[0])"

# Этот сценарий нужно запускать из каталога сборки
# с файлом compile_commands.json. Если его нет,
# Выведем сообщение об ошибке и завершимся.
if [ ! -f compile_commands.json ] ; then
  echo "This command must be run from a directory containing compile_commands.json!" 1>&2
  exit 1
fi

# Найти все файлы в дереве каталогов с корнем в каталоге исходных
# файлов с раширением .cpp, и передать их команде, вызывающей указанную
# с дополнительными аргументами, считываемыми из стандартного ввода.
find "$SOURCE_DIR" -name '*.cpp' | xargs clang-tidy -p .
^D
~/build-cmake-test % # Запустими статический анализ:↵
~/build-cmake-test % ../cmake-test/run-static-analysis↵
1 warning generated.
/home/user/cmake-test/test.cpp:5:17: warning: Division by zero
```

```
                                                    [clang-analyzer-core.DivideZero]
return x/y;
^
/home/user/cmake-test/test.cpp:4:8: note: Assuming 'y' is equal to 0
if(y==0)
^
/home/user/cmake-test/test.cpp:4:5: note: Taking true branch
if(y==0)
^
/home/user/cmake-test/test.cpp:5:17: note: Division by zero
return x/y;
^
```

Как видно, статическому анализатору подобное доказательство по силам. К его сообщениям, как и прочим предупреждениям, следует относится серьёзно, хотя он даже более, чем компилятор, подвержен ложным срабатываниям. И, разумеется, обнаруживает он далеко не все возможные проблемы. Однако, применённый системно вместе с остальными инструментальными средствами он значительно облегчает написание и отладку программ на C++ — автор настоятельно рекомендует взять его использование в привычку!

4.27. 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 инструментальным средством, с которым программист обычно проводит большую часть времени — интегрированной средой разработки.

Как было рассмотрено ранее, интегрированные среды разработки являются связующим элементом рабочего места программиста. 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.

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. Автор считает использовании русскоязычного инструмента программирования неверным выбором и будет приводить имена интерфейсных элементов только на английском языке.

Меню File содержит команды по созданию, открытию, сохранению и закрытию проектов. Для создания проектов наших первых программ следует воспользоваться командой File → New File or Project... → Non-Qt Project — Plain C++ Project. Хотя Qt Creator позволяет работать с несколькими проектами сразу, это может приводить к путанице — рекомендуется перед открытием или созданием нового проекта закрыть всё, что связано с прошлым командой File → Close All Projects and Editors.

На первой странице мастера потребуется задать имя и расположение проекта. 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.

На второй странице осуществляется выбор системы сборки для нового проекта. Вместо qmake по умолчанию необходимо выбрать CMake.

На следующей странице необходимо выбрать комплект инструментальных средств, которые будут использоваться при разработке программы. Правильная настройка этих комплектов осуществляется в Tools → Options... — C++ и зависит от окружающей среды. Будем исходить из того что настройка уже выполнена вручную или автоматически. При этом 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 изменений 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 Window.

Представление Projects отображает основные файлы, входящие в текущий проект в виде дерева и известные системе сборки. Простой проект на языке C++ изначально состоит из двух файлов: файла проекта `CMakeLists.txt` и файла исходного текста первой и пока единственной единицы трансляции `main.cpp`. Файл проекта отличается от рассмотренного нами, первым делом следует заменить его содержимое требуемым. Также воздержитесь пока от использования предложенной «рыбы» в файле `main.cpp`, содержащей неизвестное нам описание `using namespace std;`, пока нам не станет ясен его смысл. Также был создан файл `CMakeLists.txt.user`, который содержит настройки самого Qt Creator, относящиеся к проекту, в дереве Projects он не отображается. При желании этот текстовый файл в формате XML можно просмотреть и отредактировать.

В левом нижнем углу интерфейса среды находятся три кнопки, отвечающие за сборку и запуск программы: Run, Start Debugging и Build Project (для них имеются аналогичные команды в меню Build). Последняя запускает систему сборки, если при этом возникнут проблемы, они будут отображены в окне Issues. Информация в этом окне не всегда полная, автор считает более надёжным пользоваться окнами Compile Output (сообщения от процесса сборки) и General Messages (туда попадают ошибки CMake). Почти все ошибки привязаны к конкретному месту в тексте программы, к которому можно перейти по двойному щелчку на тексте её описания в этих окнах. Если после внесения изменения и попытки сборки программы вы получили сообщение о том, что имеются не сохранённые файлы, изменения в которых не будут учтены при сборке, нужно поставить в этом окне 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 Run выполняет сборку проекта, и, если она успешна, запускает готовую программу.

В режиме Projects на вкладке Projects можно изменить значения переменных кэша CMake — здесь можно указать значения `CMAKE_CXX_FLAGS`, `CMAKE_VERBOSE_MAKEFILE` и `CMAKE_EXPORT_COMPILE_COMMANDS`, требуемые нам. Значение `CMAKE_CXX_COMPILER` уже подставлено верно исходя из используемого набора инструментов. Значение `CMAKE_BUILD_TYPE` изменяется переключателем над кнопкой `Run` — Qt Creator создаст отдельные каталоги сборки для различных конфигураций. Не забудьте все требуемые значения переменных кэша задать для всех используемых конфигураций.

На вкладке Run в этом режиме задаются аргументы программы и её начальный текущий каталог. При включённой опции Run in terminal запуск программы будет открывать новый терминал, после завершения работы программы он останется для прочтения итогового вывода с приглашением закрыть окно. 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. При отключённом режиме Run in Terminal терминал не открывается. Стандартный вывод программы попадает в окно Application Output, а стандартный ввод невозможен (перенаправляется из пустого файла).

Также имеет смысл воспользоваться следующими возможностями:

- В Tools → Options... — Text Editor можно изменить шрифт и цветовую схему текстового редактора на желаемые. Для быстрого изменения масштаба отображения достаточно нажать `Control+Плюс`/`Control+Минус`.
- Там же на вкладке C++ можно настроить стиль автоматических отступов, осуществляемых по ходу ввода текста программы. Если вы хотите отформатировать часть или весь файл в некотором стиле, настройте clang-format — внешнюю программу, выполняющую эту задачу в разделе Tools → Options... — Beautifier — Clang Format и вызовите её из Tools → Beautifier → Clang Format. clang-format поддерживает несколько встроенных стилей и язык

конфигурации произвольных.

- clang static analyzer вызывается одноимённым пунктом из меню Analyze (настройки в Tools → Options... — Analyzer — Clang Static Analyzer). clang-tidy пока не интегрирован в Qt Creator, но может быть вызван из консоли обычным образом.

В случаях, когда те или иные проблемы системы сборки не удаётся разрешить обычным образом, можно попробовать переконфигурировать её (Build → Run CMake) или пересобрать проект начисто (Build → Rebuild All). Если это не помогает, можно, закрыв Qt Creator, отредактировать файл кэша вручную, или просто уничтожить каталог сборки и файл `CMakeFiles.txt.user` и начать сначала. При открытии файла проекта CMake без парного файла с расширением `.user`, Qt Creator предлагает либо создать требуемые конфигурации, либо указать существующие каталоги сборки для них. При этом следует использовать существующие наборы инструментов вместо предлагаемых импортируемых, чтобы избежать размножения этих временных наборов настроек. Только выбранные конфигурации будут доступны для выбора, но недостающие можно всегда добавить в режиме настройки проекта обычным образом.

4.27.1. Использование интегрированного отладчика

Команда Start Debugging аналогична команде Run, но собранная программа запускается не отдельно от среды разработки, а под управлением интегрированного отладчика. В этом случае происходит автоматическое переключение в режим Debug. Большинство рассматриваемых далее команд расположены в одноимённом меню и продублированы на разделяющей окно по вертикали в этом режиме полосе с информацией.

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 Start Debugging заменяется на 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 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 Step Over выполняет одну строку программы независимо от её содержания. Команда 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 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 Interrupt, которая заменяет команды Start Debugging и Continue, 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 Step Out.

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 Step Over, 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 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 Window → Views представление Registers для отображения регистров процессора.

При работе программы под отладчиком, вы можете принудительно прервать её выполнение в любой момент командой 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 ***точки останова по данным (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.

В любом случае, даже при использовании отладчика, не стоит забывать о самом древнем методе отладки не требующем никаких инструментальных средств: ***отладочной печати*** — добавлении в текст программы дополнительных операций вывода, позволяющих отслеживать процесс её выполнения и выводящих те или иные значения объектов для контроля их правильности.

4.28. Undefined Behavior Sanitizer (UBSAN)

Многие проблемы могут быть выявлены автоматически ещё до того, как придётся брать в руки отладчик, хотя они и возникают во время выполнения программы, и потому не доступны для динамического анализа.

В данном разделе мы познакомимся с первым средством динамического анализа — Undefined Behavior Sanitizer (UBSAN). Он входит в группу средств инструментации кода, интегрированных в компиляторы clang/gcc. Его задачей является добавление в машинный код инструкций, проверяющих выполняемые действия на различные случаи неопределённого поведения, и вывод диагностических сообщений о их возникновении. Для его активации используются следующие опции драйвера компилятора:

- `-fsanitize=undefined` — включает сам анализатор. При раздельной компиляции и компоновке, эту опцию следует указывать в обе фазы: для компилятора, чтобы он генерировал необходимые проверки, и для компоновщика, чтобы включил вспомогательную библиотеку,

которая необходима для работы инструментированного кода.

- `-fno-sanitize-recover=all` — требует немедленного завершения работы программы при обнаружении ошибки. По умолчанию программа может пытаться продолжить работу, что бессмысленно. Это опция указывается в фазу компиляции.

При использовании данных средств для вывода диагностических сообщений с указанием мест в исходном коде на C++ требуется компиляция с отладочной информацией.

При использовании CMake следует указать обе эти опции в переменной кэша `CMAKE_CXX_FLAGS_DEBUG`: это дополнительные опции компилятора только для конфигурации сборки Debug (где отладочная информация присутствует). Использование средств инструментации кода многократно замедляет трансляцию программы, поэтому в итоговых сборках обычно не применяется. Указывать его в опциях компоновки не требуется — CMake дублирует туда опции компиляции автоматически.

Рассмотрим программу:

```
1  #include <cstdint>
2  #include <iostream>
3
4  int main()
5  {
6      std::int32_t a,b;
7      std::cout << "Input two signed integers: ";
8      if(!(std::cin>>a>>b)){
9          std::cerr << "Input failed!\n";
10         return 1;
11     }
12     std::cout << a << " + " << b << " = " << a+b << '\n';
13 }
```

Нетрудно вызвать в ней неопределённое поведение, которое зависит от пользовательского ввода и потому не обнаружимо средствами статического анализа (предполагая, что на большинстве платформ `std::int32_t` не уже `int` и целочисленным повышениям не подвержен):

```
Input two signed integers: 1000000000 2000000000↵
1000000000 + 2000000000 = -1294967296
```

Однако при запуске инструментированного UBSAN кода получим следующее:

```
Input two signed integers: 1000000000 2000000000↵
sum.cpp:12:47: runtime error: signed integer overflow: 1000000000 + 2000000000
 cannot be represented in type 'int'
```

Таким образом, при надлежащем наборе тестов, поиск неопределённого поведения значительно упрощается. Отметим, что подобные средства инструментации не способны найти 100% неопределённого поведения в программах, даже полностью покрытых тестами, но всё равно оказывают колоссальную помощь в отладке. По этой причине автор настоятельно рекомендует их постоянное использование.

4.29. Задачи на структурное программирование

В этих задачах обратите внимание на корректное использование минимальных языковых средств:

- Использование предложенного вида конфигурации системы сборки.
- Включение всех требуемых заголовочных файлов стандартной библиотеки C++ и использование имён из её пространства.
- Проверка пользовательского ввода на успешность и соответствие условиям задачи.
- Выбор достаточных, но не чрезмерно, и эффективных типов данных исходя из условий задачи.
- Определение вспомогательных функций приветствуется в задачах, где их использование оправдано.

- Не используйте часто встречающиеся в примерах в других источниках средства такие как: манипулятор `std::endl`, директиву `using namespace`, константы наподобие `M_PI` и операторы, останавливающие выполнение программы в её конце, чтобы не закрывался терминал. Эти средства нам неизвестны и пока не нужны (первые два), не входят в стандарт языка (третье) или не требуются при использовании надлежащих инструментальных средств (последнее).

Также внимательно разберите задачи на наличие в них крайних случаев, которые требуют особой обработки.

Не оговоренные в условии задачи детали реализуйте по своему выбору.

0. Запросите у пользователя целое число $n, n \in [2; 1000]$ 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 (и в этом тексте) не является квадратным, даже если математически всё сделано верно, круг будет казаться сжатым по горизонтали на экране (как в этом тексте), это нормально. Поскольку такое представление неточно по своей природе, за точностью гнаться нет смысла. На каждую строку достаточно одного вычисления квадратного корня.

1. Запросите у пользователя целые числа w, h и n ($w, h \in [10; 2^{10}]; n \in [1; 2^{10}]$). Выведите n полных периодов графиков синуса и косинуса таким образом, что ось x направлена вниз, ось y — вправо, а начало координат находится посередине верхнего края прямоугольника из $w \times (h \cdot n)$ символов, в котором осуществляется вывод. Линию синуса рисуйте буквой `S`, а косинуса — `C`. Линии должны состоять из смежных символов, без пропусков по столбцам и строкам. Пример для $w = h = 20$, $n = 1$:

```
          S         C
           SSS     C
              SS  C
                CCCS
              CC    S
           CCC       S
       CCCC          S
    CC               S
 CCC               SSS
C                 SS
C           SSS
C     SSSS
 C  SS
 SCCC
S    CC
S      CCC
S         CCCC
 S            CC
  SSS           CCC
     SS            C
```

2. Введите два непробельных символа. Выведите таблицу символов с кодами между заданными символами (включая концы) в указанном ниже формате, пример для символов 'A' (код 65) и 'H' (код 72):

```
COD|0123456789
---+----------
 60|.....ABCDE
 70|FGH.......
```

Рассматривайте беззнаковое представление кодов.

3. Введите два целых неотрицательных числа в интервале $[0; 2^{32})$. Выведите запись перемножения этих чисел в столбик, например:

```
  123
*  45
-----
+ 615
 492
-----
 5535
```

4. На пустую доску для сказочных шахмат размером $n \times n, n \in [1; 100]$ ставят фигуру по координатам (x, y) (отсчёт координат начинается с 1). Эта фигура является «всадником» типа (da, db) $(da, db \geqslant 0)$: может перемещаться инкрементами данного вектора в клетках во всех возможных комбинациях направлений, прыгая через занятые клетки. Например, «ночной всадник», делающий неограниченное число ходов, аналогичных обычному коню, является всадником (2,1). На поле 8×8 ночной всадник, стоящий на $(3, 2)$, бьёт клетки $(1, 1)$, $(1, 3)$, $(1, 6)$, $(2, 4)$, $(4, 4)$, $(5, 1)$, $(5, 3)$, $(5, 6)$, $(6, 8)$ и $(7, 4)$. Выведите координаты всех полей, находящихся под боем заданной фигуры.

5. Запросите у пользователя целое число $n, n \in [0; 2^{64})$. Выведите значение n во всех позиционных системах счисления с основаниями 2–10 в порядке little-endian. Например, на ввод числа 100 следует вывести значения `0010011`, `10201`, `0121`, `004`, `442`, `202`, `441`, `121` и `001`.

6. Запросите у пользователя целое число $b, b \in [2; 10)$, после чего введите вещественное число в системе счисления с основанием b и выведите его приближённое значение в десятичной системе счисления. При вводе пользователь может использовать знак числа перед ним, цифры выбранной системы счисления, точку для отделения дробной части и букву `e` в любом регистре для последующего ввода экспоненты со знаком (экспонента задаёт показатель степени при множителе основания системы счисления и вводится всегда как целое число в десятичной системе счисления). Например, последовательность символов `-22.11e+1` при выборе троичной системы соответствует числу $-25.(3)$ в десятичной.

7. Запросите у пользователя целое число $n, n \geqslant 0$, вещественные число x и коэффициенты $a_0 \dots a_n$ и выведите в заданной точке значение многочлена $P(x) = \sum_{i=0}^{n} a_i x^i$ и его производной. Вычислить значение можно сделать за n умножений и столько же сложений, аналогично для производной.

8. 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 и общую длину сгибов.

9. 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 вещественные 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: $(x, y)^2 = (x^2 - y^2, 2xy)$.)

10. Программа отслеживает перемещение точки с вещественными координатами на плоскости. Точка начинает перемещение из начала координат. На каждом шаге после вывода её текущего положения, программа запрашивает направление движения в виде цифры 1–9 кроме 5, соответствующей одному из 8 направлений с шагом 45° (по положению соответствующих клавиш на цифровой клавиатуре) и пройденное в нём расстояние. Завершать работу программы при удалении от начала координат более, чем на 100.

11. Пользователь вводит последовательность пробельных символов, круглых скобок и цифр. Между двумя скобками находится максимум одна цифра, все скобки парные и вложены без пересечений. Вывести цифру внутри самой вложенной пары скобок (первой такой пары, если их несколько).

12. Вводите, пока вводится, последовательность целых чисел в интервале $[2; 2^{60}]$. После каждого ввода выводите наибольший общий делитель всех введённых чисел. При достижении тривиального случая программу можно завершить досрочно.

13. Введите 32-битные беззнаковые числа a, b и p. Вычислите и выведите $a^b \pmod p$. Это можно сделать без привлечения библиотек работы с числами произвольной длины и не более, чем за $\lceil \log_2 b \rceil + 1$ шагов пользуясь тем, что:

$$x \times y \pmod p = (x \pmod p) \times (y \pmod p) \pmod p \text{ и}$$
$$x^y = d_0 x \times d_1 x^2 \times d_2 x^4 \times d_3 x^8 \ldots,$$

где d_i — цифры представления y в двоичной системе счисления.

14. Алгебраической нормальной формой (АНФ) или многочленом Жегалкина называется способ задания булевой функции в виде суммы по модулю 2 различных конъюнкций её операндов (включая, возможно, константу 1 как конъюнкцию нуля операндов). Степень нелинейности булевой функции определяется как максимальное количество переменных в членах такого представления. Запросите у пользователя число операндов булевой функции n, $n \in [0; 64]$. Для всех 2^n возможных членов алгебраической нормальной формы, выведите их вид и запросите у пользователя, входит ли соответствующий член в представление. После окончания ввода выведите степень нелинейности. Пример работы с программой:

```
Input number of variables: 2↵
Enter whether these terms are in ANF:
1? 1↵
x0? 1↵
x1? 0↵
x0*x1 1↵
Nonlinearity order is 2.
```

Глава 5

Процедурное программирование на языке C++

Ознакомившись с минимальными возможностями языка для написания первой программы, приступим к дальнейшему изложению языка.

В этом разделе мы рассмотрим дополнительные элементы языка, не выходящие за рамки фундаментальной для языка C++ процедурной парадигмы. Помимо знакомства с некоторыми простыми и упрощающими жизнь конструкциями, нашей задачей будет разобраться в способе написания на языке C++ программ из нескольких единиц трансляции, для чего требуются некоторые не самые прямолинейные средства в силу отсутствия прямой поддержки языком модульной парадигмы программирования. Кроме этого мы рассмотрим полные возможности уже использованных нами средств более тщательно.

5.1. Квалификатор const

В группу спецификаторов типа входят названные по своим первым буквам т.н. ***cv-квалификаторы (cv-qualifiers)***. Квалификаторы типов не могут использоваться сами по себе для задания того или иного типа, а являются модификаторами, применяемыми к другим типам. В данном разделе вы поговорим об одном из них: `const`.

Квалификатор `const` в типе значения или объекта делает его не модифицируемым. Для объектов это означает, что соответствующие им леводопустимые выражения не могут быть использованы в качестве левого операнда простого присваивания и других операций, осуществляющих запись в объект. По правилам языка такие объекты обязательно должны быть инициализированы не по умолчанию — нет смысла в объекте с недетерминированным значением, которое нельзя изменить.

За исключением самой мощной конструкции создания производных типов в C++ — классов, не леводопустимые значения не имеют характеристик, определяемых модифицируемостью, поэтому для них cv-квалификаторы отбрасываются как часть преобразования леводопустимого выражения.

```
1  #include <cmath>
2
3  void f()
4  {
5      // Тип pi - "const double" --- неизменяемый double.
6      const double pi = std::acos(-1);
7      // ОШИБКА: левый операнд не модифицируем.
8      pi = 10;
9      // В операнде умножения преобразование
10     // const double (lvalue) -> double (prvalue).
11     std::cout << "2*pi = " << 2*pi << '\n';
12     // ОШИБКА: инициализация по умолчанию неизменяемого объекта.
13     const double e;
14 }
```

Квалификатор `const` не предоставляет каких-либо дополнительных возможностей, а только отбирает их, но тем самым позволяет избежать случайных изменений значений, по смыслу являющихся константами, и предоставляет дополнительную информацию компилятору в

оптимизациях. Обратите внимание, что термины «не изменяемый» и пока не введённый нами «константный» в C++ принципиально различаются!

5.2. 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.
- Неопределённое поведение.

Таким образом, в константных выражениях запрещены чтения и записи объектов, являющихся сущностями среды выполнения и не доступными в процессе трансляции. С той же точки зрения можно рассматривать и запрет на операцию вызова функции. Ограничение на число шагов обусловлено тем, что это единственный способ запретить бесконечное зацикливание вычисления константного выражения, что приведёт к зависанию транслятора — как доказано в вычислительной математики, задача остановки является не алгоритмизуемой в общем случае.

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.

5.2.1. Спецификатор constexpr для объектов

Современные версии языка C++ содержат средства, позволяющие расширить спектр допустимых вычислений в константных выражениях. Достаточно простые в создании и уничтожении объекты могут быть созданы и уничтожены на этапе трансляции — типы таких объектов называют ***литеральными (literal)***. Все рассмотренные нами на этот момент типы (арифметические и `void`) таковыми являются.

В описании сущности, с которой разрешено работать на этапе трансляции, используется спецификатор описания `constexpr`. Его воздействие на определение объекта таково:

- Тип объекта также считается неявно неизменяемым (`const`);
- Объект должен иметь явный инициализатор, если в нём будет использоваться выражение, оно должно быть константным.
- Снимается ограничение на использование преобразования леводопустимого выражения применительно к этому объекту в константных выражениях.

Логика этих правил следующая: если объект имеет начальное значение, известное на этапе трансляции, и неизменяем, то он бует иметь его всегда, следовательно в любом использовании этого объекта на чтение результат также известен на этапе трансляции.

> Для доступности в контантных выражениях объектов целочисленных типов достаточно просто неизменяемости (`const`), а не обязательно `constexpr`. Это соответствует старому и очень ограниченному определению константных выражений из версий языка до C++2011.

> Автор рекомендует не пользоваться этим соглашением и везде явно писать `constexpr` как более наглядный и общий вариант.

Приведём пример:

```
1  void f(int x)
2  {
3      // Тип pi - const double. Этот объект доступен на этапе трансляции.
4      constexpr double pi = 3.1415926;
5      // Инициализатор constexpr объекта должен быть константным выражением.
6      // Преобразование glvalue->prvalue, соответствующее чтению pi,
7      // допустимо, т.к. pi описан как constexpr.
8      constexpr double pi_squared = pi*pi;
9
10     // ОШИБКА: constexpr-объект должен иметь инициализатор.
11     constexpr int something;
12
13     // ОШИБКА: undefined behavior в инициализаторе,
14     // проблема гарантировано распознаётся на этапе трансляции.
15     constexpr int inf = 1/0;
16
17     // ОШИБКА: чтение x, не описанного как constexpr, недопустимо
18     // в константном выражении, которым должен быть инициализатор
19     // константного объекта.
20     constexpr int x2 = x*2;
21 }
```

В таком виде `constexpr` объекты удобны для задания «констант» и в неформальном смысле.

Основное назначение квалификатора `constexpr` — взятие программистом обязательств по обеспечению вычисления значения выражения на этапе трансляции. В современном языке рекомендуется использовать `constexpr` везде, где он уместен, особенно, если это не требует никаких других изменений, чтобы расширить сферу применимости описываемых сущностей. Это, очевидно, необходимо для контекстов, где именно сам язык требует константности выражений — с первым из них мы познакомимся в следующем разделе.

5.3. Оператор switch

Оператор `switch` имеет синтаксис

`switch` `(` выражение `)` оператор

Выражение в скобках, как и в условном операторе, называется контролирующим выражением, а оператор — телом.

Оператор `switch` осуществляет переход к меткам специального вида, имеющим вид

`case` выражение `:`

Выражения в метках `case` должны быть преобразованными константными выражениями типа контролирующего выражения, при этом в одном теле оператора `switch` не может быть двух меток `case` с одинаковыми значениями.

При выполнения оператора `switch` вычисляется значение контролирующего выражения, после чего выполнение передаётся оператору внутри его тела, помеченного меткой `case` с соответствующим значением. (В подавляющем большинстве случаев тело оператора `switch` является составным оператором.) 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 операторов `switch` — рассматриваются только метки, непосредственно входящие в тело. ***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:
        // ...
        // break; // В последней ветке опционален.
}
```

Приведём практический пример:

```
1  // Вычисление числа дней в месяце.
2  int get_number_of_days_in_month(int month,int year)
3  {
4      switch(month){
5          case 2:
6              return 28+!(year%4)-!(year%100)+!(year%400);
7          case 4:
8          // Помеченный оператор может метить другой помеченный оператор.
9          // Подобное вложение нередко применяется для меток в switch.
10         case 6:
11         case 9:
12         case 11:
13             return 30;
14         default:
15             return 31;
16   }
17 }
```

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 для обозначения места, где управление ***проваливается (falls through)*** сквозь метку в следующий фрагмент, рекомендуется использовать описанный в следующем разделе приём.

5.3.1. Атрибуты. Провал между метками case.

Атрибуты (attribute) задумывались как способ введения стандартного синтаксиса для расширений языка, вводимых различными трансляторами C++. Традиционно, при необходимости добавления новой конструкции вводились новые ключевые слова, её обозначающие, что приводило к потенциальным конфликтам с другими реализациями и имеющимся кодом. Атрибуты ввели в язык как единый синтаксис, позволяющий добавлять к различным языковым конструкциям произвольную информацию, которую компиляторы, не поддерживающих соответствующих расширений, могут игнорировать.

С этой точки зрения роль атрибутов в основном сводится к конструкциям, не влияющим существенно на семантику программы, т.к. неизвестные транслятору атрибуты игнорируются без возникновения ошибок (как максимум, может быть выдано предупреждение). В этом смысле сам стандарт языка описывает некоторые атрибуты.

Общий синтаксис атрибутов включает информацию об имени атрибутов и его аргументах, заключённых в двойные квадратные скобки. Такая конструкция может опционально применяться во многих конструкциях языка: описаниях и их частях, операторах и др. Точный список помечаемых сущностей и общий вид синтаксиса атрибутов, позволяющий передавать им произвольную информацию, в данный момент нас не интересует.

Вернёмся к проблеме из предыдущего раздела: провал между метками `case` скорее всего является ошибкой программиста, и должен быть диагностирован. Для включения этого предупреждения при использовании `clang` используется опция `-Wimplicit-fallthrough`, не входящая в `-Wall`. В то же время, остаются случаи, где такие конструкции являются умышленными, и требуется способ донесения этой информации до компилятора, чтобы неуместных предупреждений не выдывалось. Для этого стандартом определён атрибут `[[fallthrough]]`, который нужно разместить перед пустым оператором, который оставляют последним в ветке `case`, после которой необходим провал:

```
1  int f(int x)
2  {
3      switch(x){
4          case 1:
5              std::cout << "x is one!\n";
6              // Здесь выдаётся предупреждение, неожиданный провал между
7              // метками case, который в данном случае, действительно, ошибочен.
8          case 2:
9              std::cout << "x is two!\n";
10             break;
11         case 3:
12             std::cout << "x is three!\n";
13             [[fallthrough]]; // Ожидаемый провал, предупреждения нет.
14         case 4:
15             // Ветка для обоих случаев 3 и 4.
16             std::cout << "x is three or four!\n";
17     }
18 }
```

5.4. Статическое время хранения

Некоторые объекты должны существовать значительное время и быть при этом легко доступными. Простейшим примером являются неизменяемые (в том числе, константные) объекты, хранящие те или иные математические константы для удобства их использования в программе по именам. Определение их в теле функции `main` позволит им существовать бóльшую часть времени работы программы, но никак не сделает их видимыми вне этой функции. Поскольку такие константы могут понадобится в любом месте программы, передавать их через параметры другим функциям, очевидно, не выход.

Чтобы определение объекта было видно поиску имён в нескольких функциях, оно должно быть размещено вне их — в области видимости пространства имён. При этом, разумеется, никакого связанного с ним блока нет, память для такого объекта выделяется не на аппаратном стеке, а объект имеет новое для нас ***статическое (static)*** время хранения. По факту видимости таких объектов из любой точки программы по их квалифицированному имени, их также называют ***глобальными (global)*** (а описанные в блоках — ***локальными (local)***). Каждое определение объекта со статическим временем хранения соответствует одному единственному объекту, независимо от времени обращения к нему в программе.

```
1  #include <iostream>
2
3  // Объект, определённый в глобальном пространстве имён
4  // имеет статическое время хранения.
5  constexpr double pi = 3.14159265358979323846264338328O;
6
7  double circle_area(double radius)
8  {
```

```
 9      // Доступ к единственному объекту, определённому выше.
10      return pi*radius*radius;
11  }
12
13  int main()
14  {
15      // Доступ к тому же самому объекту.
16      std::cout << "pi/2 = " << (pi/2) << '\n'
17                << "Unit circle has area " << circe_area(1.) << '\n';
18  }
```

В отличие от рекурсии для объектов с автоматическим временем хранения, случаев, когда одному определению объекта со статическим временем хранения соответствует несколько объектов, нет. Поскольку такие объекты должны быть доступны всё время работы программы, фиксированный объём памяти для них выделяется компилятором как часть объектного файла, и потом становится частью образа программы.

Формально инициализация таких объектов происходит в две стадии:

1. ***Статическая (static)*** инициализация соответствует выяснению данных, которые компилятор размещает в объектном файле в процессе трансляции и производится одним из двух способов.

   (a) Если инициализатор объекта со статическим временем хранения является константным выражением, оно вычисляется во время трансляции и представление результата напрямую записывается в объектный файл — это называют ***константной (constant)*** инициализацией.

   (b) В противном случае (включая случай отсутствия инициализатора в определении объекта) в объектном файле преставление объекта заполняется байтами со всеми нулевыми битами. (Для изученных нами арифметических типов такое представление соответствует значению 0.) Такой вид инициализации называют ***инициализацией нулём (zero initialization)***.

2. ***Динамическая (dynamic)*** инициализация объектов со статическим временем хранения происходит во время выполнения программы для объектов с неконстантным инициализатором. Такой инициализатор вычисляется и устанавливает настоящее начальное значение объекта вместо записанных в его представление транслятором нулей. Стандарт языка предписывает, что динамическая инициализация должна произойти до первого использования объекта в единице трансляции, в которой он определён, хотя точный момент не уточняется. В рамках одной единицы трансляции инициализация глобальных объектов упорядочена в соответствии с порядком их определений.

   Даже если инициализатор объекта не является константным выражением, но компилятор может полностью вычислить представление инициализированного объекта на этапе компиляции, динамическая фаза инициализации может быть заменена статической.

Объекты со статическим временем хранения уничтожаются при завершении работы программы, обычно это значит, что происходит это после возврата из функции `main`. С учётом требований на динамическую инициализацию выше, это означает, что с точки зрения программы объекты со статическим временем хранения существуют всё время выполнения программы.

Из рассмотренного нами видно, что работы программы не ограничивается в точности выполнением функции `main`.

Место, с которого начинается выполнение программы, указывается в образе программы, и обычно соответствует не написанной вами функции `main`, а функции, определённой в специальном объектном файле, поставляемым вместе с компилятором, который драйвер компилятора передаёт компоновщику автоматически. Этот код и вызывает написанную программистом функцию `main`. Данный объектный файл обычно называют crt (C Run-Time) startup.

Перед этим он выполняет действия, необходимые для начала функционирования средств языка и его стандартной библиотеки, в том числе многие реализации именно в этот момент, ещё до вызова `main`, выполняют динамическую фазу инициализации объектов со статическим временем хранения, не откладывая её до первого их использования.

Формально, уничтожение объектов со статическим временем хранения происходит в процессе вызова функции `std::exit`, описанной в `cstdlib` как

```
1  [[noreturn]] void exit(int exit_code);
```

Эта функция завершает выполнение программы, передавая указанное параметром значение операционной системе как код возврата. Код crt startup после завершения инициализации обычно вызвает эту функции с аргументом, соответствующим результату вызова `main`.

Эта функция не возвращает управления в точку вызова. Для пометки функций, которые независимо от описанного типа их возвращаемого значения не возвращают в точку вызова управление штатным образом, имеется атрибут `[[noreturn]]`. Первостепенным в данном случае является документация таких необычных функций, кроме этого это подсказка оптимизатору компилятора (правда, если таким образом помеченная функция всё же вернёт управление, наступит неопределённое поведение).

Функцию `std::exit` может вызвать и самостоятельно из любого места программы для её немедленного завершения. Это может показаться удобным, но делать этого в программах на C++ обычно не следует. Хотя объекты со статическим временем хранения и будут явно уничтожены этой функцией, это не коснётся объектов с автоматическим временем хранения, существующих на момент её вызова. Пока это не представляет для нас проблемы, т.к. память аппаратного стека, где они расположены, будет возвращена операционной системе как часть уничтожения процесса. В будущем, мы познакомимся с объектами, уничтожение которых в состоит не только в освобождении занимаемой ими памяти, но и выполнении произвольного кода, который не будет выполнен в случае вызова `std::exit` из программы, что может привести к проблемам. Для однозначно корректного завершения программы в этом случае следует организовать логика программы для возвращения в функцию `main` и из неё. Пока это придётся делать явно, более удобный способ будет рассмотрен нами позднее.

Объекты со статическим временем хранения многим начинающим программистам кажутся удобным средством, заменяющим параметры функций — вместо того, чтобы использовать их многократно в глубоких цепочках вызова функции, якобы удобнее разместить требуемое многим функциям значение в объекте, который виден и доступен из них всех. Однако такой подход принципиально не приемлем:

- Сложность тестирования реализации алгоритма на императивном языке пропорциональна числу состояний, в котором он может находится. Состояние программы в конкретной точке определяется значениями объектов, доступных из неё. Объекты, описанные в области видимости пространства имён видны и могут использоваться из многих функций, что увеличивают число состояний каждой и них и вносит зависимости между ними, не видные в цепочки вызов одной функции из другой. Значения этих объектов могут читаться и писаться потенциально из любого места программы и оказывать влияние произвольным образом. Так же, как не структурные конструкции передачи управления нарушают структурную парадигму, поддерживающую запись алгоритма в удобочитаемой форме, объекты со статическим временем хранения нарушают модульную парадигму, позволяя любой части программы оказывать влияние на другую, бесконечно запутывая процесс изменения состояний в программе.
- Автоматические объекты создаются по одному на каждый случай выполнения того или иного блока, даже если при рекурсивных вызовах это ведёт к существованию нескольких объектов, соответствующих одному определению. Объект, соответствующий определению со статическим временем хранения, в программе будет всегда только один. Этого может быть достаточно в простых случаях, но концептуально множество уникальных сущностей не так широко, как это кажется. Рассмотрим пример:

  1. Для хранения одной из настроек библиотеки А её автор выбирает объект со статическим временем хранения, потому что он влияет на многие её функции (см. предыдущий недостаток!). Этот параметр кажется уникальным для любого случая применения этой библиотеки, вероятно, потому что библиотека возникла как обобщение случая решения некоторой одной задачи.
  2. Библиотека Б1, написанная другим автором, использует А и её в процессе работы требуется определённое значение рассматриваемой настройки библиотеки А, которое она и устанавливает. Библиотека Б1 использует А только в своей реализации, и многие её пользователи не задумываются о её зависимости от неё.
  3. Библиотека Б2, написанная третьим автором, также использует А в своей реализации, и устанавливает для обеспечения своей работы другое значение рассматриваемой настройки.
  4. Последний программист пишет программу, использующую библиотеки Б1 и Б2. Неожиданно для него, одна из них не работает корректно, т.к. обе из них попытались изменить

настройку библиотеки А на нужную им, но силу имеет только последняя запись. Программист может не подозревать о причастности к проблеме библиотеки А, т.к. в интерфейсах Б1 и Б2, которые использует он, она не упоминается, что только больше его запутывает.

Это весьма часто встречающийся на практике пример того, что «уникальные» на первый взгляд сущности перестают быть таковыми при увеличении масштабов рассматриваемых проблем.

- Имеется нередко встречающийся в реальности класс труднообнаружимых проблем, связанных с динамической инициализацией объектов, использующих в своих инициализаторах другие подобные объекты — об этом чуть позднее.

Таким образом, использовать модифицируемые объекты со статическим временем хранения не рекомендуется. Неизменяемые (включая константные) объекты вполне приемлемы и часто применяются. Как обычно, следует предпочитать `constexpr` вместо `const` по возможности, чтобы устранить возможность наличия динамической фазы инициализации и связанных с ней проблем.

> Чтобы убедиться, что не константный объект инициализируется без динамической фазы, при использовании clang можно пометить описание атрибутом `[[clang::require_constant_initialization]]`. Если указанный инициализатор потребует динамической фазы, помеченное таким образом определение вызовет ошибку компиляции.
>
> ```
> 1 // Атрибут перед описанием относится к нему всему, т.е.
> 2 // ко всем описываемым в нём сущностям.
> 3 [[clang::require_constant_initialization]]
> 4 double x = 1.23, // ошибки нет, 1.23 - константное выражение.
> 5        y = x;    // ОШИБКА: x - не константное выражение.
> ```
>
> Поскольку данный атрибут не входит в стандарт языка, clang использует для него одноимённое ***пространство имён атрибутов (attribute namespace)***. Пространства имён атрибутов — способ, аналогично пространствам имён для описаний, избежать конфликтов в именах атрибутов, вводимых независимыми разработчиками как собственные расширения языка. Пространства имён атрибутов не имеют отношения к обычным пространствам имён — это просто аналогичный синтаксис выделения префиксов (только на один уровень вглубь) в атрибутах.

5.4.1. Спецификатор static для определений объектов в блоках

TODO

5.5. Связанность

TODO

5.6. Анонимные пространства имён

TODO

5.7. Устройство объектных файлов

TODO

5.8. 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:

```
1  // Выводит ABCDEF в одну строку, символов \ и конца строки с точки
2  // зрения транслятора в строковом литерале нет.
3  std::cout << "ABC\
4  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.
- Несколько подряд идущих токенов-строковых литералов склеиваются в один. Эта возможность на практике даже более полезна, чем удаления символов перевода строки экранированием:

```
1  int x = 12, y = 34;
2  // ...
3  std::cout << "x = " << x << "\n"
4               "y = " << y << '\n';
```

  Обратите внимание, что каждая строка в данном примере заканчивается `\n`, что максимально соответствует выводимому на терминал содержимому. Между литералом с первым переводом строки и следующим на строке ниже нет лишней операции — они склеются к основной фазе трансляции и выведутся как один. Также окончательный перевод строки — один символ, для него достаточно символьного литерала.

- 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.

5.8.1. 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. Форма с угловыми скобками ищет текст в определяемой реализацией последовательности мест, включая его из первого, где он будет найден. Вторая форма должна непосредственно 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  // 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.

5.8.2. 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++ практически полностью вытеснен определениями `constexpr` объектов, которые лишены всех недостатков макросов и должны использоваться вместо них кроме исключительных случаев. В языке C++ макросы-объекты следует применять только в рамках работы с самим препроцессором, а именно в условной компиляции, которой посвящён следующий раздел.

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 5.8.2.

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 5.1: 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 на уровне токенов 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 примеров предлагает типичные проблемы, возникающие 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);
9
10  // В такой форме макросы описывают, чтобы не было
11  // проблем с порядоком операций, показанных выше.
12  #define SQR(x) ((x)*(x))
13
14  // После раскрытия макроса получилось ((++a)*(++a)),
15  // и вместо одного инкремента стало два!
16  int b = SQR(++a);
```

Автор надеется, что означенных проблем достаточно, чтобы отговорить читателя от их преждевременного использования. Макросы-функции в своё время помогали в задачах оптимизации, но в современном языке опять же есть средства основного этапа трансляции, свободные от недостатков макросов — о них несколько позднее в этой главе.

5.8.3. 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. 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.

В выражениях директивы `#if` и только в них дополнительно разрешены следующие специальные виды выражений:

- Идентификатор `defined`, за которым следует (опционально, в круглых скобках) идентификатор заменяется при вычислении на 1, если макрос с таким именем на данный момент определён, иначе 0.
- Идентификатор `__has_include`, за которым в круглых скобках следует имя включаемого файла в угловых скобках или кавычках, аналогично синтаксису директивы `#include`, заменяется на 1, если указанный заголовочный файл найден, иначе на 0.

Так как информацией о системе типов препроцессор тоже не обладает, все целочисленные вычисления производятся в типе 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`.

5.9. 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 также упоминали об интерфейсах единиц трансляции в виде заголовочных файлов и необходимости придания нужным только в реализации данной единицы трансляции функциям и объектам со статическим временем хранения внутренней связанности.

Заголовочные файлы используются для описания интерфейса единицы трансляции в языке C++ и использует по традиции расширение `.h` или `.hpp`. Использованные нами заголовочные файлы стандартной библиотеки без расширения являются историческим исключением. 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:

- Включения других заголовочных файлов, в терминах описаний которого описывается интерфейс данного модуля.
- Описания сущностей, которые составляют интерфейс модуля и предоставляют доступ к его функциональности.
- Определения сущностей, которые должны быть именно определены, а не просто описаны для их успешного использования в интерфейсе. Это встраиваемые функции, определения псевдонимов и другие конструкции, не имеющие описаний, не являющихся определениями.
- 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 ***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_HPP
#define CARDS_HPP

// В описании интерфейса требуется тип фиксированной ширины.
#include <cstdint>

namespace cards
{
    // Проверка и классификация номеров банковских карт.

    // Псевдоним типа для хранения номеров карт.
    using card_t = std::uint64_t;

    // 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;
        }
    }

```

```
15      bool is_valid(card_t cn)
16      {
17          if(!number_in_range(cn))
18              return false;
19          // Алгоритм Луна, стандарт ISO/IEC 7812.
20          unsigned sum = 0;
21          for(int i=0;i<16;++i){
22              unsigned digit = cn%10;
23              cn /= 10;
24              if(i%2){
25                  digit *= 2;
26                  if(digit>9)
27                      digit -= 9;
28              }
29              sum += digit;
30          }
31          return !(sum%10);
32      }
33
34      char type(card_t cn)
35      {
36          if(number_in_range(cn)){
37              card_t first2 = cn/100000000000000;
38              if(first2==34||first2==37)
39                  return 'A';
40              if(first2>=51&&first2<=55)
41                  return 'M';
42              if(first2==50||(first2>=56&&first2<=69))
43                  return 'm';
44              if(first2>=40&&first2<=49)
45                  return 'V';
46          }
47          return '\0';
48      }
49  }
```

```
1   // Файл main.cpp
2
3   #include <iostream>
4
5   #include "cards.hpp"
6
7   int main()
8   {
9       std::cout << "Enter card number: ";
10      cards::card_t cn;
11      if(!(std::cin >> cn)){
12          std::cerr << "Failed to read card number!\n";
13          return 1;
14      }
15      if(!cards::is_valid(cn)){
16          std::cout << "Invalid card number!\n";
17          return 2;
18      }
19      switch(cards::type(cn)){
20          case 'A':
21              std::cout << "This is an American Express card.\n";
22              break;
23          case 'M':
24              std::cout << "This is a MasterCard card.\n";
25              break;
26          case 'm':
27              std::cout << "This is a Maestro card.\n";
28              break;
29          case 'V':
30              std::cout << "This is a VISA card.\n";
31              break;
```

```
32          default:
33              std::cout << "This is an unknown card that appears valid.\n";
34              return 3;
35      }
36      return 0;
37  }
```

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 самодостаточны, например, содержат только определения типов и/или встраиваемые функции, в таком случае они могут не иметь парной единицы трансляции — наличие у интерфейса отдельной реализации для пользователей интерфейса значения не имеет. 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.

5.9.1. Программы из нескольких единиц трансляции с использованием CMake

TODO

5.10. Спецификатор inline

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 функции 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 может употребляться спецификатор `inline`, помечающий функцию как ***встраиваемую (inline)***. Такая пометка означает, что программист предпочёл бы в точках вызова этой функции подставить копию её тела, а не вызывать по обычным правилам. Такая подстановка позволяет на уровне исходного кода по прежнему избегать многократного копирования кода, а в машинном коде получить максимальную производительность. 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. В такой роли встраиваемые функции являются современной альтернативе большинству случаев применения макросов-функций, и должны использоваться вместо них по возможности.

Для встраиваемых функций имеют место следующие дополнительные ограничения на их использование:

- Определение встраиваемой функции должно быть доступно в каждой точке вызова, 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 вида связанности, называемого ***слабой (weak)***, 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  namespacec myutils
5  {
6      // Определение встраиваемой функции в заголовочном файле
7      // интерфейса единицы трансляции.
8      inline bool is_odd(int x)
9      {
10         return x%2;
11     }
12 }
13 #endif
```

Данный спецификатор является лишь просьбой компилятору, которую он вправе игнорировать, например, на нулевом уровне оптимизации, или если по имеющейся информации

встраивание в конкретном месте не выгодно — в этом случае генерируется стандартный вызов функции и её отдельное тело со слабой связанностью. Современные компиляторы значительно лучше программиста могут решать, какие функции выгодны к встраиванию и в каком конкретном случае их вызова. С другой стороны, в режиме полных оптимизаций компиляторы встраивают и не помеченные `inline` функции, если посчитают целесообразным. С этой точки зрения спецификатор `inline` в настоящее время обозначает исключительно внешнюю связанность как таковую, несмотря на то, что в языке она формально не закреплена, хоть и используется многими конструкциями.

> Спецификатор `inline` допустим и на описаниях объектов. В этом случае они также получают слабую связанность и требуют определения в точке использования.

5.11. Спецификатор constexpr для функций

Спецификатор `constexpr` также может применяться в описаниях функций, что означает следующее:

- Функция считается встраиваемой (`inline`) по умолчанию.
- Типы параметров и возвращаемого значения должны быть литеральными типами.
- Не должна содержать оператор `goto` и соответствующие помеченные операторы.
- При вызове функции, если все вычисления, которые будут совершены в процессе её вычисления, соответствуют ограничениям константных выражений, соответствующая операция вызова функции также считается константным выражением.
- Все определяемые в функции объекты должны иметь литеральные типы и быть явно инициализированы. Операции доступа (чтение и запись) над такими объектами разрешаются с точки зрения константности выражения по факту их создания в процессе его вычисления.

Таким образом, если в точке вызова `constexpr` функции все аргументы — константные выражения, её тело доступно, и при вычислении потребуются только вычисления и создания объектов, которые могут быть выполнены на этапе трансляции, то и результат будет доступен на нём:

```
1  #include <cstdlib>
2  #include <iostream>
3
4  // int - литеральный тип.
5  // x - параметр функции, который может быть создан
6  // на этапе трансляции в процессе вычисления константного
7  // выражения в которое входит вызов f, инициализация
8  // копированием из аргумента.
9  constexpr int f(int x)
10 {
11     // int - литеральный тип, инициализатор константен,
12     // если константен x.
13     int y = x+2;
14     // Чтение y константно, если y создан в процессе
15     // вычисления константного выражения.
16     if(y>5)
17         // Модификация y константна, если он создан в процессе
18         // вычисления константного выражения.
19         y += 3;
20     if(y<-10)
21         // Никогда не константно, т.к., к сожалению,
22         // все математические функции не являются constexpr.
23         y = std::abs(y);
24     return y;
25 }
26
27 void g()
28 {
29     // Ошибки нет. Вызов f(100) --- константное выражение,
30     // т.к. в процессе вычисления f нет запрещённых в константных
31     // выражениях вычислений.
```

```
32      constexpr int r1 = f(100);
33
34      // ОШИБКА: Вычисление f(-100) содержит вызов std::abs,
35      // который не является константным. Значит и сам вызов f
36      // с такими аргументами не константный и не может использоваться
37      // в инициализаторе константного объекта.
38      constexpr int r2 = f(-100);
39
40      // Ошибки нет, f(-100) --- не константное выражение, но используется
41      // в контексте, где константность не требуется.
42      std::cout << f(-100) << '\n';
43  }
```

Следует отметить, что даже если все условия по константности вызова функции выполнены, но это выражение не используется к контексте, где требуется константность выражения, компилятор вычислять функция на этапе трансляции не обязан. С другой стороны, в результате оптимизаций могут быть вычислены на этапе компиляции значения вызовов и не `constexpr` функций. Таким образом, спецификатор `constexpr` ортогонален квалификатору const: не все неизменяемые объекты — константные, и не все объекты, участвующие в вычислении константных выражений — неизменяемые.

5.12. Дополнительные виды описаний

В этом разделе мы рассмотрим дополнительные формы описаний, не являющиеся частными случаями описаний стандартной формы. Из основная задача — облегчить использование имён в сложных, многоуровневых иерархия пространств имён.

5.12.1. Определения псевдонима пространства имён

TODO

5.12.2. Описание using

TODO

5.12.3. Директива using

TODO

5.13. Аргументы по умолчанию

TODO

5.14. Леводопустимые ссылки

TODO

5.15. Побитовые операции

TODO

5.16. Перечисления

TODO

5.17. Встроенные функции. Проверки на этапе конфигурации системы сборки.

TODO

5.18. Предусловия и постусловия. Контракты.

TODO

5.18.1. Утверждения. Unit-тестирование.

TODO

5.18.2. Анализ покрытия

TODO

Глава 6

Введение в ООП

TODO

6.1. Задачи на простое использование классов

В данной задаче необходимо реализовать класс с указанным интерфейсом (в виде открытых функций-членов и свободных функций) и убедиться в его работоспособности тестированием.

Во всех вариантах следует выполнить следующие требования:

- Все требования предыдущей задачи остаются в силе (проверка пользовательского ввода, статический и динамический анализ и т.д.)
- Указанный класс должен быть классом с инкапсуляцией, содержащим в своём интерфейсе только указанные функции. Члены класса, определяющие его реализацию, можно вводить любые по необходимости.
- Класс должен быть размещён в интерфейсе отдельной единицы трансляции, и реализован ею.
- Проверки предусловий следует выполнить с помощью утверждений.
- Тесты в виде проверки результатов и постусловий с помощью утверждений следует выполнить в основной единице трансляции с функцией `main`. Тестирование функций вывода можно выполнить вручную, вызвав их, и возложив на пользователя тестов обязанность по сверке результата с эталоном. При тестировании результатов в типах с плавающей точкой следует использовать сравнения с учётом возможной погрешности результатов. Что убедиться, что тесты охватывают весь написанный код, следует использовать инструментальные средства оценки покрытия.

0. Реализуйте класс, представляющий элемент кольца $\mathbb{Z}_n$, $n \in [2; 2^{31})$. Элементами этой алгебраической структуры являются числа $0, 1, 2, \ldots, n-1$.

   Требуемый интерфейс:

   - Конструирование: из n и числа, приводимого по этому модулю, значение по умолчанию — 0.
   - Операции смены, сохранения знака, сложения, вычитания, умножения: выполняются как обычные арифметические с приведением результата по модулю. Предусловием бинарных операций является принадлежность обоих операндов одному кольцу.
   - Функция поиска обратного элемента `inverse`. Применяя к хранимому значению элемента a и модулю n расширенный алгоритм Евклида, получим на выходе u и v, такие что $au + nv = \gcd(a, n)$. Если этот НОД — единица, то u — искомый обратный элемент. Иначе обратного не существует, в качестве признака этого факта следует вернуть нулевой элемент.
   - Явное преобразование к `bool`, ложное только для элемента 0.
   - Явное преобразование к целому типу, используемому для хранения вычета в самом классе.
   - Операция вывода элементов в виде целых чисел.

1. Реализуйте класс, представляющий элемент поля $\mathbb{F}_{2^n}$, $n \in [2; 32]$. Элементами этой алгебраической структуры являются многочлены степени меньшей n от произвольной переменной с коэффициентами из $0, 1$. В памяти они могут быть представлены как целые числа, рассматривая биты их представления как коэффициенты многочлена.

   Требуемый интерфейс:

   - Конструирование: из двух чисел, задающих коэффициенты двух многочленов. Первый определяет представление элементов данного поля, и должен быть многочленом степени n, неприводимым над $\mathbb{F}_2$, например, $x^2 + x + 1$ для $n = 2$, $x^4 + x + 1$ для $n = 4$ или $x^20+x^18+x^17+x^13+x^10+x^9+x^7+x^6+x^4+x^2+1$ для $n = 20$ (проверку неприводимости не производить). Второй задаёт конкретный элемент поля, значение по умолчанию 0.
   - Операции сохранения и смены знака. Над данными полями они эквивалентны.
   - Операции сложения и вычитания элементов. Предусловием является идентичность полей и их представлений для операндов. Операции идентичны и соответствуют сложению многочленов, где коэффициенты приводятся по модулю 2.
   - Операция перемножения элементов. Предусловия как у сложения. Реализуется как перемножение многочленов, а затем взятие остатка от деления на многочлен, задающий представление, все операции с коэффициентами по модулю 2.
   - Явное преобразование к `bool`, ложное только для элемента 0.
   - Явное преобразование к целому типу, используемому для хранения многочлена в самом классе, с соответствующим значением.
   - Операция вывода элементов в виде последовательности 0 и 1.

2. Реализуйте класс, хранящий множество элементов, пронумерованных $0, 1, \ldots, n - 1$, $n \in [2; 64]$.

   Требуемый интерфейс:

   - Конструирование: по значению n.
   - Функции `add` и `remove`, добавляющие или убирающие из множества элемент с указанным номером. Добавление существующих и удаление не существующих элементов корректно и игнорируется.
   - Функция `has`, возвращающая принадлежность указанного элемента множеству.
   - Операции `|`, `&`, `-`, `^` и `~`, реализующие объединение, пересечение, разность, симметрическую разность и вычитание из универсального множества соответственно.
   - Явное преобразование к `bool`, ложное только для пустого множества.
   - Явное преобразование к целому типу, используемому для хранения элементов множества в самом классе.
   - Операция вывода элементов в виде последовательности их номеров через запятую.

3. Реализуйте класс, который параллельно применяет к элементам указанного 64-битного значения унарные булевы функции из множества {сохранение, инверсия, константа 0, константа 1}.

   Требуемый интерфейс:

   - Конструирование: выбрана функция "сохранения" для всех 64 элементов.
   - Функция `transform`, задающая указанную функцию для указанного номера бита.
   - Функция `get_transform`, возвращающая текущую назначенную функцию для указанного номера бита.
   - Операция вызова функции, который передаётся преобразуемое по заданным правилам значение. Результат должен быть вычислен за 3 операции.
   - Операция вывода состояния преобразования в виде последовательности 64 букв, обозначающих выбранные преобразования: `PN01`.

4. Реализуйте класс представляющий интервалы на числовой прямой (границы в формате с плавающей точкой) с указанием принадлежности концов. Предусловием операций над интервалами является представимость результата в виде одного интервала (которую можно выразить через другие функции этого интерфейса).

   Требуемый интерфейс:

- Конструирование: по указанным границам a и b и двум логическим значением, указывающим их вхождение в интервал. Предусловие: $a \leqslant b$.
- Акцессоры для возврата границ интервала `left` и `right` и их принадлежности `includes_left` и `includes_right`.
- Явное приведение к `bool`, возвращающее ложь только для пустых интервалов.
- Две перегрузки функции `contains`, проверяющие принадлежность указанной точки или интервала данному.
- Функция `intersects`, проверяющая наличие пересечения данного интервала с указанным.
- Операция сложения интервалов, возвращающая их объединение. Предусловие: интервалы пересекаются или хотя бы один из них пуст.
- Операция вычитания интервала, возвращающая интервал, который входит в левый операнд, но не во второй. Предусловие: второй интервал не входит в первый, или они совпадают хотя бы одной из границ, которая не входит в первый интервал или входит во второй.
- Операция вывода в виде границ в квадратных скобках через запятую.

5. Реализуйте класс, хранящий вещественное значение, измеренное с погрешностью в виде $a \pm e$.

   Требуемый интерфейс:

   - Конструирование: по значениям a и e, значение e по умолчанию — 0. Предусловие: $e \geqslant 0$.
   - Акцессоры `value` и `error` для чтения значений a и e.
   - Все 6 арифметических операций для вещественных чисел, вычисляющие результаты с соответствующими погрешностями.
   - Операция вывода в виде `a+/-e`.

6. Реализуйте класс, хранящий координаты точки на плоскости в полярных координатах (r, ϕ). Инвариант: при $r = 0$ $\phi = 0$, $\phi \in [0; 2\pi)$.

   Требуемый интерфейс:

   - Конструирование: по умолчанию и по заданным значениям r и ϕ.
   - Акцессоры `radius` и `angle` для возврата значений координат.
   - Операция сложения двух точек в векторном смысле.
   - Функция поворота точки вокруг начала координат на заданный угол `rotate`.
   - Свободная функция нахождения расстояния между двумя точками `distance`.
   - Операция вывода в виде `(r,`ϕ`)`.

7. Реализуйте класс, хранящий дату.

   Требуемый интерфейс:

   - Конструирование: по заданным году, месяцу и дню месяца. Предусловие: корректная дата не раньше, чем начало используемой для реализации системы отсчёта.
   - Акцессоры `year`, `month` и `day` для возврата компонент хранимой даты.
   - Операции сложения и вычитания даты с целым числом, возвращающие исходную дату, сдвинутую на указанное число дней в направлении, определяемой знаком.
   - Операция вычитания двух дат, возвращающая знаковый интервал между датами в днях.
   - Функция `day_of_week`, возвращающая день недели как элемент типа-перечисления.
   - Операция вывода в формате `год-месяц-день`.

8. Реализуйте класс, хранящий время суток с точностью до секунды.

   Требуемый интерфейс:

   - Конструирование по указанным часам, минутам и секундам.
   - Акцессоры `hour`, `minute` и `second` для возврата компонент хранимого времени.
   - Операции сложения и вычитания времени с целым числом, возвращающие исходное время, сдвинутое на указанное число секунд в направлении, определяемым знаком. Переход границы суток допускается и учитывается.

- Операция вычитания двух времён суток, возвращающая знаковый временной промежуток в секундах (считая оба времени принадлежными одному дню).
- Операция вывода в формате `часы:минуты:секунды`.

9. Реализуйте класс, представляющий температуру (в формате с плавающей точкой) в одной из систем отсчёта: Цельсия, Кельвина или Фаренгейта.

   Требуемый интерфейс:

   - Конструирование: по значению и системе отсчёта (Цельсия по умолчанию).
   - Акцессоры `value` и `scale` для возврата хранимого значения и системы отсчёта.
   - Функция `convert`, возвращающая новую температуру в указанной системе отсчёта, эквивалентную данной.
   - Операции сложения и вычитания температур, возвращающие новые температуры в системе счисления левого операнда.
   - Операция вывода температуры в виде числа и буквы системы отсчёта.

10. Реализуйте класс, хранящий цвет в виде набора красной, зелёной и синей цветовой составляющих, хранимых как вещественные числа из $[0; 1]$.

    Требуемый интерфейс:

    - Конструирование: по умолчанию (чёрный) и по заданным компонентам.
    - Акцессоры `red`, `green` и `blue` для чтения соответствующих компонент.
    - Операция сложения цветов, возвращающая новый цвет, компоненты которого есть средние арифметические от операндов.
    - Функция `mix`, возвращающая новый цвет — результат смешения данного цвета и указанного в данной пропорции $p \in [0; 1]$.
    - Функция `luma`, возвращающая яркость цвета в виде скалярного произведения вектора (r, g, b) на $(0.299, 0.587, 0.114)$.
    - Операция вывода в виде значений компонент в скобках через запятую.

11. Реализуйте класс регистра сдвига с линейной обратной связью в конфигурации Фибоначчи. Это регистр из n бит,$n \in [2; 64]$, в котором $(n-1)$-ый бит является его выходным значением. Чтобы сгенерировать новое состояние регистра, новый бит вычисляется как сложение по модулю 2 некоторых битов регистра, всегда включающих выходной. После этого все биты сдвигаются на 1 в сторону выходного (который теряется), а младший получает значение нового.

    Требуемый интерфейс:

    - Конструирование: по n, числу, задающему конфигурацию — каждый бит числа определяет, входит ли соответствующий бит регистра в формулу вычисления нового значения — и начальному состоянию (по умолчанию — 0).
    - Функции чтения конфигурации `config`, состояния `state` и сброса состояния в указанное значение `reset`.
    - Операция вызова функции, которая вычисляет новое состояние регистра и возвращает его выходной бит.
    - Операция вывода в виде последовательности 0/1, соответствующих текущему состоянию, под которыми звёздочками отмечены биты, входящие в вычисление нового состояния.

12. Реализуйте класс генератора псевдослучайных 64-битных беззнаковых чисел. Внутренним состоянием такого генератора является пара чисел s_0 и s_1 (они и все остальные вычисления в беззнаковом 64-битном типе с учётом арифметики по модулю 2^{64}). Алгоритм генерации нового значения:

    (a) Вычислить x как результат побитового сложения по модулю 2 s_0 и s_0, сдвинутого на 23 бита влево.

    (b) Положить y как копию значения $s1$.

    (c) Записать y в s_0.

    (d) Записать в $s1$ результат побитового сложения по модулю 2 x, y, результата сдвига x на 17 бит вправо и результата сдвига y на 26 бит вправо.

(e) Определить очередное выходное значение как сумму $s1$ и y.

Требуемый интерфейс:

- Конструирование по начальному состоянию $s0$ и $s1$ (по умолчанию — 0 и 1). Предусловие: $s0$ и $s1$ одновременно не равны 0.
- Функции чтения состояния генератора `s0` и `s1` и сброса состояния генератора в указанное `seed`.
- Операция вызова функции, вычисляющая новое состояние и возвращающая выходной результат этого алгоритма.
- Функция `discard`, пропускающая указанное число выходных элементов.
- Операция вывода состояния генератора в произвольном формате.

13. Реализуйте декодер потока сжатой информации в формате RLE (Run-Length Encoding). На вход подаётся последовательность байтов. Исходные данные закодированы в виде двух типов подпоследовательностей:

    - Байт с установленным старшим битом, за которым следует второй байт содержания. Соответствует более чем двукратному повтору байтов во входной последовательности, при декодировании заменяется на число копий второго байта в количестве, записанным первым без учёта старшего бита. При кодировании выгодно представлять в таком виде повторы из более чем трёх одинаковых байтов подряд.
    - Байт со сброшенным старшим битом, за которым следует последовательность байтов в количестве, равном его значению. Соответствует остальным подпоследовательностям входных данных, при декодировании байт количества опускается, а сами данные выдаются без изменений.

    Например, входная последовательность `01 02 03 04 04 04 04 04 04 04 05 06 07` сжимается в `03 01 02 03 87 04 03 05 06 07`.

    Требуемый интерфейс:

    - Конструирование без параметров.
    - Функция `reset`, сбрасывающая внутреннее состояние в исходное.
    - Функция `push`, передающая очередной байт в декодер. Предусловие: декодер только что сконструирован, сброшен или вызов `pull` вернул ложь.
    - Функция `pull`, которая возвращает наличие очередного выходного байта и, если он имеется, записывает его в дополнительный выходной параметр. Таким образом взаимодействие с декодером осуществляется по схеме: передавать в `push` по одному байту, и после каждого вызывать в цикле `pull`, пока он не вернёт ложь.
    - Операция вывода внутреннего состояния в произвольном формате.

14. Реализуйте класс, моделирующий состояние космического корабля на подлёте к планете. Моделирование рассматривает одномерное перемещение корабля по оси, направленной от центра планеты к кораблю (т.е. при движении к планете скорость отрицательная). Корабль оснащён фотонными двигателями, поэтому их использование не влияет на массу корабля.

    Требуемый интерфейс:

    - Конструирование: по начальным характеристикам: расстояние до планеты, скорость корабля, ускорение свободного падения на рассматриваемой планете.
    - Функции получения текущей скорости `velocity`, положения `distance` корабля и общего прошедшего времени моделирования `time`.
    - Функция моделирования. Рассматривается движение корабля за указанное время при заданном ускорении, вносимом двигателями корабля (по умолчанию — 0). Функция обновляет состояние объекта и возвращает фактически промоделированное время, которое может быть меньше указанного, если в течении этого периода моделирования корабль достиг планеты.
    - Операция вывода текущих характеристик корабля в произвольном формате.

Глава 7

Шаблоны

TODO

7.1. Примеры использования шаблонов

7.1.1. Комплексные числа

TODO: <complex>

Пользовательские литералы

TODO

7.1.2. Генерация значений случайных величин

TODO: <random>

7.1.3. Работа с временем

Глава 8

Массивы

TODO

8.1. Примеры использования массивов

8.1.1. Обобщённое представление последовательностей объектов в памяти

TODO:

8.1.2. Параметры программы

Начиная с нашей первой задачи мы пользовались возвращаемым значением функции `main` в качестве кода возврата нашей программы. Как нам известно из работы с командной строкой, помимо файлов и кода возврата для взаимодействия с программой можно использовать параметры программы — передаваемый ей при запуске набор строк. Для получения к нему доступа в самой программе потребуется воспользоваться другой формой описания функции `main`:

```
1  // Имена и точные формы параметров не имеют значения,
2  // здесь приведены традиционные.
3  int main(int argc,char* argv[]);
```

Первый целочисленный параметр, обычно именуемый `argc`, указывает число аргументов программы, а второй — `argv` — является указателем на начало непрерывной последовательности указателей на нуль-терминированные строки, являющиеся этими аргументами. Эти указатели и строки доступны всё время выполнения программы.

`argc` всегда имеет неотрицательное значение, на практике оно всегда как минимум 1, поскольку нулевым параметром программы является всегда передаваемое программе её имя — это может быть относительный или абсолютный путь к имени файла образа программы или произвольное значение, заданное вызывающей стороной в зависимости от ОС. Таким образом, все указанные пользователем параметры получают индексы от 1 и выше. Приведём пример программы, выводящей все свои аргументы:

```
1  #include <gsl/span>
2
3  #include <iostream>
4
5  int main(int argc,char* argv[])
6  {
7      std::cout << "Program arguments:\n";
8      int i = 0;
9      for(char* arg:gsl::span{argv,argc})
10         std::cout << i++ << ": \"" << arg << "\"\n";
11 }
```

Форма описания `main` без параметров и разобранная в этом разделе являются двумя формами, поддержка которых предписана стандартом языка, также допускаются и другие, зависящие от реализации описания.

8.1.3. Последовательности бит константного размера

TODO: std::bitset

8.2. Задачи на массивы

В данной задаче необходимо реализовать требуемые обобщённые алгоритмы и/или структуры данных и решить поставленную с помощью них задачу. Вспомогательные конструкции следует реализовать в отдельном модуле, вероятно, только из одного заголовочного файла; основное решение можно провести в той же единице трансляции, где расположена `main`.

Для хранения последовательностей следует использовать шаблон класса `std::array`, для представления интервалов элементов в памяти — `gsl::span`, для перебора элементов по возможности цикл `for` для диапазона, для работы со случайными величинами — средства из `<random>`.

В данной задаче интерфейсы могут не быть строго специфицированы и должны быть разработаны вами с учётом ваших знаний. Обратите особое внимание на следование всем известным вам стандартным приёмам (уместная перегрузка операций, именование функций, const-корректность интерфейсов и т.д.). Достаточно реализации минимального самодостаточного интерфейса, пригодного для решения требуемой задачи.

Прочие применимые к данной структуре проекта требования предыдущих задач остаются в силе.

Варианты:

0. Реализуйте шаблон класса, представляющий двумерный массив константной размерности таким образом, что обращение к индексам элементов за пределами интервалов циклически заворачивает на другую сторону той же размерности. Например, для массива 10×20 доступ к элементу $(-2; 31)$ должен соответствовать элементу $(8; 11)$. Для индексов массива и размерностей используйте тип `std::ptrdiff_t`.

   Решите с помощью этого шаблона класса задачу на моделирование 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.

1. Реализуйте шаблон класса, представляющий двумерный массив константной размерности таким образом, что обращение к индексам элементов за пределами интервалов возвращает фиксированное значение, заданное при конструировании экземпляра объекта данного класса.

   Решите с помощью данного шаблона класса задачу генерации псевдослучайного начального пололожения кораблей в игре «Морской бой». При выборе положения корабля после выбора точки достаточно выбрать одно из двух направлений — вниз или вправо — без ограничения общности. Для проверки возможности размещения корабля следует проверить свободность клекток в прямоугольнике $3 \times (2 + n)$, где n — длина корабля.

2. Реализуйте шаблон алгоритма быстрого Фурье-подобного преобразования, применяемого к интервалу элементов размера $N = 2^n$. Помимо самого интервала, который алгоритм преобразует на месте, его параметром является матрица K размера 2×2 из элементов того же типа, что и интервал. Сам алгоритм состоит в следующем: каждую пару элементов a_i и $a_{i+\frac{N}{2}}$, $i \in [0; \frac{N}{2})$ заменить на соответствующие элементы произведения вектора-строки из этих элементов на матрицу K; после этого применить этот алгоритм к обоим половинам исходного интервала, если его размер больше 2. Алгоритм реализовать без рекурсии.

   С помощью этого алгоритма решите задачу преобразования псевдослучайной булевой функции от 5 аргументов в многочлен Жегалкина: нормальную форму в виде сложения по модулю 2 конъюнкций аргументов. Для этого примените к вектору булевых значений,

соответствующему таблице истинности данной функции быстрое преобразование с $K = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, где сложение элементов осуществляется по модулю 2. Для этого потребуется написать вспомогательный класс, неявно преобразуемый в `bool` и обратно, для которого перегружены операции `+` в смысле сложения по модулю 2 и * в смысле конъюнкции. Результирующий вектор показывает, какие из конъюнкций входят в представление функции: если i-ый элемент истинен, в форму входит конъюнкция элементов, соответствующих номерам единичных бит в числе i, для $i = 0$ — константы 1. Выведите исходную таблицу истинности и Алгебраическую Нормальную Форму (АНФ) — другое название многочлена Жегалкина — в виде формулы.

3. Реализуйте шаблон класса большого числа, хранимого в виде последовательности из N слов беззнакового целого типа `T` (по умолчанию — `std::uint32_t`), где каждый элемент — цифра в записи этого числа в позиционной системе счисления с основанием 2^w, w — ширина типа `T`. Предусмотрите конструирование из одного элемента типа `T`, заносимого в младший разряд. Перегрузите для специализаций этого класса операции `+=` и `*=`, а также вывод в поток в шестнадцатеричной форме.

   Вычислите и выведите с помощью этого шаблона класса значение 100!.

4. Формат хранения чисел BCD (Binary-Coded Decimal) является гибридным представлением в системах основания 10 и 16: одному десятичному разряду соответствует ровно 4 бита представления, при этом значения 10–15 не используются. Реализуйте шаблон класса большого числа, хранимого в виде последовательности из N слов беззнакового целого типа `T` (по умолчанию — `std::uint32_t`), где каждый элемент представляет $\frac{w}{4}$ десятичных разрядов, w — ширина типа `T`. Сложение в таком представлении можно осуществлять как обычное двоичное, но после этого в каждый десятичный разряд, если из него осуществился перенос в следующий (можно определить по его значению, ставшему меньше такого в любом из слагаемых), или он получил запрещённое значение от 10 и выше, следует добавить 6.

   С помощью этого шаблона класса введите два числа, не длиннее выделенных под их хранение ресурсов, посчитайте и выведите их сумму.

5. Реализуйте шаблоны алгоритмов рисования вертикальных и горизонтальных линий на двумерной индексируемой матрице символов произвольного типа. Рисование осуществлять символами `|` и `-`, которые затирают любые старые символы, кроме друг друга и `+` — рисование горизонтальной черты поверх вертикальной или любой из них поверх плюса должно приводить к записи последнего.

   С помощью этих алгоритмов реализуйте моделирование движения частицы по участку плоскости размера 50×30, оставляющей за собой след. Частица начинает движение в случайной точке и совершает 20 движений строго вертикально или горизонтально на любое расстояние от 1 до края поверхности в случайном направлении, отличающемся от предыдущего. Результат вывести.

6. Реализуйте шаблона класса стека из элементов указанного типа и максимальной ёмкости.

   С помощью этого шаблона решите задачу определения, является ли вводимая последовательность символов корректной скобочной: все виды скобок (круглые, квадратные и фигурные) парные и вложены без пересечений с максимальной глубиной 128.

7. Реализуйте шаблон класса очереди из элементов указанного типа и максимальной ёмкости. Интерфейс очереди аналогичен стеку — операции `push`, `pop` и `empty`, но очередь является структурой данных по принципу FIFO (First In — First Out), так что первым извлекается элемент, помещённый последним, но ещё не извлечённый. Чтобы не перемещать элементы при операциях над очередью, следует отдельно отслеживать индексы или указатели на начало и конец очереди, которые перемещать по последовательности циклически: доходя до конца перемещать в конец или наоборот.

   С помощью этого шаблона решите задачу на моделирование с системы массового обслуживания в дискретном промежутке времени на интервале $[0; T]$: бармен за столиком, к которому имеется N подходов, может выдавать прохладительный напиток не чаще, чем раз в t единиц времени. Каждую единицу времени к каждому из подходов с вероятностью p подходит новый клиент, образуя отдельную очередь, если там уже есть люди. Бармен отдаёт напитки подходящим сразу, если готов, иначе в момент готовности напитка он выбирает из всех очередей ту, головной клиент в которой ждал дольше всех. Клиент, получивший напиток, уходит, но если он прождал в очереди больше, чем L единиц времени, он зовёт

администратора, и тот увольняет бармена до наступления времени T. Запросить все требуемые параметры у пользователя и смоделировать данную систему с отображением событий прихода и ухода клиентов, идентифицируя их по времени прихода и номера очереди.

8. Реализуйте обобщённый алгоритм генерации псевдослучайных величин, равномерно распределённых в окрестности указанного радиуса от указанного центра. Реализуйте алгоритм в общем виде для типов с плавающей точкой и предусмотрите специализации для специализаций шаблона класса `std::complex`.

   Решите с помощью этого алгоритма следующую задачу: колонии добрых инопланетян живут в различных N-мерных пространствах, располагая свои колонии в начале координат. Враждующие с ними злые пришельцы выбирают случайную точку высадки на расстоянии не более D от них, высаживая n отрядов в окрестности радиуса d от точки высадки. У хороших пришельцев есть орудие массового поражения — «умное излучение», импульс которого распространяющееся от их колонии во все направления сразу, и который можно запрограммировать на интервал расстояний, в течение которого он будет «активным» и уничтожать всё живое. Энергозатраты на такой импульс зависят только от длины активного интервала, который необходимо минимизировать. В первую половину активного периода работы умного излучения, его интенсивность равномерно растёт, а во вторую — падает. Любой уровень излучения в активный период летален, но подвергшиеся более высокому его уровню существа погибают быстрее. Запросите у пользователя все требуемые параметры и смоделируйте оборону колонии добрых пришельцев в пространствах $\mathbb{R}$ и $\mathbb{C}$, выводя все генерируемые и расчётные характеристики. Также найдите и выведите координаты злодея, погибшего с минимальными мучениями.

9. Реализуйте шаблон класса из указанного количества счётчиков. Начальное значение всех счётчиков — 0, класс позволяет увеличивать на 1 значение любого счётчика по указанному индексу, и отслеживает счётчик с максимальным значением, предоставляя доступ к его номеру.

   С помощью этого класса смоделируйте процесс вытаскивания кард из колоды до того момента, когда из вытянутых карт можно будет собрать «флэш» — 5 карт одной масти (джокеры считаются за любую масть). Выведите все вытащенные карты и искомую пятёрку.

10. Реализуйте обобщённый алгоритм вывода гистограммы указанной последовательности. Параметрами алгоритма являются отображаемый интервал значений и высота гистограммы в символах. При выводе каждому значению последовательности соответствует столбик символов высоты от 0 до высоты гистограммы, количество символов пропорционально положению значения внутри диапазона, выходящие за границы значения приравниваются к границам интервала соответственно. Поток для вывода, а также символы для рисования столбиков и пустых мест задаются параметрами функции, значения по умолчанию `#` и пробел.

    Решите с помощью этого задачу моделирования бомбардировки поверхности дна ущелья метеоритами. Глубокое ущелье имеет плоское дно. Поверхность планеты с ущельем попала в метеоритный дождь, падающий строго перпендикулярно поверхности дна. Из-за узкости оврага его можно считать одномерным, рассматривая с точностью до 128 позиций. В ущелье попадает N метеоритов в случайных местах, каждый из который при соприкосновении с планетой взрывается, уничтожая землю в случайном радиусе из интервала $[r; R]$. Запросите значения всех параметров у пользователя и выведите карты высот оврага после бомбардировки.

11. Метрика Левенштейна — мера близости двух последовательностей, равная минимальному числу операций вставки, удаления или замены элемента, которые необходимы для преобразования одной последовательности в другую. Пусть длины последовательностей равны m и

n. Введём рекуррентное соотношение

$$D(i,j) = \begin{cases} 0, & \text{для } i = 0, j = 0 \\ i, & \text{для } i > 0, j = 0 \\ j, & \text{для } i = 0, j > 0 \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{для } i > 0, j > 0 \\ \quad D(i-1,j-1)+m_{ij} & \\ \} & \end{cases}$$

, где m_{ij} равно 1, если i-ый элемент первой последовательности и j-ый элемент второй последовательности не совпадают, иначе 0. Сама метрика Левенштейна есть значение $D(m,n)$ и может быть вычислена полным заполнением таблицы значений функции D, при этом считая меньшую размерность длиной строки такой таблицы (метрика коммутативна), в процессе вычислений достаточно держать в памяти только текущую строку и предыдущую. Реализуйте обобщённый алгоритм поиска значения метрики Левенштейна двух указанных последовательностей объектов в памяти, передавая также доступный на запись интервал значений типа `std::size_t` размером не менее $2(\min m, n+1$ для хранения промежуточных значений.

С помощью этого алгоритма вычислите и выведите расстояние Левенштейна для двух строк, заданных через параметры программы.

12. Реализуйте обобщённый алгоритм, который считывает все доступные символы из указанного потока, и записывает в указанный выходной поток результат применения данного функтора к каждому из них.

    Решите с помощью этого алгоритма задачу преобразования программой символов по заданным правилам замены через стандартные потоки ввода/вывода. Реализуйте требуемый класс функтора для алгоритма выше, который содержит в себе полную таблицу подстановки в виде массива из 256 беззнаковых символов, начальное значение которых равно своим индексам, предусмотрите задание правил замены через этот массив. Правила замены возьмите из параметров программы, где каждый из них должен быть парой символов, указывающих что заменять и на что.

13. Реализуйте шаблон класса, параметризуемого беззнаковым целым типом, который может хранить и позволять задавать перестановки битов внутри таких значений, хранимые внутри как массив новых индексов старых битов в количество ширины обрабатываемого типа. Класс должен являться функтором, реализующим данное преобразование.

    Решите с помощью данного шаблона задачу преобразования программой символов по заданным правилам замены бит через стандартные потоки ввода/вывода. Требуемые замены указываются через параметры программы, каждый из которых состоит из двух цифр 0–7, указывающих, какие два бита нужно поменять местами.

14. Реализуйте шаблон класса для поиска последовательностей значений из алфавита в N символов алгоритмом Bitap не длиннее известного наперёд константного размера. Алгоритм на каждом шаге оперирует последовательностью бит той же длины, что и искомая последовательность, начальное значение которой — все единицы. При чтении очередного элемента последовательности, в которой производится поиск, все элементы сдвигаются на 1 вперёд (последний пропадает, младший принимает значение 0), после чего введённый элемент сравнивается с каждым элементом искомой последовательности, и для каждого несовпадения соответствующий по индексу элемент последовательности заменяется на 1. Если после этого старший бит последовательности — 0, то только что был прочитан последний элемент вхождения последовательности, которая искалась. При поиске последовательности бит в последовательности бит, можно найти различия двух последовательностей их побитовым сложением по модулю 2, и применить операцию побитового или к результату и последовательности отслеживания, для установки в ней единиц в нужных местах. При использовании алфавита большей размерности, можно заранее заготовить по одной битовой маске на каждое возможное значение элемента последовательности, соответствующей результату сравнения каждого элемента искомой с конкретным элементом. Это позволит оставить

число операций на шаг алгоритма равным одному сдвигу, одной выборке элемента массива, одному побитовому ИЛИ и проверки старшего бита на ноль.

С помощью этого алгоритма решите задачу поиска в файле стандартного ввода последовательности полубайт, заданной параметром программы в виде строки в шестнадцатеричной системе (1 знак — 1 элемент) не длиннее 128 элементов. Выведите номера всех смещений в файле, где искомая последовательность нашлась.

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 4.5.

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] ***N1570***, последний черновик языка C11, 12 апреля 2011.
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[2] ***N4700***, Working Draft, Standard for Programming Language C++, 2017-10-16.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4687.pdf

[3] 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

[4] System V Application Binary Interface. AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models). Draft Version 0.99.8
https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-r252.pdf

[5] ***Donald E. Knuth***. Big Omicron and big Omega and big Theta // SIGACT News, Apr.-June 1976, pp. 18-24.

[6] ***Donald E. Knuth***. Sorting and Searching, volume 3 of The Art of Computer Programming // Addison-Wesley, 1973. Second edition, 1998.

[7] ***Henry S. Warren***. Hacker's Delight, 2nd edition // Addison-Wesley Professional, 2012.

[8] ***Кауфман В.Ш.*** Языки программирования. Концепции и принципы. // М. ДМК Пресс, 2010 — 464 с.

[9] ***Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein***. Introduction to Algorithms, Third Edition // The MIT Press, 2009, 1292 p.

[10] ***Нестеренко А.Ю.*** Теоретико-числовые методы в криптографии // Московский государственный институт электроники и математики, 2012, 224 с.

[11] ***Igor Zhirkov*** Low-level Programming. C, Assembly, and Program Execution on Intel® 64 Architecture // apress, 2017.

[12] ***Daniel Kusswurm*** Modern X86 Assembly Language Programming. 32-bit, 64-bit, SSE, and AVX // apress, 2014

[13] ***Guillaume Lazar, Robin Penea*** Mastering Qt 5. // Packt Publishing, 2016

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.

- ***13.04.2018***
  - Добавлены задачи на шаблоны и массивы.
- ***29.01.2018***
  - Добавлен раздел «Визуализация представления программы в процессе трансляции».
  - Раздел «Квалификатор const» перенесён в новую главу «Процедурное программирование на языке C++».
  - Написана часть раздела «Процедурное программирование на языке C++».
  - Уточнены условия задач на простое использование классов.
- ***14.12.2017*** Добавлены задачи на простое использование классов.
- ***07.11.2017***
  - Добавлен материал лекции 02.11.
  - Мелкие исправления и добавления.
- ***29.10.2017***
  - Добавлен материал остальной части октября 2017.
  - Добавлены задачи на структурное программирование.
  - Незначительные исправления и добавления.
- ***08.10.2017***
  - Добавлен материал 05.10.2017.
- ***02.10.2017***
  - Первая версия для нового потока 2017-го года.