В.В. Борисенко

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Интернет-университет информационных технологий
Москва • 2005

УДК 004.4(075.8)
ББК 32.973.26-018я73
Б-82

М **Борисенко В.В.**
Основы программирования : [учеб. пособие] / В. В. Борисенко; Интернет ун-т информ. технологий. - М. : Интернет - ун-т информ. технологий, 2005. - 328 с. - (Основы информатики и математики). - ISBN 5-9556-0039-6.

Книга предназначена для обучения основам программирования. Рассматриваются основные понятия программирования - алгоритма, исполнителя, алгоритмического языка, переменной, основные типы данных, управляющие конструкции алгоритмического языка и т.п. Излагаются общие приемы программирования, основанные на применении математики, такие, как вычисление функций на последовательностях с помощью применения теории индуктивных функций и схема построения цикла с помощью инварианта.
Рассматриваются общие принципы устройства и работы компьютера, типичные команды и регистры процессора, методы адресации, способы вызова функций и передачи параметров и т.п. Приводятся примеры записи программ как на виртуальном Ассемблере RTL, так и на Ассемблере процессора Intel 80386. Кратко рассмотрены аппаратные средства поддержки многозадачности.
Значительная часть книги посвящена основам языка Си. Помимо основ языка, в ней приведено много примеров реализации алгоритмов на Си, таких как вычисление корня функции, приведение матрицы к ступенчатому виду методом Гаусса, работа с файлами и текстами и т.п.
Заключительная глава посвящена структурам данных и их реализациям. Рассматриваются структуры последовательного и прямого доступа, такие как стек, очередь, список, дерево, множество и нагруженное множество, а также их непрерывные и ссылочные реализации. Значительное место уделено реализациям множеста с помощью бинарного поиска, на базе сбалансированнных деревьев и с помощью хеш-функции.
Книга полезна студентам и преподавателям ВУЗов.

ISBN 5-9556-0039-6

ОСНОВЫ ИНФОРМАТИКИ И МАТЕМАТИКИ

Серия издается совместно

МОСКОВСКИМ ГОСУДАРСТВЕННЫМ УНИВЕРСИТЕТОМ
имени М.В.Ломоносова
и
Интернет-Университетом
Информационных Технологий
при поддержке корпорации
Microsoft

Информация о серии

Серия учебных пособий по информатике и ее математическим основам открыта в 2005 г. с целью современного изложения широкого спектра направлений информатики на базе соответствующих разделов математических курсов, а также примыкающих вопросов, связанных с информационными технологиями.

Особое внимание предполагается уделять возможности использовать материалы публикуемых пособий в преподавании информатики и ее математических основ для непрофильных специальностей. Редакционная коллегия также надеется представить вниманию читателей широкую гамму практикумов по информатике и ее математическим основам, реализующих основные алгоритмы и идеи теоретической информатики.

Выпуск серии начат при поддержке корпорации Microsoft в рамках междисциплинарного научного проекта МГУ имени М.В. Ломоносова.

ПРЕДИСЛОВИЕ

Основой книги является курс программирования, который на протяжении многих лет читается на механико-математическом факультете МГУ. Главные идеи курса принадлежат А. Г. Кушниренко и Г. В. Лебедеву, они содержатся в книге [3]. Особенность курса заключается в том, что не придается большого значения изучению конкретных языков программирования — скорее, он посвящен общим принципам программирования, следовать которым можно, работая в любой языковой среде. Основным понятием курса является понятие исполнителя, которое соответствует понятию класса в объектно-ориентированных языках. Большое внимание уделяется методам разработки объемных проектов, в частности, реализации исполнителя на основе нижестоящих «базовых» исполнителей и технологии «сверху вниз». Поскольку курс предназначен студентам-математикам, в нем последовательно применяется математический подход к программированию. Например, рассматривается схема вычисления функций на последовательностях (таких как вычисление суммы или максимума последовательности, вычисление значения многочлена в заданной точке по последовательности его коэффициентов и т.п.), основанная на применении теории индуктивных функций и индуктивных расширений. Излагаются простейшие способы доказательства правильности программ по их тексту. Наиболее важными из них является явная формулировка утверждений в различных точках программы и схема построения цикла «пока» с помощью инварианта. Значительная часть курса посвящена структурам данных и их реализациям. Теоретический материал курса иллюстрируется практическими проектами небольшого объема (не более тысячи строк), такими как стековый калькулятор, текстовый редактор, простейший компилятор и др.

Вместе с тем данная книга не предназначена исключительно студентам-математикам, поэтому ее материал несколько упрощен по сравнению с курсом программирования мехмата и более направлен в сторону практического программирования. Больше внимания уделено понятию алгоритма и алгоритмического языка, представлению копьютерных целых чисел как элементов кольца вычетов, логическим выражениям, что частично дублирует начальные математические курсы.

Значительная часть книги посвящена изложению основ языка Си. Традиционно на механико-мататематическом факультете изучению конкретных языков уделяется мало внимания, считается, что большинство студентов способно самостоятельно прочитать учебник по языку Си и

C++. Тем не менее практика показывает, что даже у сильных студентов остаются пробелы в знании языка и умении применять его в конкретных задачах. Проблемой является и содержание многих учебников по Си: зачастую они предлагают не самые правильные решения (такие как реализация матрицы с помощью массива указателей), и даже содержат примеры, написанные, с точки зрения автора этой книги, в дурном стиле (например, заголовок функции `main` не содержит типа возвращаемого значения, что на самом деле неявно подразумевает тип `int`, а тело функции `main` не возвращает никакого значения). Для изучения Си и C++ можно порекомендовать лишь книги [6] и [8]. Но даже великолепная книга [6] написана уже очень давно, и многолетняя практика использования Си требует некоторых «стилевых» коррекций. Все это явилось причиной включения в данную книгу главы, посвященной основам Си, в которой автор попытался передать свой опыт программирования на языках Си и C++.

Заключительная глава книги посвящена структурам данных - традиционной для учебников по программированию теме. Рассматриваются структуры данных последовательного и прямого доступа — стек, очередь, список, дерево, множество и нагруженное множество. Подчеркивается разница между абстрактным описанием структуры данных и ее реализациями. (Так, для очереди возможны как непрерывная реализация на базе «циклического» массива, так и ссылочная реализация.) Приводятся основные способы непрерывных и ссылочных реализаций структур данных. Значительное внимание уделяется различным реализациям множества: с помощью бинарного поиска, на базе сбалансированных деревьев (рассматриваются AVL и красно-черные деревья) и с помощью хеш-функции.

Автор надеется, что книга принесет пользу как студентам-математикам, так и студентам непрофильных специальностей.

ОБ АВТОРЕ

Владимир Витальевич Борисенко,
кандидат физико-математических наук, старший научный сотрудник лаборатории вычислительных методов при кафедре вычислительной математики механико-математического факультета МГУ, автор и соавтор более 20 работ по алгебре, компьютерной алгебре и информатике, включая одну монографию по алгебре и школьные учебники информатики. На протяжении многих лет читал лекции по программированию на механико-математическом факультете МГУ. Имеет большой опыт системного и прикладного программирования, включая иммитационное моделирование, разработку сетевых драйверов, разработку компиляторов, компьютерную графику и компьютерную томографию.

Оглавление

Глава 1

Общее понятие алгоритма

Понятие алгоритма — одно из основных понятий программирования и математики. Алгоритм — это последовательность команд, предназначенная исполнителю, в результате выполнения которой он должен решить поставленную задачу. Алгоритм должен записывается на формальном языке, исключающем неоднозначность толкования. Исполнитель — это человек, компьютер, автоматическое устройство и т.п. Он должен уметь выполнять все команды, составляющие алгоритм, причем механически, «не раздумывая».

Запись алгоритма на формальном языке называется программой. Иногда само понятие алгоритма отождествляется с его записью, так что слова «алгоритм» и «программа» — почти синонимы. Небольшое различие заключается в том, что при упоминании алгоритма, как правило, имеют в виду основную идею его построения, общую для всех алгоритмических языков. Программа же всегда связана с записью алгоритма на конкретном формальном языке.

В математике рассматриваются различные виды алгоритмов — программы для машин Тьюринга, алгоритмы Маркова (нормальные алгоритмы), частично рекурсивные функции и т.п. Знаменитый тезис Чёрча утверждает, что все виды алгоритмов эквивалентны друг другу, т.е. классы задач, решаемых разными типами алгоритмов, всегда совпадают. Тезис этот недоказуем (можно лишь доказать совпадение для двух конкретных типов алгоритмов, например, машин Тьюринга и нормальных алгоритмов), но никто в его верности не сомневается. Так что все языки программирования эквивалентны друг другу

и различаются лишь тем, насколько они удобны для решения конкретных классов задач. Например, объектно-ориентированные языки оптимальны для программирования в оконных средах, а язык Фортран успешно применяется в научных и инженерных расчетах.

Большинство используемых в программировании алгоритмических языков имеют общие черты. В то же время, при изложении идеи алгоритма, например, при публикации в научной статье, не всегда целесообразно пользоваться каким-либо конкретным языком программирования, чтобы не загромождать изложение несущественными деталями. В таких случаях применяется неформальный алгоритмический язык, максимально приближенный к естественному. Язык такого типа называют псевдокодом. Для специалиста не составляет труда переписать программу с псевдокода на любой конкретный язык программирования. Запись алгоритма на псевдокоде зачастую яснее и нагляднее, она дает возможность свободно выбирать уровень детализации, начиная от описания в самых общих чертах и кончая подробным изложением.

Псевдокод объединяет существенные черты множества алгоритмических языков. Для записи алгоритмов в данном курсе будет использоваться как псевдокод, так и конкретные языки: Си, C++ и C#.

1.1. Алгоритмические языки

Программирование начиналось с записи программ непосредственно в виде машинных команд (в кодах, как говорят программисты). Позже для облегчения кодирования был разработан язык Ассемблера, который позволяет записывать машинные команды в символическом виде. Например, программисту не нужно помнить числовой код операции сложения, вместо этого можно использовать символическое обозначение *ADD*. Язык Ассемблера зависит от системы команд конкретного компьютера. Он достаточно удобен для программирования небольших задач, требующих максимальной скорости выполнения. Однако, крупные проекты разрабатывать на языке Ассемблера трудно. Главная проблема состоит в том, что программа, написанная на Ассемблере, привязана к архитектуре конкретного компьютера и не может быть перенесена на другие машины. При усовершенствова-

нии компьютера все программы на Ассемблере приходится переписывать заново.

Почти сразу с возникновением компьютеров были разработаны языки высокого уровня, т.е. языки, не зависящие от конкретной архитектуры. Для выполнения программы на языке высокого уровня ее нужно сначала перевести на язык машинных команд. Специальная программа, выполняющая такой перевод, называется *транслятором* или *компилятором*. Оттранслированная программа затем выполняется непосредственно компьютером. Существует также возможность перевода программы на промежуточный язык, не зависящий от архитектуры конкретного компьютера, но тем не менее максимально приближенный к языку машинных команд. Затем программа на промежуточном языке выполняется специальной программой, которая называется *интерпретатором*. Возможен также вариант *компиляции «на лету»* (Just In Time Compilation), когда выполняемый фрагмент программы переводится с промежуточного языка на язык машинных команд непосредственно перед выполнением.

Наиболее распространенные компилируемые языки — это Си, C++, Фортран, Паскаль. Интерпретируемые и компилируемые «на лету» языки — это в основном объектно-ориентированные языки, такие как Java, Visual Basic и C#. Все они вначале переводятся на промежуточный язык: для Java это так называемый байткод языка Java, для Visual Basic и C# — так называемый промежуточный язык (Intermediate Language или просто IL), являющийся одним из основных компонентов платформы ".Net" фирмы Microsoft. Промежуточный язык может интерпретироваться специальным исполнителем (например, виртуальной Java-машиной), но, как правило, в современных системах применяется компиляция «на лету», что позволяет достичь большего быстродействия.

Исторически одним из первых языков высокого уровня был Фортран. Он оказался исключительно удачным — простым и в то же время очень эффективным. До сих пор большая часть научных и инженерных программ написана на Фортране. Тем не менее, в последние 20 лет программисты отдают предпочтение языку Си и связанной с ним линии объектно-ориентированных языков — C++, Java и C#.

Другой значительной вехой в истории алгоритмических языков является разработка языка Алгол-60 (расшифровывается как алго-

ритмический язык — ALGOrithmic Language). Возникновение языка Алгол-60 связано с развитием структурного подхода к программированию, в котором используется вложение конструкций языка друг в друга. Так, основная единица языка — оператор — может быть простым или составным, т.е. состоящим в свою очередь из нескольких операторов, заключенных в блок с помощью ключевых слов begin и end. Внутри блока можно описывать локальные переменные, недоступные извне блока, и даже подпрограммы или функции.

Язык Алгол-60 способствовал развитию алгоритмических языков, его наследником является, например, Паскаль и вся линия связанных с ним языков: Modula-2, Oberon и Delphy. Тем не менее, Алгол-60 оказался далеко не таким удачным, как Фортран. В нем присутствовали непродуманные решения, в частности, возможность вложения подпрограмм внутрь других подпрограмм, а также неудачный механизм передачи параметров подпрограмм. Из-за этого Алгол-60 не был реализован на практике в полном соответствии со стандартом (в отличие от языков типа Алгамс, отступавших от стандарта в сторону простоты и удобства использования). Язык Паскаль появился тоже как коррекция Алгола-60, но, к сожалению, унаследовал его главное неудачное решение — вложенность подпрограмм друг в друга. Также в первоначальном варианте языка Паскаль отсутствовала возможность разбиения программы на файлы. Эти недостатки были затем исправлены автором Паскаля, замечательным швейцарским ученым и педагогом Никлаусом Виртом, в языках Modula-2 и Oberon. Но, к сожалению, программистское сообщество проигнорировало язык Oberon, остановившись на немного улучшенном варианте языка Паскаль. В настоящее время Паскаль, как правило, используется для обучения программированию, но не в практической работе.

Наконец, самый успешный язык программирования — язык Си и связанная с ним линия объектно-ориентированных языков: C++, Java, C#. В отличие от Алгола-60, язык Си был создан не теоретиками, а практическими программистами, обладающими при этом высокой математической культурой. Язык был разработан в конце 60-х годов XX века. Он впервые позволил реально избавиться от Ассемблера при создании операционных систем. Например, практически весь текст операционной системы Unix написан на языке Си и, таким образом, не зависит от конкретного компьютера. Главным

достоинством Си является его простота и отсутствие псевдонаучных решений (таких, как вложенность блоков программ друг в друга: в Си функция не может содержать внутри себя другую функцию, а переменные четко разделяются на глобальные и локальные — не так, как в Алголе, где локальные переменные подпрограммы являются глобальными для всех вложенных в нее подпрограмм). Просто и ясно описан механизм передачи параметров в функцию (только по значению). Программист, создающий программу на Си, всегда четко понимает, как эта программа будет выполняться. Понятие указателя, статические и автоматические (стековые) переменные языка Си максимально близко отражают устройство любого современного компьютера, поэтому программы на Си эффективны и удобны для отладки.

В настоящее время подавляющая часть программ пишется на языках Си и C++. Интерфейс любой операционной системы (так называемый API — Application Program Interface), т.е. набор системных вызовов, предназначенных для разработчиков прикладных программ, как правило, представляет собой набор функций на языке Си. Наконец, современные объектно-ориентированные языки также основаны на языке Си. Это язык C++, занимающий промежуточное положение между традиционными и объектно-ориентированными языками, а также объектно-ориентированные языки Java и C#.

В курсе будем использовать псевдокод для неформальной записи алгоритмов, а также языки Си, C++ и C# для практического программирования. Применение объектно-ориентированных языков C++ и C# значительно облегчает программирование оконных приложений в системах типа Windows, тогда как при разработке программ, не связанных с графическим интерфейсом (например, математических расчетов), можно обойтись и более простым языком Си.

1.2. Управляющие конструкции алгоритмического языка

Большинство алгоритмических языков относится к так называемым процедурным языкам, в которых основной единицей является оператор. Оператор представляет собой команду на выполнение некоторого действия. Язык, таким образом, состоит в основном из фраз

в повелительном наклонении. Альтернативой операторам являются описания, определяющие объекты или типы объектов и их взаимосвязи. Считается, что чем больший процент составляют описания, тем более совершенным является язык. Существуют алгоритмические языки, состоящие в основном из описаний (функциональные языки), однако, данный курс ограничивается процедурными языками.

Всякий алгоритм предназначен исполнителю, который однозначно понимает команды алгоритма. Пример: опишем алгоритм проезда от Аэровокзала в Москве до аэропорта «Домодедово».

```
алгоритм Проезд от Аэровокзала до Домодедово через МКАД
| Дано: находимся у Аэровокзала
| Надо: оказаться в аэропорту Домодедово
начало алгоритма
| повернуть направо на центральный проезд
|   Ленинградского проспекта в сторону центра;
| проехать до второго светофора;
| выполнить разворот на перекрестке
| проехать по Ленинградскому проспекту из центра
|   до пересечения с Московской кольцевой дорогой;
| переехать мост над кольцевой дорогой и
|   повернуть направо на внешнюю часть кольцевой дороги;
| двигаться по кольцевой дороге в направлении против
|   часовой стрелки до Каширского шоссе;
| повернуть направо на Каширское шоссе в сторону из города;
| двигаться, никуда не сворачивая, до
|   аэропорта Домодедово;
конец алгоритма
```

Строки алгоритма представляют собой фразы в повелительном наклонении, которые предназначены исполнителю алгоритма, т.е. любому водителю, который может отличить внешнюю сторону кольцевой дороги от внутренней. Строки алгоритма выполняются последовательно; считается, что исполнитель алгоритма способен не задумываясь выполнить каждую его команду.

Большинство алгоритмов не сводится, однако, к последовательному выполнению команд, в них присутствуют ветвления и циклы. При ветвлении в зависимости от условия выполняется одна из двух

ветвей программы; для этого используется оператор “если ... то ... иначе ... конец если”. Например, можно модифицировать приведенный выше алгоритм, используя выбор одного из двух альтернативных путей, в зависимости от наличия транспортной пробки.

```
алгоритм Оптимальный путь от Аэровокзала до Домодедово
| Дано: находимся у Аэровокзала
| Надо: оказаться в аэропорту Домодедово
начало алгоритма
| если нет пробки на Ленинградском проспекте
| |   в направлении из центра
| | то
| |  // ...выполняем предыдущий алгоритм...
| |  Проезд от Аэровокзала до Домодедово через МКАД
| | иначе
| |  повернуть направо на боковой проезд
| |     Ленинградского проспекта в сторону центра;
| |  доехать до пересечения с Беговой улицей;
| |  повернуть направо на Третье транспортное кольцо;
| |  ехать по Третьему транспортному кольцу против
| |     часовой стрелки до пересечения с Варшавским шоссе;
| |  повернуть направо на Варшавское шоссе
| |     в сторону из центра;
| |  ехать прямо до развилки с Каширским шоссе;
| |  на развилке с Каширским шоссе проехать прямо в сторону
| |     Каширского шоссе; // Варшавское уходит направо
| |  двигаться, никуда не сворачивая, до
| |     аэропорта Домодедово;
| конец если
конец алгоритма
```

Здесь исполнитель алгоритма сначала должен проверить условие

```
нет пробки на Ленинградском проспекте
в направлении из центра
```

Если это условие истинно, то выполняется первый алгоритм “Проезд от Аэровокзала до Домодедово через МКАД”; если ложно — часть алгоритма между строками “иначе” и “конец если”. Следует отметить, что

1) здесь выполняется алгоритм “Проезд от Аэровокзала до Домодедово через МКАД”, описанный ранее. Возможность использования (вызова) описанных ранее алгоритмов является важной чертой любого алгоритмического языка, позволяющей строить более сложные алгоритмы из имеющихся заготовок;

2) дважды был использован символ комментария //. Текст, расположенный справа от этого символа, игнорируется исполнителем алгоритма, он нужен лишь составителю алгоритма или тому, кто затем будет его исправлять или модифицировать. Комментарии являются важнейшей составной частью любых программ, это способ общения программистов друг с другом (или даже с самим собой, что нужно при разработке больших программ, которая занимает длительное время). Комментарии объясняют, что составитель алгоритма имел в виду в случаях, когда идея алгоритма не очевидна с первого взгляда.

Второй важнейшей конструкцией алгоритмического языка является конструкция “цикл пока”. Заголовок цикла состоит из ключевых слов “цикл пока”, за которыми следует некоторое условие. Дальше записывается тело цикла, завершаемое строкой “конец цикла”. При выполнении цикла исполнитель сначала проверяет условие в заголовке тела цикла. Если условие истинно, то выполняется тело цикла. Затем вновь проверяется условие в заголовке цикла, опять выполняется тело цикла, если условие истинно, и так до бесконечности. Если же условие ложно с самого начала или становится ложным в результате предыдущего выполнения тела цикла, то тело цикла не выполняется и цикл завершается. Таким образом, по выходу из цикла условие, записанное в его заголовке, всегда ложно. Если условие ложно перед началом цикла, то цикл не выполняется ни разу! Программисты иногда называют “цикл пока” циклом с предусловием, поскольку условие продолжения цикла проверяется *перед* выполнением тела цикла, а не после него. Иногда используют циклы с постусловием (do... while), когда тело цикла всегда выполняется хотя бы один раз, а условие продолжения проверяется *после* каждой итерации. Всегда предпочтительнее использовать цикл с предусловием, это помогает избежать многих ошибок.

Для иллюстрации конструкции “цикл пока” можно привести сле-

дующую модификацию алгоритма проезда.

```
алгоритм Добраться из Аэровокзала до Домодедово
| Дано: находимся у Аэровокзала
| Надо: оказаться в аэропорту Домодедово
начало алгоритма
|
| цикл пока пробка на Ленинградском проспекте
| | выпить чашку кофе в кафе Аэровокзала
| | ждать полчаса
| конец цикла
|
| Проезд от аэровокзала до Домодедово через МКАД
конец алгоритма
```

Здесь снова использован определенный ранее алгоритм “Проезд от аэровокзала до Домодедово”. Условие продолжения цикла проверяется перед выполнением тела цикла, но не в процессе его выполнения! Так, если пробка рассосалась после чашки кофе, то все равно нужно ждать полчаса.

Теперь можно подвести итоги.

Запись алгоритма на неформальном языке представляет собой последовательность команд исполнителю алгоритма. Запись может также включать управляющие конструкции: ветвление, или условный оператор, и цикл “пока”. Условный оператор выглядит следующим образом:

```
если условие
| то
|   последовательность действий 1
| иначе
|   последовательность действий 2
конец если
```

Последовательность действий 1 выполняется, когда условие истинно; в противном случае выполняется последовательность действий 2. Ключевое слово “иначе” и последовательность действий 2 могут отсутствовать; в этом случае, когда условие ложно, исполнитель ничего не делает.

Цикл “пока”, или цикл с предусловием выглядит следующим образом:

```
цикл пока условие
| последовательность действий
конец цикла
```

Сначала проверяется условие в заголовке цикла. Если оно истинно, то выполняется последовательность действий, составляющая тело цикла. Это повторяется неограниченное число раз, пока условие истинно. Цикл заканчивается, когда условие при очередной проверке оказывается ложным. Важно отметить, что условие проверяется перед каждым выполнением тела цикла, но не в процессе его выполнения.

Помимо элементарных действий, в записи алгоритма можно использовать другие алгоритмы. Для вызова другого алгоритма нужно просто указать его название. (В некоторых языках, например, в Фортране, для вызова алгоритма используется ключевое слово CALL.) Также в записи алгоритма могут присутствовать комментарии, которые игнорируются исполнителем алгоритма. Для отделения комментария будут использоваться знаки // (двойная косая черта) в соответствии с синтаксисом языка C++.

1.3. Понятие переменной

Алгоритм состоит из команд исполнителю. Исполнитель может, в свою очередь, командовать другими исполнителями. Компьютер можно рассматривать как универсальный исполнитель, который управляет другими исполнителями. Рассмотрим, к примеру, автомобиль с инжекторным двигателем. В нем работой двигателя управляет компьютер (его иногда называют «микропроцессорный блок»). Компьютер получает данные от разнообразных датчиков (датчики положения коленчатого вала и дроссельной заслонки, температуры охлаждающей жидкости, скорости, детонации, кислорода и др.) и отдает приказания исполняющим системам двигателя — модулю зажигания, бензонасосу, форсункам двигателя, регулятору холостого хода, системе продувки адсорбера и т.д. Таким образом, и датчики, и исполняющие

системы двигателя управляются компьютером, который выступает в роли универсального исполнителя.

Запись алгоритма для универсального исполнителя может включать команды, адресованные ему непосредственно, а также команды, которые нужно передать подчиненным исполнителям. В чем разница между универсальным и простейшими подчиненными исполнителями?

Как правило, универсальный исполнитель имеет собственную память, и выполнение им команд может приводить не к каким-либо внешним действиям, а к изменению его внутреннего состояния. Например, используя сигналы от датчика фазы, компьютер автомобиля вычисляет текущие обороты двигателя (которые показывает на тахометре). Используя эти данные и информацию, поступающую от датчика скорости автомобиля, компьютер может вычислить, какая передача включена в определенный момент времени. Далее вычисляется текущая нагрузка на двигатель и устанавливается, какой должна быть смесь бензина и воздуха, подаваемая в цилиндры двигателя. В зависимости от этого подаются команды на открытие форсунок. От степени обогащения смеси зависит момент зажигания — чем богаче смесь, тем позже момент зажигания; таким образом, подаче команды модулю зажигания предшествуют достаточно сложные вычисления.

Таким образом, компьютер автомобиля, управляющий работой двигателя, хранит в любой момент времени в своей памяти текущие скорость, передачу, нагрузку на двигатель, температуру охлаждающей жидкости, требуемую степень обогащенности смеси и многие другие параметры. Эти параметры периодически перевычисляются на основании сигналов от разнообразных датчиков. В зависимости от значений параметров, компьютер передает те или иные сигналы управляющим системам двигателя.

Значение каждого параметра хранится в определенном участке памяти компьютера и может меняться в процессе выполнения алгоритма. Такой участок памяти компьютера называется «переменной». Понятие переменной — важнейшее понятие алгоритмического языка. Переменные встроены в конструкцию универсального исполнителя.

Каждой переменной присваивается имя. В рассмотренном примере используются переменные “скорость”, “обороты двигателя”, “передача”, “нагрузка”, “температура”, “обогащенность смеси”, “угол опе-

режения зажигания" и другие. С каждой переменной связан ее тип, т.е. множество значений, которое она может принимать. Например, "передача" принимает целые значения от 1 до 5 (обратная и первая передачи не различаются), тогда как "скорость", а также "обогащенность смеси" принимают вещественные значения (скорость измеряется в м/сек, обогащенность смеси может измеряться либо соотношением кислорода и паров бензина в единице объема, либо в процентах относительно «стехиометрической» смеси 14/1, соответствующей полному сгоранию паров бензина).

С переменной можно выполнять два действия:

1) прочитать текущее значение переменной;

2) записать новое значение в переменную или, как говорят программисты, присвоить новое значение переменной.

В алгоритмическом языке чтение значения переменной выполняется в результате использования ее имени в любом выражении. Запись нового значения переменной выполняется с помощью так называемого *оператора присваивания*. Он выглядит следующим образом:

```
имя переменной := выражение;
```

Знак := читается как присвоить значение. Во многих языках вместо него используется просто знак равенства:

```
имя переменной = выражение;
```

При выполнении оператора присваивания сначала вычисляется значение выражения в правой части, затем оно записывается в переменную, имя которой указано в левой части. Старое значение переменной при этом стирается. Например, скорость автомобиля вычисляется по количеству импульсов от датчика скорости в единицу времени: датчик скорости посылает 6 импульсов на каждый пройденный метр.

```
скорость := число импульсов от датчика скорости /
            (6 * интервал времени);
```

Переменная "число импульсов от датчика скорости" в течение каждого интервала времени суммирует число импульсов. В начале каждого интервала она обнуляется. Полученная в результате скорость

выражается в м/с. Если нужно получить скорость в км/час, то дополнительно выполняется следующее действие:

```
скорость := скорость * 3600 / 1000;
```

Здесь переменная "скорость" входит как в правую, так и в левую части оператора присваивания. В правой части используется старое значение этой переменной, вычисленное в м/сек. Поскольку час содержит 3600 секунд, то при домножении на 3600 получается расстояние в метрах, проходимое за 1 час; после деления на 1000 получается расстояние в километрах. Вычисленное значение затем присваивается переменной "скорость".

Суммируем сказанное выше:

1) универсальный исполнитель, или компьютер, — это исполнитель, который может управлять другими исполнителями. Запись алгоритма для универсального исполнителя может включать команды, которые он должен передать подчиненным исполнителям, и команды, изменяющие внутреннее состояние самого универсального исполнителя;

2) внутреннее состояние универсального исполнителя определяется состоянием его памяти. Память — это материальный носитель (лента машины Тьюринга, ламповая или ферритовая память первых компьютеров, полупроводниковая память современных компьютеров), который хранит информацию. Эту информацию можно читать и перезаписывать;

3) переменная — это область памяти универсального исполнителя, хранящая порцию информацию. Любая переменная имеет имя и тип. Тип переменной определяется множеством всех значений, которые она может принимать. Память универсального исполнителя можно рассматривать как набор переменных;

4) с переменной можно выполнять два действия: прочитать ее текущее значение и записать в нее новое значение (старое теряется). В алгоритмическом языке значение переменной читается, когда ее имя используется в любом выражении, значение которого надо вычислить. Для записи нового значения в переменную применяется оператор присваивания, который имеет

вид

```
имя переменной := выражение;
```

При его выполнении сначала вычисляется значение выражения справа от знака присваивания :=, затем оно записывается в переменную. Выражение в правой части может включать имя переменной в левой части. В этом случае при вычислении выражения используется старое значение переменной.

1.4. Типы переменных

Тип переменной определяется множеством значений, которое она может принимать. Кроме того, тип определяет операции, которые возможны с переменной. Например, с численными переменными возможны арифметические операции, с логическими — проверка, истинно или ложно значение переменной, с символьными — сравнение, с табличными (или массивами) — чтение или запись элемента таблицы с заданным индексом и т.п. Как правило, в любом современном языке имеется базовый набор типов и несколько конструкций, которые позволяют строить новые типы из уже созданных. Наборы базовых типов и конструкций различаются для разных языков. В описании неформального алгоритмического языка будут использоваться типы и конструкции, которые присутствуют в большинстве языков практического программирования.

1.4.1. Целочисленные переменные

Тип «целое число» является основным для любого алгоритмического языка. Связано это с тем, что содержимое ячейки памяти или регистра процессора можно рассматривать как целое число. Адреса элементов памяти также представляют собой целые числа, с их помощью записываются машинные команды и т.д. Символы представляются в компьютере целыми числами — их кодами в некоторой кодировке. Изображения также задаются массивами целых чисел: для каждой точки цветного изображения хранятся интенсивности ее красной, зеленой и синей составляющей (в большинстве случаев — в диапазоне от 0 до 255). Как говорят математики, целые числа даны свыше, все остальное сконструировал из них человек.

Общепринятый в программировании термин «целое число» или «целочисленная переменная», строго говоря, не вполне корректен. Целых чисел бесконечно много, десятичная или двоичная запись целого числа может быть сколь угодно длинной и не помещаться в области памяти, отведенной под одну переменную. Целая переменная в компьютере может хранить лишь ограниченное множество целых чисел в некотором интервале. В современных компьютерах под целую переменную отводится 4 байта, т.е. 32 двоичных разряда. Она может хранить числа от нуля до 2 в 32-й степени минус 1. Таким образом, максимальное целое число, которое может храниться в целочисленной переменной, равно

$$2^{32} - 1 = 4294967295.$$

Сложение и умножение значений целых переменных выполняется следующим образом: сначала производится арифметическая операция, затем старшие разряды результата, вышедшие за границу тридцати двух двоичных разрядов (т.е. четырех байтов), отбрасываются. Определенные таким образом операции удовлетворяют традиционным законам коммутативности, ассоциативности и дистрибутивности:

$$\begin{aligned} &a + b = b + a, \quad ab = ba, \\ &(a + b) + c = a + (b + c), \quad (ab)c = a(bc), \\ &a(b + c) = ab + ac. \end{aligned}$$

Кольцо вычетов по модулю m

Целочисленный тип компьютера в точности соответствует важнейшему понятию математики — понятию кольца вычетов по модулю m. В качестве m выступает число $2^{32} = 4294967296$. В математике кольцо $\mathbf{Z}_m$ определяется следующим образом. Все множество целых чисел Z разбивается на m классов, которые называются классами эквивалентности. Каждый класс содержит числа, попарная разность которых делится на m. Первый класс содержит числа

$$\{\ldots, -2m, -m, 0, m, 2m, \ldots\}$$

второй

$$\{\ldots, -2m + 1, -m + 1, 1, m + 1, 2m + 1, \ldots\}$$

последний

$$\{\dots, -m-1, -1, m-1, 2m-1, 3m-1, \dots\}$$

Элементами кольца $\mathbf{Z}_m$ являются классы эквивалентности. Их ровно m, так что, в отличие от множества целых чисел $\mathbf{Z}$, кольцо $\mathbf{Z}_m$ содержит конечное число элементов. Операции с классами выполняются следующим образом: надо взять по одному представителю из каждого класса, произвести операцию и определить, в какой класс попадает результат. Этот класс и будет результатом операции. Легко показать, что он не зависит от выбора представителей.

Все числа, принадлежащие одному классу эквивалентности, имеют один и тот же остаток при делении на m. Таким образом, класс эквивалентности однозначно определяется остатком от деления на m. Традиционно остаток выбирается неотрицательным, в диапазоне от 0 до $m-1$. Остатки используют для обозначения классов, при этом используются квадратные скобки. Так, выражение $[5]$ обозначает класс эквивалентности, состоящий из всех чисел, остатки которых при делении на m равны пяти. Все кольцо $\mathbf{Z}_m$ состоит из элементов

$$[0], [1], [2], \dots, [m-1],$$

например, кольцо $\mathbf{Z}_5$ состоит из элементов

$$[0], [1], [2], [3], [4].$$

В элементарной школьной математике результат операции остатка от деления традиционно считается неотрицательным. Операция нахождения остатка будет обозначаться знаком процента %, как в языке Си. Тогда, к примеру,

$$\begin{aligned} &3\%5 = 3,\\ &17\%5 = 2,\\ &(-3)\%5 = 2,\\ &(-17)\%5 = 3. \end{aligned}$$

Отсюда видно, что в школьной математике не выполняется равенство

$$(-a)\%b = -(a\%b),$$

т.е. операции изменения знака и нахождения остатка не перестановочны (на математическом языке, не коммутируют друг с другом). В компьютере операция нахождения остатка от деления для отрицательных чисел определяется иначе, ее результат может быть отрицательным. В приведенных примерах результаты будут следующими:

$$3\%5 = 3,$$
$$17\%5 = 2,$$
$$(-3)\%5 = -3,$$
$$(-17)\%5 = -2.$$

При делении на положительное число знак остатка совпадает со знаком делимого. При таком определении тождество

$$(-a)\%b = a\%(-b) = -(a\%b)$$

справедливо. Это позволяет во многих алгоритмах не следить за знаками (так же, как в тригонометрии формулы, выведенные для углов, меньших 90 градусов, автоматически оказываются справедливыми для любых углов).

Вернемся к рассмотрению кольца $\mathbf{Z}_m$. Выберем по одному представителю из каждого класса эквивалентности, которые составляют множество $\mathbf{Z}_m$. Систему таких представителей называют «системой остатков». Традиционно рассматривают две системы остатков: неотрицательную систему и симметричную систему. Неотрицательная система остатков состоит из элементов

$$0, 1, 2, 3, \ldots, m-1.$$

Очень удобна также симметричная система остатков, состоящая из отрицательных и неотрицательных чисел, не превосходящих m/2 по абсолютной величине. Пусть

$$k = \textit{целая часть}(m/2),$$

тогда симметричная система остатков при нечетном m состоит из элементов

$$-k, -k+1, \ldots, -1, 0, 1, \ldots, k-1, k,$$

а при четном m — из элементов

$$-k, -k+1, \ldots, -1, 0, 1, \ldots, k-1.$$

Например, при $m = 5$ симметричная система остатков состоит из элементов

$$-2, -1, 0, 1, 2.$$

Кольцо $\mathbf{Z}_m$ можно представлять состоящим из элементов, принадлежащих выбранной системе остатков. Арифметические операции определяются следующим образом: надо взять два остатка, произвести над ними операцию как над обычными целыми числами и выбрать тот остаток, которой лежит в том же классе эквивалентности, что и результат операции. Например, для симметричной системы остатков множества $\mathbf{Z}_5$ имеем:

$$\begin{array}{ll} 1+1=2, & 1+2=-2, \\ 1+(-2)=-1, & 1+(-1)=0, \\ (-2)+2=0, & (-2)+(-2)=1. \end{array}$$

Интерпретация положительных и отрицательных чисел

В кольце вычетов невозможно определить порядок, согласованный с операциями (т.е. так, чтобы, к примеру, сумма двух положительных чисел была положительной). Таким образом, в компьютере нет, строго говоря, положительных и отрицательных целых чисел, поскольку компьютерные «целые» числа — это на самом деле элементы кольца вычетов. Выбирая либо неотрицательную, либо симметричную систему остатков, можно интерпретировать эти числа либо как неотрицательные в диапазоне от нуля до $m - 1$, либо как отрицательные и положительные числа в диапазоне от $-k$ до k, где k — целая часть от деления m на 2.

В программировании симметричная система остатков более популярна, поскольку трудно обойтись без отрицательных чисел. При этом следует понимать, что сумма двух положительных чисел может оказаться отрицательной, или, наоборот, сумма двух отрицательных чисел — положительной. Иногда в программировании такую ситуацию называют *переполнением*. Привычные свойства целочисленных операций в компьютере выполняются лишь для небольших чисел, когда результат операции не превосходит числа $m = 2^{32}$. В случае целочисленных переменных переполнение не является экстраординарной ситуацией и не приводит к аппаратным ошибкам или прерываниям.

(Это, кстати, отличает компьютерные целые числа от вещественных.) Переполнение — совершенно нормальная ситуация, если вспомнить, что компьютер работает с элементами кольца вычетов по модулю m, а не с настоящими целыми числами.

Следует также отметить, что симметричная система остатков кольца $\mathbf{Z}_m$ в случае четного m (а m для компьютера равно 2^{32}, т.е. четно) не вполне симметрична. Поскольку ноль не имеет знака, то число положительных остатков не может равняться числу отрицательных.

Какой остаток выбрать в классе эквивалентности числа $k = m/2$? Для этого элемента выполняется непривычное с точки зрения школьной математики равенство

$$k + k \equiv 0 \pmod{m},$$

т.е.

$$k \equiv -k \pmod{m}$$

Как отрицательный остаток $-k$, так и положительный k в равной мере подходят для представления этого класса эквивалентности. По традиции выбирается отрицательный остаток. Таким образом, в компьютере количество отрицательных целых чисел на единицу больше, чем количество положительных. Так как $m = 2^{32} = 4294967296$, то $k = 2^{31} = 2147483648$, и симметричная система остатков состоит из элементов

$$-2147483648, -2147483647, \ldots, -2, -1, 0, 1, 2, \ldots, 2147483647.$$

В двоичном представлении старший разряд у отрицательных целых чисел равен единице, у положительных — нулю. Двоичные разряды представления целого числа в программировании нумеруют от 0 до 31 справа налево. Старший разряд имеет номер 31 и часто называется знаковым разрядом. Таким образом, знаковый разряд равен единице у всех отрицательных чисел и нулю у неотрицательных. Двоичное представление максимального по абсолютной величине отрицательного числа k состоит из единицы и тридцати одного нуля:

$$-2147483648_{10} = 10000000000000000000000000000000_2.$$

Двоичное представление числа -1 состоит из тридцати двух единиц:

$$-1_{10} = 11111111111111111111111111111111_2.$$

Двоичное представление максимального положительного числа состоит из нуля в знаковом разряде и тридцати одной единицы:

$$2147483647_{10} = 01111111111111111111111111111111_2.$$

Следует отметить, что в программировании часто используют также короткие целые числа, двоичная запись которых занимает восемь разрядов, т.е. один байт, или шестнадцать разрядов, т.е. два байта. Работа с такими короткими целыми числами поддерживается на аппаратном уровне. В языки Си однобайтовым целым числам соответствует тип char (тип char в Си — это именно целые числа, символы представляются их целочисленными кодами), двухбайтовым — тип short. Однобайтовые целые числа — это элементы кольца вычетов $\mathbf{Z}_m$, где $m = 2^8 = 256$. Симметричная система остатков в этом случае состоит из элементов

$$-128, -127, \ldots, -2, -1, 0, 1, 2, \ldots, 127.$$

В случае двухбайтовых целых чисел (тип short) $m = 2^{16} = 65536$, а симметричная система остатков состоит из элементов

$$-32768, -32767, \ldots, -2, -1, 0, 1, 2, \ldots, 32767.$$

1.4.2. Вещественные переменные

Вещественные числа представляются в компьютере в так называемой экспоненциальной, или плавающей, форме. Вещественное число r имеет вид

$$r = \pm 2^e \cdot m$$

Представление числа состоит из трех элементов:

1) знак числа — плюс или минус. Под знак числа отводится один бит в двоичном представлении, он располагается в старшем, т.е. знаковом разряде. Единица соответствует знаку минус, т.е. отрицательному числу, ноль — знаку плюс. У нуля знаковый разряд также нулевой;

2) показатель степени e, его называют порядком или экспонентой. Экспонента указывает степень двойки, на которую домножается число. Экспонента может быть как положительной, так и

отрицательной (для чисел, меньших единицы). Под экспоненту отводится фиксированное число двоичных разрядов, обычно восемь или одиннадцать, расположенных в старшей части двоичного представления числа, сразу вслед за знаковым разрядом;

3) мантисса m представляет собой фиксированное количество разрядов двоичной записи вещественного числа в диапазоне от 1 до 2:

$$1 \leq m < 2$$

Следует подчеркнуть, что левое неравенство нестрогое — мантисса может равняться единице, а правое — строгое, мантисса всегда меньше двух. Разряды мантиссы включают один разряд целой части, который ввиду приведенного неравенства всегда равен единице, и фиксированное количество разрядов дробной части. Поскольку старший двоичный разряд мантиссы всегда равен единице, хранить его необязательно, и в двоичном коде он отсутствует. Фактически двоичный код хранит только разряды дробной части мантиссы.

В языке Си вещественным числам соответствуют типы float и double. Элемент типа float занимает 4 байта, в которых один бит отводится под знак, восемь — под порядок, остальные 23 — под мантиссу (на самом деле, в мантиссе 24 разряда, но старший разряд всегда равен единице, поэтому хранить его не нужно). Тип double занимает 8 байтов, в них один разряд отводится под знак, 11 — под порядок, остальные 52 — под мантиссу. На самом деле в мантиссе 53 разряда, но старший всегда равен единице и поэтому не хранится. Поскольку порядок может быть положительным и отрицательным, в двоичном коде он хранится в *смещенном виде*: к нему прибавляется константа, равная абсолютной величине максимального по модулю отрицательного порядка. В случае типа float она равна 127, в случае double — 1023. Таким образом, максимальный по модулю отрицательный порядок представляется нулевым кодом.

Основным типом является тип double, именно он наиболее естественен для компьютера. В программировании следует по возможности избегать типа float, так как его точность недостаточна, а процессор все равно при выполнении операций преобразует его в тип double.

(Один из немногих случаев, где применение типа float оправдано, — трехмерная компьютерная графика.)

Несколько примеров представления вещественных чисел в плавающей форме:

1) $1.0 = +2^0 \cdot 1.0$
   Здесь порядок равен 0, мантисса — 1. В двоичном коде мантисса состоит из одних нулей, так как старший разряд мантиссы (всегда единичный) в коде отсутствует. Порядок хранится в двоичном коде в смещенном виде, он равен 127 в случае float и 1023 в случае double;

2) $3.5 = +2^1 \cdot 1.75$
   Порядок равен единице, мантисса состоит из трех единиц, из которых в двоичном коде хранятся две: 1100...0; смещенный порядок равен 128 для float и 1024 для double;

3) $0.625 = +2^{-1} \cdot 1.25$
   Порядок отрицательный и равен -1, дробная часть мантиссы равна 0100...0; смещенный порядок равен 126 для float и 1022 для double;

3) $100.0 = +2^6 \cdot 1.5625$
   Порядок равен шести, дробная часть мантиссы равна 100100...0; смещенный порядок равен 133 для float и 1029 для double.

При выполнении сложения двух положительных плавающих чисел происходят следующие действия:

1) выравнивание порядков. Определяется число с меньшим порядком. Затем последовательно его порядок увеличивается на единицу, а мантисса делится на 2, пока порядки двух чисел не сравняются. Аппаратно деление на 2 соответствует сдвигу двоичного кода мантиссы вправо, так что эта операция выполняется быстро. При сдвигах правые разряды теряются, из-за этого может произойти потеря точности (в случае, когда правые разряды ненулевые);

2) сложение мантисс;

3) нормализация: если мантисса результата стала равна или превысила двойку, то порядок увеличивается на единицу, а мантисса делится на 2. В результате этого мантисса попадает в интервал $1 \leq m < 2$. При этом возможна потеря точности, а также переполнение, когда порядок превышает максимально возможную величину.

Вычитание производится аналогичным образом. При умножении порядки складываются, а мантиссы перемножаются как целые числа, после чего у результата правые разряды отбрасываются.

Машинный эпсилон

Действия с плавающими числами из-за ошибок округления лишь приближенно отражают арифметику настоящих вещественных чисел. Так, если к большому плавающему числу прибавить очень маленькое, то оно не изменится. Действительно, при выравнивании порядков все значащие биты мантиссы меньшего числа могут выйти за пределы разрядной сетки, в результате чего оно станет равным нулю. Таким образом, с плавающими числами возможна ситуация, когда

$$a + b = a \qquad \text{при} \qquad b \neq 0.$$

Более того, для сложения не выполняется закон ассоциативности:

$$a + (b + c) \neq (a + b) + c.$$

Действительно, пусть ε — максимальное плавающее число среди чисел, удовлетворяющих условию

$$1.0 + \varepsilon = 1.0$$

(приведенные выше рассуждения показывают, что такие числа существуют). Тогда

$$(1.0 + \varepsilon) + \varepsilon \neq 1.0 + (\varepsilon + \varepsilon),$$

поскольку левая часть неравенства равна единице, а правая строго больше единицы (это следует из максимальности числа ε).

Число ε часто называют *машинным эпсилоном* или, чуть менее корректно, машинным нулем, поскольку при прибавлении к единице

оно ведет себя как ноль. Величина машинного эпсилона характеризует точность операций компьютера. Она примерно одинакова для всех современных компьютеров: большинство процессоров работают с восьмибайтовыми плавающими числами (тип double в Си), а арифметика плавающих чисел подчиняется строгим международным стандартам.

Оценим величину машинного эпсилона для типа double. Число 1.0 записывается в плавающей форме как

$$1.0 = +2^0 \cdot 1.0.$$

Порядок плавающего числа 1.0 равен нулю. При сложении 1.0 с числом ε производится выравнивание порядка путем многократного сдвига мантиссы числа ε вправо и увеличения его порядка на 1. Поскольку все разряды числа ε должны в результате выйти за пределы разрядной сетки, должно быть выполнено 53 сдвига. Порядок числа ε после этого должен стать равным порядку числа 1.0, т.е. нулю. Следовательно, изначально порядок числа ε должен быть равным -53:

$$\varepsilon = 2^{-53} \cdot m,$$

где m — число в диапазоне от единицы до двух. Таким образом, величина машинного эпсилона составляет примерно

$$2^{-53} \approx 10^{-16}.$$

Приблизительно точность вычислений составляет 16 десятичных цифр. (Это также можно оценить следующим образом: 53 двоичных разряда составляют примерно 15.95 десятичных, поскольку $53/\log_2 10 \approx 53/3.321928 \approx 15.95$.)

В случае четырехбайтовых плавающих чисел (тип float языка Си) точность вычислений составляет примерно 7 десятичных цифр. Это очень мало, поэтому тип float чрезвычайно редко применяется на практике. К тому же процессор сконструирован для работы с восьмибайтовыми вещественными числами, а при работе с четырехбайтовыми он все равно сначала приводит их к восьмибайтовому типу. В программировании следует избегать типа float и всегда пользоваться типом double.

Некоторые процессоры применяют внутреннее представление плавающих чисел с бóльшим количеством разрядов мантиссы. Например, процессор Intel использует 80-битовое (десятибайтовое) представление. Поэтому точность вычислений, которые не записывают промежуточные результаты в память, может быть несколько выше указанных оценок.

Кроме потери точности, при операциях с вещественными числами могут происходить и другие неприятности:

1) переполнение — когда порядок результата больше максимально возможного значения. Эта ошибка часто возникает при умножении больших чисел;

2) исчезновение порядка — когда порядок результата отрицательный и слишком большой по абсолютной величине, т.е. порядок меньше минимально допустимого значения. Эта ошибка может возникнуть при делении маленького числа на очень большое или при умножении двух очень маленьких по абсолютной величине чисел.

Кроме того, некорректной операцией является деление на ноль. В отличие от операций с целыми числами, переполнение и исчезновение порядка считаются ошибочными ситуациями и приводят к аппаратному прерыванию работы процессора. Программист может задать реакцию на прерывание — либо аварийное завершение программы, либо, например, при переполнении присваивать результату специальное значение «плюс» или «минус бесконечность», а при исчезновении порядка — ноль. Заметим, что среди двоичных кодов, представляющих плавающие числа, имеется несколько специальных значений. Перечислим некоторые из них:

1) бесконечно большое число — это плавающее число с очень большим положительным порядком и, таким образом, очень большое по абсолютной величине. Оно может иметь знак плюс или минус;

2) бесконечно малое, или денормализованное, число — это ненулевое плавающее число с очень большим отрицательным порядком (т.е. очень маленькое по абсолютной величине);

3) Not a Number, или NaN — двоичный код, который не является корректным представлением какого-либо вещественного числа.

Любые операции с константой NaN приводят к прерыванию, поэтому она удобна при отладке программы — ею перед началом работы программы инициализируются значения всех вещественных переменных. Если в результате ошибки программиста при вычислении выражения используется переменная, которой не было присвоено никакого значения, то происходит прерывание из-за операции со значением NaN и ошибка быстро отслеживается. К сожалению, в случае целых чисел такой константы нет: любой двоичный код представляет некоторое целое число.

Запись вещественных констант

Вещественные константы записываются в двух формах — с фиксированной десятичной точкой или в экспоненциальном виде. В первом случае точка используется для разделения целой и дробной частей константы. Как целая, так и дробная части могут отсутствовать. Примеры:

1.2, 0.725, 1., .35, 0.

В трех последних случаях отсутствует либо дробная, либо целая часть. Десятичная точка должна обязательно присутствовать, иначе константа считается целой. Отметим, что в программировании именно точка, а не запятая, используется для отделении дробной части; запятая обычно служит для разделения элементов списка.

Экспоненциальная форма записи вещественной константы содержит знак, мантиссу и десятичный порядок (экспоненту). Мантисса — это любая положительная вещественная константа в форме с фиксированной точкой или целая константа. Порядок указывает степень числа 10, на которую домножается мантисса. Порядок отделяется от мантиссы буквой “e” (от слова exponent), она может быть прописной или строчной. Порядок может иметь знак плюс или минус, в случае положительного порядка знак плюс можно опускать. Примеры:

```
1.5e+6      константа эквивалентна  1500000.0
1e-4        константа эквивалентна  0.0001
-.75E3      константа эквивалентна  -750.0
```

1.4.3. Символьные переменные

Значением символьной переменной является один символ из фиксированного набора. Такой набор обычно включает буквы, цифры, знаки препинания, знаки математических операций и различные специальные символы (процент, амперсенд, звездочка, косая черта и др.). Подчеркнем, что, в отличие от строковой переменной, символьная всегда содержит ровно один символ. (Строковая содержит строку из нескольких символов.)

Конечно, в памяти компьютера никаких символов не содержится. Символы представляются их целочисленными кодами в некоторой фиксированной кодировке. Кодировка определяется тремя параметрами:

1) диапазоном значений кодов. Например, самая распространенная в мире кодировка ASCII (от слов American Standard Code of Information Interchange — Американский стандартный код обмена информацией) имеет диапазон значений кодов от 0 до 127, т.е. требует семи бит на символ. Большинство современных кодировок имеют диапазон кодов от 0 до 255, т.е. один байт на символ. Наконец, сейчас во всем мире осуществляется переход на кодировку Unicode, которая использует коды в диапазоне от 0 до 65535, т.е. 2 байта на символ;

2) множеством изображаемых символов. Например, кодировка ASCII содержит буквы латинского алфавита, в западноевропейской кодировке к символам ASCII добавлены буквы с умлаутами и акцентами, дополнительные знаки препинания, в частности, испанские перевернутые вопросительные и восклицательные знаки, и другие символы европейских языков, основанных на латинской графике. Любая из русских кодировок содержит кириллицу;

3) отображением множества кодов на множество символов. Например, русские кодировки КОИ-8 (Код обмена информацией восьмибитовый) и “Windows CP-1251”, традиционно используемые в операционных системах Unix и MS Windows, имеют один и тот же диапазон кодов и один и тот же набор символов, но отображения их различны (одни и те же символы имеют

разные коды в кодировках КОИ-8 и Windows).

К сожалению, российские программисты не сумели договориться о единой кодировке русских букв. В настоящее время в России широко используются четыре различные кодировки:

1) кодировка КОИ-8 (это наиболее старый стандарт, принятый еще в конце 70-х годов XX века). КОИ-8 в основном используется в системе Unix и до недавнего времени была стандартом де-факто для русскоязычной электронной почты. Последнее время, однако, все чаще в электронной почте используют кодировку Windows;

2) так называемая альтернативная кодировка CP-866, которая используется в системе MS DOS. Она не удовлетворяет некоторым требованиям международных стандартов — например, ряд русских букв совпадает с кодами символов, используемых для управления передачей по линии. Альтернативная кодировка постепенно уходит в прошлое вместе с системой DOS;

3) кодировка Windows CP-1251, которая появилась значительно позже кодировки КОИ-8, но создатели русской версии Windows не захотели воспользоваться КОИ-8 (по-видимому, из-за того, что коды русских букв в КОИ-8 не упорядочены в соответствии с алфавитом; в CP-1251 коды русских букв упорядочены, за исключением буквы ё). В связи с распространением операционной системы Windows, кодировка Windows получает все большее распространение;

4) кодировка, используемую в компьютерах Apple Macintosh.

Существование различных кодировок русских букв сильно осложняет жизнь как программистам, так и обыкновенным пользователям: файлы при переносе из одной системы в другую приходится перекодировать, периодически возникают трудности при чтении писем, просмотре гипертекстовых страниц и т.п. Отметим, что ничего подобного нет ни в одной европейской стране.

С повсеместным переходом на кодировку Unicode все проблемы такого рода должны исчезнуть. Кодировка Unicode включает символы алфавитов всех европейских стран и кириллицу. К сожалению,

большинство существующих компьютерных программ приспособлено к представлению одного символа в виде одного байта. Поэтому в настоящее время часто используется промежуточное решение: компьютерные программы работают с внутренним представлением символов в кодировке Unicode (такое решение принято в языках Java и C#). При записи в файл символы Unicode приводятся к однобайтовой кодировке в соответствии с текущей языковой установкой. При этом, конечно, часть символов теряется — например, в кодировке Windows невозможно одновременно записать русские буквы и немецкие умлауты, поскольку умлауты в западно-европейской кодировке имеют те же коды, что и русские буквы в русской кодировке.

1.4.4. Логические переменные и выражения

Логические переменные принимают два значения: «истина» и «ложь». Логические, или условные, выражения используются в качестве условия в конструкциях ветвления “если ... то ... иначе ... конец если” и цикла “пока”. В первом случае в зависимости от истинности условия выполняется либо ветвь программы после ключевого слова “то”, либо после “иначе”; во втором случае цикл выполняется до тех пор, пока условие продолжает оставаться истинным.

В качестве элементарных условных выражений используются операции сравнения: можно проверить равенство двух выражений или определить, какое из них больше. Любая операция сравнения имеет два аргумента и вырабатывает логическое значение “истина” или “ложь” (true и false в языке C++). Мы будем обозначать операции сравнения так, как это принято в языке Си:

— операция проверки равенства двух выражений обозначается двойным знаком равенства == (мы не используем обычный знак равенства во избежание путаницы, поскольку часто знак равенства применяется для обозначения операции присваивания);

— неравенство $\neq$ обозначается != (в Си восклицательный знак используется для отрицания);

— для сравнения величин выражений применяются четыре операции больше >, больше или равно ‘>=, меньше <, меньше или равно <=.

Несколько примеров логических выражений:

x == 0 — выражение истинно, если значение переменной x равно нулю, и ложно в противном случае;

0 != 0 — выражение ложно;

3 >= 2 — выражение истинно.

Из элементарных логических выражений и логических переменных можно составлять более сложные выражения, используя три логические операции “и”, “или”, “не”:

1) результат логической операции “и” истинен, когда истинны оба ее аргумента. Например, логическое выражение

```
0 <= x и x <= 1
```

истинно, когда значение переменной x принадлежит отрезку [0, 1]. Логическую операцию “и” называют также логическим умножением или конъюнкцией; в языке Си логическое умножение обозначается двойным амперсандом &&;

2) результат логической операции “или” истинен, когда истинен хотя бы один из ее аргументов. Например, логическое выражение

```
x != 0 или y != 0
```

ложно в том и только том случае, когда значения обеих переменных x и y равны нулю. Логическую операцию “или” называют также логическим сложением или дизъюнкцией; в Си логическое сложение обозначается двойной вертикальной чертой ||;

3) в отличие от логических операций “и” и “или”, логическая операция “не” имеет только один аргумент. Ее результат истинен,

когда аргумент ложен, и, наоборот, ложен, когда аргумент истинен. Например, логическое выражение

```
не x == 0
```

истинно, когда значение переменной x отлично от нуля. Логическая операция "не" называется логическим отрицанием (иногда негацией); в Си логическое отрицание обозначается восклицательным знаком "!".

В сложных логических выражениях можно использовать круглые скобки для указания порядка операций. При отсутствии скобок считается, что наивысший приоритет имеет логическое отрицание; затем идет логическое умножение, а низший приоритет у логического сложения.

Обратим внимание на чрезвычайно важную особенность операций реализации логического сложения и умножения — так называемое сокращенное вычисление результата. А именно, в случае логического умножения всегда сначала вычисляется значение первого аргумента. Если оно ложно, то значение выражения полагается ложным, а второй аргумент не вычисляется вообще! Благодаря этой особенности можно корректно использовать выражения вроде

```
x != 0 и y/x > 1
```

При вычислении значения этого выражения сначала вычисляется первый аргумент конъюнкции "x != 0". Если значение переменной x равно нулю, то первый аргумент ложен и значение второго аргумента "y/x > 1" уже не вычисляется. Это очень хорошо, поскольку при попытке его вычислить произошло бы аппаратное прерывание из-за деления на ноль.

То же самое относится и к логическому сложению. Сначала всегда вычисляется первый аргумент логической операции "или". Если он истинен, то значение выражения полагается истинным, а второй аргумент не вычисляется вообще. Таким образом, операции логического сложения и умножения, строго говоря, не коммутативны. Может так случиться, что выражение "a и b" корректно, а выражение "b и a" — нет. Программисты очень часто сознательно используют эту особенность реализации логических операций.

1.4.5. Массивы

Кроме базовых типов, в большинстве алгоритмических языков присутствует конструкция *массив*. Иногда массив называют также *таблицей* или *вектором*. Массив позволяет объединить множество элементов одного типа в единую переменную.

Все элементы массива имеют один и тот же тип. Элементы массива обычно нумеруются индексами от 0 до $n-1$, где n — число элементов массива. В некоторых языках можно задавать границы изменения индексов, в других нижняя граница значения индекса равна единице, а не нулю. Мы, тем не менее, будем придерживаться языка Си (а также C++, Java, C#), в котором нижней границей индекса всегда является ноль. Это очень удобно, т.к. индекс элемента массива в этом случае равен его смещению относительно начала массива. Длина массива задается при его описании и не может быть изменена в процессе работы программы.

При описании массива указывается тип и число его элементов. Тип записывается перед именем массива, размер массива указывается в квадратных скобках после его имени. Примеры:

```
цел a[100];  описан массив целых чисел размера 100
             (индекс меняется от 0 до 99)
вещ r[1000]; описан вещ. массив из 1000 элементов.
```

В языке Си соответствующие описания выглядят следующим образом:

```
int a[100];
double r[1000];
```

Для доступа к элементу массива указывается его имя и в квадратных скобках — индекс нужного элемента. С элементом массива можно работать как с обычной переменной, т.е. можно прочитать его значение или записать в него новое значение. Примеры:

```
a[3] := 0;        элементу массива a с индексом 3
                  присваивается значение 0;
a[10] := a[10]*2;      элемент массива a с индексом
                       10 удваивается.
```

Массив — это самая важная конструкция алгоритмического языка. Важность массива определяется тем, что память компьютера логически представляет собой массив (его можно рассматривать как массив байтов или как массив четырехбайтовых машинных слов). Индекс в этом массиве обычно называют *адресом*. Элементы массива читаются и записываются исключительно быстро, за одно действие, независимо от размера массива и величины индекса. Для программиста конструкция массива как бы дана свыше. Большинство других структур данных, используемых в программировании, моделируются на базе массива.

1.4.6. Текстовые строки

Текстовые строки представляются массивами символов. Строковая переменная содержит на самом деле адрес этого массива. В отличие от символа, который занимает либо один, либо два байта в зависимости от используемой кодировки, строка имеет переменную длину. Существуют два способа указания длины строки:

1) строка заканчивается символом с нулевым кодом, т.е. либо нулевым байтом в случае однобайтового представления символов, либо двумя нулевыми байтами в случае двухбайтового представления. Такой способ принят в языке Си. Отметим, что нулевой байт — это вовсе не символ '0'! Символ '0' имеет код 48 в кодировках ASCII и UNICODE, а изображаемых символов с нулевым кодом не существует;

2) строка в качестве первого элемента (байта или двух байтов) содержит общее число символов, не включая начального элемента. Затем идут сами символы в указанном количестве. Такой способ используется в языке Паскаль.

Недостаток первого способа состоит в том, что для вычисления длины строки необходимо последовательно просмотреть все ее элементы, начиная с первого, пока не будет найден нулевой байт. Такая операция может быть долгой для длинной строки. Недостаток второго способа заключается в том, что длина строки ограничена. В случае однобайтовых символов максимальная длина строки равна 255,

т.е. максимальному числу, которое можно записать в одном байте. Длина строки двухбайтовых символов ограничена числом 65535.

Впрочем, существуют и другие способы представления строк, которые используются в объектно-ориентированных языках. Строка рассматривается как объект, внутреннее устройство которого скрыто от пользователя, хотя, как правило, он содержит массив или адрес массива символов и длину строки. Обычно в случае представления строк в виде объектов ограничения на длину строки отсутствуют.

1.5. Примеры алгоритмов

В этом разделе мы рассмотрим некоторые типичные примеры алгоритмов. Как фрагменты они встречаются практически во всех больших программах.

1.5.1. Вычисление функций на последовательностях

Как правило, компьютеры прежде всего используются для обработки значительных объемов информации. В большинстве алгоритмов информация читается последовательно, от начала к концу. Поэтому в программах очень часто встречаются фрагменты кода, вычисляющие некоторую функцию на последовательности элементов.

Предположим для простоты, что последовательность элементов находится в массиве. В реальных программах она также может читаться из файла или из более сложных структур данных — например, из списка. В любом случае можно встать в начало последовательности, прочесть ее очередной элемент, а также определить, есть ли еще непрочитанные элементы.

Рассмотрим пример: дана последовательность вещественных чисел, требуется вычислить сумму ее элементов. Запишем алгоритм на неформальном языке в самом общем виде.

```
вещ алгоритм сумма последовательности
| дано: последовательность вещественных чисел
| надо: вернуть сумму ее элементов
начало алгоритма
| вещ s, x;
```

```
| s := 0.0;
| встать в начало последовательности
| цикл пока есть непрочитанные элементы
| | прочесть очередной элемент последовательности в (вых:x)
| | s := s + x;
| конец цикла
| ответ := s;
конец алгоритма
```

Здесь перед словом алгоритм записывается тип возвращаемого значения, т.е. "вещ" для вещественного числа. Это означает, что алгоритм можно использовать как функцию. Например, для функции "sin" можно записать выражение

$$y = \sin(x).$$

При вычислении выражения сначала вызывается алгоритм (функция), вычисляющий sin, затем значение, которое возвращается этим алгоритмом, присваивается переменной y. В нашем случае можно использовать выражение

$$y := \text{сумма последовательности};$$

для вызова алгоритма и записи возвращаемого значения в переменную y.

В случае, когда последовательность чисел находится в массиве, алгоритм выглядит следующим образом:

```
вещ алгоритм сумма последовательности(вх: цел n, вещ a[n])
| дано: n    -- число элементов последовательности,
|       a[n] -- массив элементов последовательности
| надо: вернуть сумму элементов последовательности
начало алгоритма
| вещ s; цел i;
| s := 0.0;          // Инициализация значения ф-ции
| i := 0
| цикл пока i < n
| | s := s + a[i]; // Вычисление нового значения по старому
| |                //     значению и очередному элементу
```

```
| | i := i + 1;     // Переходим к следующему элементу
| конец цикла
| ответ := s;
конец алгоритма
```

Здесь целочисленная переменная i используется в качестве индекса элемента массива. Она последовательно принимает значения от 0 до $n-1$. Очередной элемент последовательности записывается как $a[i]$.

Отметим следующие составные части алгоритма, вычисляющего функцию на последовательности элементов:

1) инициализация значения функции для пустой последовательности — в данном случае

```
s := 0.0;
```

2) вычисление нового значения функции по прочтению очередного элемента последовательности. Новое значение вычисляется по старому значению и очередному прочитанному элементу. В данном случае это суммирование

```
s := s + a[i].
```

Эти две части присутствуют в любом алгоритме, вычисляющем функцию на последовательности.

Рассмотрим еще один важный пример: вычисление максимального элемента последовательности. В отличие от суммы элементов, здесь не вполне ясно, каким значением надо инициализировать максимум, то есть чему равен максимум пустой последовательности. Для того, чтобы максимум одноэлементной последовательности вычислялся правильно, надо, чтобы максимум пустой последовательности был меньше любого числа. Поэтому максимум пустой последовательности не может быть обычным числом. В математике и в программировании очень полезен следующий прием: к обычным элементам добавляется специальный воображаемый элемент с заданными свойствами. Так были изобретены ноль, отрицательные числа, иррациональные и комплексные числа и т.п. В данном случае, таким воображаемым элементом является «минус бесконечность».

```
вещ алгоритм максимум последовательности(вх: цел n, вещ a[n])
| дано: n    -- число элементов последовательности,
|       a[n] -- массив элементов последовательности
| надо: вернуть максимум элементов последовательности
начало алгоритма
| вещ m; цел i;
| m := минус бесконечность; // Инициализация значения ф-ции
| i := 0;
| цикл пока i < n
| | если a[i] > m   // Вычисление нового значения по
| | | то m := a[i]; // старому значению и очередному эл-ту
| | конец если
| | i := i + 1;
| конец цикла
| ответ := m;
конец алгоритма
```

Здесь переменная m на любом шаге содержит максимальное значение для просмотренного начального отрезка последовательности, т.е. кандидата на максимум. Если очередной элемент больше, чем m, то он запоминается в переменной m и становится новым кандидатом на максимум.

Значение «минус бесконечность» в случае операции взятия максимума двух чисел обладает следующим замечательным свойством: для всякого числа x выполняется равенство

$$\max(\text{минус бесконечность}, x) = x.$$

Можно сравнить с операцией сложения:

$$0 + x = x.$$

Таким образом, значение «минус бесконечность» играет роль нуля для операции взятия максимума двух чисел. Ноль — это нейтральный элемент для операции сложения: будучи прибавленным слева к произвольному числу x, он не изменяет числа x. Точно так же значение «минус бесконечность» является нейтральным для операции взятия максимума.

Для операции «минимум» нейтральным элементом является «плюс бесконечность». Таким образом, алгоритм нахождения минимума последовательности выглядит следующим образом:

```
вещ алгоритм минимум последовательности(вх: цел n, вещ a[n])
| дано: n    -- число элементов последовательности,
|       a[n] -- массив элементов последовательности
| надо: вычислить минимум элементов последовательности
начало алгоритма
| вещ m; цел i;
| m := плюс бесконечность; // Инициализация значения ф-ции
| i := 0;
| цикл пока i < n
| | если a[i] < m   // Вычисление нового знач. по старому
| | | то m := a[i]; // значению и очередному элементу
| | конец если
| | i := i + 1;
| конец цикла
| ответ := m;
конец алгоритма
```

Значения «минус» и «плюс бесконечность»

Как реализовать воображаемые элементы «минус бесконечность» и «плюс бесконечность» при программировании на конкретных алгоритмических языках, а не на псевдокоде? Вспомним, что компьютер может представлять не все возможные числа, а только их ограниченное подмножество. Поэтому для компьютера существует минимальное и максимальное целое и вещественное числа. В языке Си эти константы записаны в стандартных заголовочных файлах “limits.h” для целочисленных типов и “float.h” для вещественных типов. Для типа int эти константы называются INT_MIN и INT_MAX.

```
INT_MIN = (-2147483647 - 1)
INT_MAX = 2147483647
```

Для вещественных типов максимальное и минимальное числа равны по абсолютной величине и отличаются лишь знаками, поэтому специального названия для максимальной по абсолютной величине отрицательной константы не существует. Максимальное число типа float называется FLT_MAX, типа double — DBL_MAX.

```
FLT_MAX = 3.402823466e+38
DBL_MAX = 1.7976931348623158e+308
```

Стоит отметить, что через FLT_MIN и DBL_MIN обозначены *минимальные положительные* числа, а вовсе не максимальные по абсолютной величине отрицательные!

```
FLT_MIN = 1.175494351e-38
DBL_MIN = 2.2250738585072014e-308
```

Константа DBL_MAX является нормальным числом, она не равна специальному бесконечно большому значению, см. с. 1.4.2. Использовать бесконечно большое значение опасно, т.к. операции с ним могут приводить к к ошибкам.

Итак, в качестве значений «минус бесконечность» и «плюс бесконечность» можно использовать константы INT_MIN и INT_MAX для типа int. Для типа double в качестве значений «минус бесконечность» и «плюс бесконечность» можно использовать выражения (-DBL_MAX) и DBL_MAX. Не забудьте только при программировании на Си подключить стандартные заголовочные файлы:

```
#include <limits.h>
```

для целых типов и

```
#include <float.h>
```

для вещественных. Впрочем, вовсе не обязательно помнить названия этих констант и имена стандартных заголовочных файлов. В качестве значения «минус бесконечность» всегда можно использовать произвольное значение, заведомо меньшее, чем любое конкретное число, которое может встретиться в программе. Например, если известно, что программа работает только с неотрицательными числами, то в качестве значения «минус бесконечность» можно использовать произвольное отрицательное число, например, минус единицу. Аналогично, в качестве значения «плюс бесконечность» можно применять любое достаточно большое число. Оно должно быть заведомо больше, чем все конкретные числа, которые могут встретиться в алгоритме. Пусть, например, известно, что в программе могут встретиться вещественные числа не больше миллиона. Тогда в качестве значения «плюс бесконечность» можно использовать константу

```
1.0e+30
```

т.е. десять в тридцатой степени. (Можно даже использовать 1.0e+7, т.е. десять миллионов, но не стоит мелочиться.)

Схема Горнера

Рассмотрим еще один важный пример функции на последовательности. Пусть дана последовательность коэффициентов многочлена p(x) по убыванию степеней:

$$p(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n.$$

Нужно вычислить значение многочлена в точке $x = t$. Алгоритм, основанный на просмотре последовательности коэффициентов в направлении от старшего к младшему, называется схемой Горнера. Проиллюстрируем его идею на примере многочлена третьей степени:

$$p(x) = ax^3 + bx^2 + cx + d.$$

Его можно представить в виде

$$p(x) = ((ax + b)x + c)x + d.$$

Для вычисления значения многочлена достаточно трех умножений и трех сложений. В общем случае, многочлен представляется в следующем виде:

$$p(x) = (\ldots((a_0x + a_1)x + a_2)x + \ldots + a_{n-1})x + a_n.$$

Обозначим через $p_k(x)$ многочлен k-ой степени, вычисленный по коэффициентам a_0, a_1, ..., a_k:

$$p_k(x) = a_0x^k + a_1x^{k-1} + \ldots + a_k.$$

Тогда

$$p_{k+1} = p_k(x)x + a_{k+1},$$

т.е. при считывании нового коэффициента многочлена надо старое значение многочлена умножить на значение x, а затем прибавить к нему новый коэффициент.

Выпишем алгоритм:

```
вещ алгоритм схема Горнера(вх: цел n, вещ a[n+1], вещ t)
| дано: n      -- степень многочлена
|       a[n+1] -- массив коэффициентов многочлена по
```

```
|                  убыванию степеней
| надо: вычислить значение многочлена в точке t
начало алгоритма
| вещ p; цел i;
| p := 0.0;      // Инициализация значения многочлена
| i := 0;
| цикл пока i <= n
| | p := p * t + a[i]; // Вычисление нового значения по
| |     // старому значению и добавленному коэффициенту
| | i := i + 1;
| конец цикла
| ответ := p;
конец алгоритма
```

Арифметический цикл

В рассмотренных выше программах в цикле перебираются элементы массива с индексом i, где i пробегает значения от 0 до $n-1$ (в последней программе — от 0 до n, поскольку многочлен n-й степени имеет $n+1$ коэффициент). Для удобства записи таких циклов большинство языков программирования предоставляет конструкцию арифметического цикла. В нем используется так называемая *переменная цикла*, т.е. целочисленная переменная, которая последовательно принимает значения в указанных пределах. Для каждого значения переменной цикла выполняется тело цикла, в котором эта переменная может использоваться.

```
цикл для i от a до b
| . . .
| тело цикла
| . . .
конец цикла
```

Здесь переменная цикла i последовательно принимает значения от a до b с шагом 1, где a и b — некоторые целочисленные выражения. Таким образом, всего тело цикла выполняется $b-a+1$ раз. Если b меньше, чем a, то цикл не выполняется ни разу. Возможна также конструкция арифметического цикла с шагом s, отличным от единицы:

```
цикл для i от a до b шаг s
| . . .
| тело цикла
| . . .
конец цикла
```

Переменная цикла последовательно принимает значения a, $a+s$, $a+2s$, ... до тех пор, пока ее значение содержится в отрезке $[a,b]$. Для каждого значения переменной цикла выполняется тело цикла. Шаг может быть и отрицательным, в этом случае b должно быть не больше, чем a, иначе цикл не выполняется ни разу.

В принципе, без конструкции арифметического цикла можно обойтись, поскольку ее можно смоделировать с помощью цикла "пока". А именно, конструкция

```
цикл для i от a до b
| . . .
| тело цикла
| . . .
конец цикла
```

эквивалентна конструкции

```
i := a
цикл пока i <= b
| . . .
| тело цикла
| . . .
| i := i + 1
конец цикла
```

Однако, традиционно арифметический цикл включается в большинство языков высокого уровня. С использованием арифметического цикла схема Горнера переписывается следующим образом:

```
вещ алгоритм схема Горнера(вх: цел n, вещ a[n+1], вещ t)
| дано: n      -- степень многочлена
|       a[n+1] -- массив коэффициентов многочлена по
|                  убыванию степеней
| надо: вычислить значение многочлена в точке t
```

```
начало алгоритма
| вещ p; цел i;
| p := 0.0;      // Инициализация значения многочлена
| цикл для i от 0 до n
| | p := p * t + a[i]; // Вычисление нового значения
| |                    // при добавлении коэффициента
| конец цикла
| ответ := p;
конец алгоритма
```

Аналогично можно переписать и другие приведенные выше алгоритмы вычисления функций на последовательностях. Приведем также пример использования арифметического цикла с отрицательным шагом. Пусть коэффициенты многочлена заданы по возрастанию, а не по убыванию степеней. В схеме Горнера следует просматривать коэффициенты многочлена от старшего к младшему. Для этого удобно использовать арифметический цикл с отрицательным шагом:

```
вещ алгоритм схема Горнера2(вх: цел n, вещ b[n+1], вещ t)
| дано: n      -- степень многочлена
|       b[n+1] -- массив коэффициентов многочлена по
|                  возрастанию степеней
| надо: вычислить значение многочлена в точке t
начало алгоритма
| вещ p; цел i;
| p := 0.0;      // Инициализация значения многочлена
| цикл для i от n до 0 шаг -1
| | p := p * t + b[i]; // Вычисление нового значения
| |                    // при добавлении коэффициента
| конец цикла
| ответ := p
конец алгоритма
```

Индуктивные функции на последовательностях и индуктивные расширения

В рассмотренных выше примерах при добавлении к последовательности еще одного элемента новое значение функции на последовательности можно было вычислить, зная только старое значение

функции и добавленный элемент. Обозначим через S_n последовательность

$$S_n = \{a_0, a_1, \ldots, a_{n-1}\}$$

длины n. С помощью знака & обозначим операцию приписывания нового элемента справа к последовательности (ее называют также конкатенацией):

$$S_{n+1} = S_n \,\&\, a_n = \{a_0, a_1, \ldots, a_{n-1}, a_n\}.$$

Пусть $f(S)$ — некоторая функция на множестве последовательностей, например, сумма элементов последовательности. Функция называется индуктивной, если при добавлении нового элемента к последовательности новое значение функции можно вычислить, зная только старое значение функции и добавленный элемент. На математическом языке функция

$$f : W \to Y,$$

где W — множество всех последовательностей, составленных из элементов некоторого множества X, индуктивна, если существует функция G от двух аргументов

$$G : Y \times X \to Y$$

такая, что для любой последовательности S из W и любого элемента a из X значение функции f на последовательности S, к которой добавлен элемент a, вычисляется с помощью функции G:

$$f(S \,\&\, a) = G(f(S), a).$$

Функция G по паре (y, a), где y — старое значение функции f на последовательности S и a — элемент, добавленный к последовательности, вычисляет новое значение y, равное значению функции f на новой последовательности.

В примере с суммой элементов последовательности функция G равна сумме элементов y и a:

$$G(y, a) = y + a.$$

В примере с максимальным элементом последовательности функция G равна максимуму:

$$G(y, a) = \max(y, a).$$

В примере со схемой Горнера вычисления значения многочлена в точке t, где коэффициенты многочлена заданы в последовательности по убыванию степеней, функция G равна

$$G(y, a) = yt + a.$$

Во всех трех случаях рассматриваемая функция на последовательности индуктивна.

Общая схема вычисления значения индуктивной функции на последовательности выглядит следующим образом:.

```
алгоритм значение индуктивной функции(
  вх: последовательность S
)
| дано: последовательность S
| надо: вычислить функцию y = f(S)
начало алгоритма
| y := значение функции f на пустой последовательности;
| встать в начало последовательности S;
| цикл пока в последовательности S есть
| |           непрочитанные элементы
| | прочесть очередной элемент
| |     последовательности S в (вых:x);
| | y := G(y, x);
| конец цикла
| ответ := y;
конец алгоритма
```

Таким образом, для каждой конкретной индуктивной функции надо лишь правильно задать ее значение на пустой последовательности (инициализация) и определить, как новое значение функции вычисляется через старое при добавлении к последовательности очередного элемента, т.е. задать функцию $G(y, x)$. Схема вычисления для всех индуктивных функций одна и та же.

Однако, не все функции на последовательностях являются индуктивными. Рассмотрим следующий пример. Пусть коэффициенты многочлена заданы в последовательности по убыванию степеней. Надо вычислить значение производной многочлена в точке $x = 2$. Обозначим через

$$S = \{a_0, a_1, \ldots, a_n\}$$

последовательность коэффициентов многочлена

$$p(x) = a_0 x^n + a_1 x^n - 1 + \ldots + a_n$$

и через $f(S)$ значение производной многочлена $p'(x)$ в точке $x = 2$:

$$f(S) = p'(2).$$

Покажем, что функция f не индуктивна. Достаточно указать две последовательности S_1 и S_2, такие, что значения функции f на них совпадают, но при добавлении к последовательностям S_1 и S_2 одного и того же элемента a новые значения функции уже не равны:

$$\begin{aligned} &f(S_1) = f(S_2), \\ &f(S_1 \,\&\, a) \neq f(S_2 \,\&\, a). \end{aligned}$$

Возьмем последовательности

$$\begin{aligned} S_1 &= \{1\}, \\ S_2 &= \{1, -4, 1\}. \end{aligned}$$

Им соответствуют многочлены

$$\begin{aligned} p_1(x) &= 1, \\ p_2(x) &= x^2 - 4x + 1. \end{aligned}$$

Производные многочленов равны

$$\begin{aligned} p_1'(x) &= 0, \\ p_2'(x) &= 2x - 4. \end{aligned}$$

Значения обеих производных в точке $x = 2$ равны нулю, т.е.

$$\begin{aligned} f(S_1) &= p_1'(2) = 0, \\ f(S_2) &= p_2'(2) = 2 \cdot 2 - 4 = 0. \end{aligned}$$

Припишем теперь к обеим последовательностям элемент $a = 1$:

$$S_1 \,\&\, 1 = \{1, 1\},$$
$$S_2 \,\&\, 1 = \{1, -4, 1, 1\}.$$

Новым последовательностям соответствуют многочлены

$$q_1(x) = x + 1,$$
$$q_2(x) = x^3 - 4x^2 + x + 1.$$

Их производные равны

$$q_1'(x) = 1,$$
$$q_2'(x) = 3x^2 - 8x + 1.$$

Значения производных в точке $x = 2$ равны соответственно

$$f(S_1 \,\&\, 1) = q_1'(2) = 1,$$
$$f(S_2 \,\&\, 1) = q_2'(2) = 12 - 16 + 1 = -3.$$

Мы видим, что значения $f(S_1)$ и $f(S_2)$ совпадают, но значения $f(S_1 \,\&\, 1)$ и $f(S_2 \,\&\, 1)$ не совпадают. Следовательно, функция f не индуктивна.

Как поступать в случае, когда функция f не индуктивна? Общий рецепт следующий: надо придумать индуктивную функцию F, такую, что, зная значение F, легко можно вычислить исходную функцию f. Функция F называется индуктивным расширением функции f.

Приведем формальные определения. Пусть исходная функция на множестве W всех последовательностей

$$f : W \to Y$$

не индуктивна. Индуктивная функция

$$F : W \to Z$$

называется индуктивным расширением функции f, если существует отображение

$$P : Z \to Y$$

такое, что для всякой последовательности S, принадлежащей W, выполняется равенство

$$f(S) = P(F(S))$$

(т.е. функция f равна композиции отображений F и P, $f = P \circ F$.) Отображение P обычно называют проекцией множества Z на Y.

Как построить индуктивное расширение функции f? Это творческий момент, готового рецепта на все случаи не существует. Неформальный рецепт следующий: надо понять, какой информации не хватает для того, чтобы уметь вычислять новое значение последовательности при добавлении к ней нового элемента. Эту информацию надо хранить дополнительно к значению последовательности. Отсюда и появился термин «расширение»: вычисляется более сложная, расширенная, функция, чтобы по ней затем восстановить исходную. Как правило, значением индуктивного расширения F является пара (y, h), где y — значение исходной функции f, а h — некоторая дополнительная информация, позволяющая перевычислять значение y при добавлении нового элемента к последовательности. Таким образом, множество Z значений индуктивного расширения

$$F : W \to Z$$

чаще всего является множеством пар (y, h), т.е. декартовым произведением:

$$Z = Y \times H.$$

Отображение P на практике должно легко вычисляться. Так оно и есть в случае декартого произведения — это просто проекция на первый аргумент.

$$P(y, h) = y.$$

Рассмотрим пример с вычислением производной многочлена в точке; коэффициенты многочлена заданы в последовательности по убыванию степеней. При добавлении к последовательности

$$S_k = \{a_0, a_1, \ldots, a_k\}$$

нового коэффициента a_{k+1} получаем последовательность

$$S_{k+1} = S_k \,\&\, a_{k+1} = \{a_0, a_1, \ldots, a_k, a_{k+1}\}.$$

Пусть этим двум последовательностям соответствуют многочлены $p_k(x)$ и $p_{k+1}(x)$. Тогда

$$p_{k+1}(x) = p_k(x) \cdot x + a_{k+1}.$$

Дифференцируя это равенство, получим:

$$p'_{k+1}(x) = p'_k(x) \cdot x + p_k.$$

Мы видим, что для вычисления нового значения производной нужно знать старое значение производной, а также старое значение многочлена. Следовательно, дополнительно к значению производной многочлена надо хранить еще значение самого многочлена. Таким образом, индуктивным расширением функции, равной производной многочлена в точке t, является пара (значение производной, значение многочлена):

$$F : S \mapsto (p'(t), p(t)).$$

Новое значение производной вычисляется по приведенной выше формуле через старое значение производной и старое значение многочлена. После этого вычисляется новое значение многочлена по схеме Горнера.

Выпишем алгоритм вычисления производной многочлена.

```
вещ алг. значение производной(вх: цел n, вещ a[n+1], вещ t)
| дано: n      -- степень многочлена
|       a[n+1] -- массив коэффициентов многочлена по
|                  возрастанию степеней
| надо: найти значение производной многочлена в точке t
начало алгоритма
| вещ p, dp; цел i;
| p := 0.0;     // Инициализация значения многочлена
| dp := 0.0;    // Инициализация значения производной
| цикл для i от 0 до n
| | dp := dp * x + p;  // Новое значение производной
| | p := p * t + a[i]; // Новое значение многочлена
| конец цикла
| ответ := dp;
конец алгоритма
```

Другой пример неиндуктивной функции — это среднее арифметическое значение элементов последовательности. Индуктивным расширением является пара (сумма элементов последовательности, длина последовательности):

$$F(S) = (\text{сумма}(S), \text{длина}(S)).$$

Легко видеть, что функция F индуктивна. При известном значении функции F не составляет труда вычислить исходную функцию:

$$\text{среднее арифметическое}(S) = \text{сумма}(S)/\text{длина}(S).$$

В данном случае отображение P не является в чистом виде проекцией, т.к. в процессе вычислений удобнее хранить сумму элементов прочитанного отрезка последовательности, а не среднее арифметическое. Вычисления проще и, кроме того, сумма определена на пустой последовательности в отличие от среднего арифметического.

Итак, в каждом конкретном случае при вычислении неиндуктивной функции f надо придумать ее индуктивное расширение F и в программе вычислять сначала индуктивное расширение F, а затем по значению F вычислять требуемое значение исходной функции f.

Задачи по теме «Индуктивные функции»

1. Указать, какие из нижеперечисленных функций на последовательностях являются индуктивными. В каждом случае надо привести доказательство индуктивности или неиндуктивности функции. Для неиндуктивных функций построить их индуктивные расширения:

   — произведение элементов последовательности;

   — число максимальных элементов последовательности;

   — число локальных максимумов последовательности (элемент является локальным максимумом, если он не меньше своих соседей);

   — сумма первого и последнего элементов последовательности;

   — число перемен знака в последовательности;

   — значение многочлена, коэффициенты которого заданы по возрастанию степеней, в точке $x = 1$;

   — та же задача при $x = -1$;

   — вторая производная многочлена, коэффициенты которого заданы по убыванию степеней, в точке $x = 2$.

2. Построить индуктивное расширение и выписать алгоритм вычисления функции, которая последовательности коэффициентов многочлена по убыванию степеней ставит в соответствие значение k-ой производной многочлена в точке t, где k — фиксированное натуральное число.

1.5.2. Построение цикла с помощью инварианта

Правильное использование конструкции цикла всегда представляет некоторую трудность. Применение элементарной теории помогает избежать ошибок и облегчает написание сложных программ.

Основная идея состоит в следующем. В процессе выполнения цикла изменяются значения набора переменных. Надо найти соотношение между меняющимися переменными, которое остается постоянным. Это соотношение называется инвариантом цикла. Сознательное построение цикла "пока" всегда связано с явной формулировкой и использованием инварианта цикла.

Явная формулировка инварианта помогает выписать инициализацию переменных, выполняемую до начала цикла, и тело цикла. Инициализация должна обеспечить выполнение инварианта до начала работы цикла. Тело цикла должно быть сконструировано таким образом, чтобы обеспечить сохранение инварианта. (Более точно, из того, что инвариант выполняется до начала исполнения тела цикла, должно следовать выполнение инварианта после окончания тела цикла. В процессе исполнения тела цикла инвариант может нарушаться.)

Завершение цикла, как правило, связано с ограниченной величиной, которая монотонно возрастает или монотонно убывает при каждом выполнении тела цикла. Цикл "пока" завершается, когда условие после слова "пока" в заголовке цикла становится ложным. Следовательно, это условие должно прямо или косвенно зависеть от величины, монотонно убывающей или возрастающей в процессе выполнения цикла. По достижению ее определенного значения условие должно становиться ложным. Условием завершения цикла называют отрицание условия, стоящего после слова "пока" в заголовке цикла.

Выполнение инварианта цикла и одновременно условия завершения должно обеспечивать решение требуемой задачи.

Общая схема

Обозначим через X множество всевозможных наборов значений всех переменных, меняющихся в ходе выполнения цикла. Множество X иногда называют фазовым, или конфигурационным, пространством задачи. Инвариант — это некоторое условие $I(x)$, зависящее от точки x из множества X и принимающее значение "истина" или "ложь". (Математики называют такие условия предикатами.) В процессе инициализации точке x присваивается такое значение x_0, что условие $I(x_0)$ истинно.

Обозначим условие завершения цикла через $Q(x)$. Условия $I(x)$ и $Q(x)$ должны быть подобраны таким образом, чтобы одновременная истинность $I(x)$ и $Q(x)$ обеспечивала решение требуемой задачи: нахождение точки x с требуемыми свойствами.

Тело цикла можно трактовать как отображение точки x в новую точку $T(x)$ из того же множества X:

$$T : X \to X.$$

Условие $I(x)$ является инвариантом для отображения T: если $I(x)$ истинно, то $I(T(x))$ также истинно.

Общая схема построения цикла с помощью инварианта выглядит следующим образом:

```
x := x0;     // x0 выбирается так, чтобы условие
             //                 I(x0) было истинным
утверждение: I(x);

цикл пока не Q(x)
| инвариант: I(x);
| x := T(x);  // точка x преобразуется в T(x)
конец цикла

утверждение: Q(x) и I(x);
ответ := x;
```

Конечно, эта схема не имеет никакой ценности без умения применять ее на практике. Рассмотрим несколько важных примеров ее использования.

Алгоритм Евклида вычисления наибольшего общего делителя

Пусть даны два целых числа m и n, хотя бы одно из которых не равно нулю. Требуется найти их наибольший общий делитель. Напомним, что наибольшим общим делителем двух чисел m и n называется такой их общий делитель d, который делится на любые другие общие делители d'. Такое определение НОД подходит не только для чисел, но и для многочленов, поскольку в нем не используется сравнение по величине. Наибольший общий делитель определен с точностью до обратимого множителя; в частности, поскольку в кольце чисел обратимы только элементы ± 1, НОД целых чисел определен с точностью до знака.

В качестве пространства X рассматривается множество пар целых чисел

$$X = \{(a, b) |\ a, b \in \mathbf{Z},\ a \neq 0 \text{ или } b \neq 0\}$$

Надо вычислить НОД для заданной пары чисел (m, n). В качестве инварианта используем утверждение, что НОД текущей пары чисел равен НОД исходной пары:

$$I(a, b): \quad \text{НОД}(a, b) = \text{НОД}(m, n).$$

Следовательно, цикл надо строить таким образом, чтобы при изменении переменных a, b наибольший общий делитель пары (a, b) оставался неизменным. В качестве начальной точки x_0 используется пара (m, n).

Обозначим через r остаток от деления a на b:

$$a = qb + r, \quad \text{где } |r| < |b|.$$

Тогда нетрудно доказать, что $\text{НОД}(b, r) = \text{НОД}(a, b)$. Достаточно показать, что множества общих делителей пары (b, r) и пары (a, b) совпадают. Пусть d делит b и r. Тогда из равенства $a = qb + r$ вытекает, что b делит a. Обратно, пусть d делит a и b. Из определения остатка имеем:

$$r = a - qb.$$

Так как правая часть равенства делится на d, то r тоже делится на d.

Итак, при замене пары (a, b) на пару (b, r) НОД не меняется. Обозначим через T отображение

$$T : (a, b) \to (b, r).$$

Условие $I(a, b)$ является инвариантным для отображения T.

Осталось только определить условие завершения цикла $Q(a, b)$. Выполнение этого условия должно обеспечивать решение задачи, т.е. нахождение НОД чисел a, b. Для какой пары чисел их НОД можно сразу вычислить? Проще всего, когда одно из чисел равно нулю. В этом случае

$$\text{НОД}(a, 0) = a.$$

Итак, в качестве условия завершения цикла используем условие, что вторая компонента пары (a, b) нулевая:

$$Q(a, b) : \quad b = 0.$$

Теперь можно выписать алгоритм нахождения наибольшего общего делителя:

```
цел алгоритм НОД(вх: цел m, цел n)
| дано: целые числа m, n, хотя бы одно отлично от нуля
| надо: вычислить наибольший общий делитель пары (m, n)
начало алгоритма
| цел a, b, r;
| // инициализация
| a := m; b := n;
| утверждение: НОД(a, b) == НОД(m, n);
|
| цикл пока b != 0
| | инвариант: НОД(a, b) == НОД(m, n)
| | r := a % b;      // находим остаток от деления a на b
| | a := b; b := r; // заменяем пару (a, b) на (b, r)
| конец цикла
|
| утверждение: b == 0 и НОД(a, b) == НОД(m, n);
| ответ := a;
конец алгоритма
```

Алгоритм Евклида — один из самых замечательных алгоритмов теории чисел и программирования. Работает он исключительно быстро, за время, линейно зависящее от длины записи входных чисел. (Действительно, легко показать, что за два выполнения тела цикла число b уменьшается не менее чем в четыре раза. Следовательно, число выполнений тела цикла в худшем случае равно длине двоичной записи максимального из чисел a, b.) Это позволяет применять алгоритм Евклида к очень большим целым числам — например, к двухсотзначным десятичным. Алгоритм Евклида (более точно, расширенный алгоритм Евклида, который будет рассмотрен ниже) применяется для таких больших чисел в схеме кодирования с открытым ключом RSA, которая в настоящее время широко используется на практике для защиты информации.

Быстрое возведение в степень

Второй важнейший алгоритм элементарной теории чисел — это алгоритм быстрого возведения в степень. Наряду с алгоритмом Евклида, он встречается буквально на каждом шагу, когда речь идет о применении теории чисел в программировании, — например, в теории кодирования.

Пусть требуется возвести элемент a в целую неотрицательную степень n. В качестве a может фигурировать целое или вещественное число, квадратная матрица, элемент кольца вычетов по модулю m и т.п. — требуется только, чтобы элемент a принадлежал алгебраической структуре, в которой определена ассоциативная операция умножения (т.е. в общем случае, a — элемент полугруппы).

Идея алгоритма состоит в том, чтобы возвести a в произвольную степень, применяя элементарные операции возведения в квадрат и умножения.

В качестве фазового пространства X этой задачи рассмотрим множество троек

$$X = \{(b, k, p)\}.$$

Здесь b выступает в роли текущего основания степени, k — в роли текущего показателя степени, p — это уже вычисленная «часть» степени. Ключевым моментом всегда является формулировка инва-

рианта цикла:

$$I(b,k,p): \quad b^k \cdot p = a^n = \text{const},$$

т.е. величина $b^k \cdot p$ постоянна и равна a^n. Легко подобрать начальные значения так, чтобы инвариант выполнялся:

$$b_0 = a; \quad k_0 = n; \quad p_0 = 1.$$
$$I(b_0,\ k_0,\ p_0) = I(a,\ n,\ 1): \quad a^n \cdot 1 = a^n$$

Условие завершения совместно с выполнением инварианта должно обеспечить легкое решение требуемой задачи, т.е. вычисление a^n. Действительно, если $k = 0$, то из инварианта следует, что

$$b^0 \cdot p = p = a^n,$$

т.е. искомая величина содержится в переменной p. Итак, условие завершения состоит в равенстве нулю числа k:

$$Q(b,k,p): \quad k = 0.$$

Осталось написать преобразование T точки $x = (b,k,p)$, которое сохраняет инвариант и одновременно уменьшает k. Определим преобразование T следующим образом:

$$T(b,k,p) = \begin{cases} (b \cdot b,\ k/2,\ p), & \text{если } k \text{ четное} \\ (b,\ k-1,\ p \cdot b), & \text{если } k \text{ нечетное} \end{cases}$$

Легко видеть, что инвариант сохраняется и k монотонно убывает. Итак, выпишем алгоритм быстрого возведения в степень для случая вещественного основания:

```
вещ алг. быстрое возведение в степень(вх: вещ a, цел n)
| дано: основание a и показатель степени n >= 0
| надо: вычислить a в степени n
начало алгоритма
| вещ b, p; цел k;
|
| // инициализация
| b := a; p := 1.0; k := n;
| утверждение: b^k * p == a^n;
|
```

```
| цикл пока k > 0
| | инвариант: b^k * p == a^n;
| | если k четное
| | | то
| | |   k := k / 2;
| | |   b := b * b;
| | | иначе
| | |   k := k - 1;
| | |   p := p * b;
| | конец если
| конец цикла
|
| утверждение: k == 0 и b^k * p == a^n;
| ответ := p;
конец алгоритма
```

Вычисление логарифма без использования разложения в ряд

Схема построения цикла с помощью инварианта позволяет легко написать алгоритм вычисления логарифма заданного числа без использования разложения в ряд.

Пусть задано вещественное число x. Требуется вычислить логарифм числа x по основанию a с точностью ε, где ε — некоторое положительное очень маленькое число. Для определенности, пусть $a > 1$ (для $a < 1$ можно воспользоваться тождеством $\log_{1/a} x = -\log_a x$).

Из определения логарифма следует, что надо найти число y такое, что

$$a^y = x.$$

Нам достаточно, чтобы это равенство выполнялось приближенно. В качестве инварианта используем условие

$$a^y z^t = x = \text{const}.$$

Таким образом, в цикле будут меняться три переменные

$$(y, z, t),$$

и инвариант записывается в виде

$$I(y,z,t): \quad a^y z^t = x.$$

Начальные значения переменных y, z, t выбираются так, чтобы выполнялся инвариант:

$$y_0 = 0, \quad z_0 = x, \quad t_0 = 1.$$

Определим условие завершения цикла $Q(y,z,t)$. Необходимо, чтобы искомая величина по окончанию цикла содержалась в переменной y. Следовательно, величина z^t должна быть близка к единице: тогда приблизительно выполняется равенство

$$a^y \approx a^y z^t = x,$$

т.е. y приближенно равен искомому логарифму. Для того, чтобы величина z^t была близка к единице, нужно, чтобы показатель степени t был близок к нулю, а основание z было не очень велико и не очень мало. Для этого достаточно выполнения трех неравенств

$$|t| < \varepsilon, \quad 1/a < z < a.$$

Можно доказать строго, что при выполнении этих неравенств, а также условия $a^y z^t = x$, величина y отличается от $\log_a x$ не больше чем на ε.

Выполнение этих трех неравенств и являются условием завершения цикла:

$$Q(y,z,t): \quad |t| < \varepsilon \quad \text{и} \quad 1/a < z \quad \text{и} \quad z < a.$$

Наконец, тело цикла должно преобразовывать переменные (y,z,t) так, чтобы абсолютная величина t монотонно убывала, а переменная z рано или поздно попадала бы в интервал $(1/a,\, a)$, и при этом сохранялся инвариант. Такое преобразование T легко выписывается по инварианту цикла:

$$T(y,z,t) = \begin{cases} (y+t,\ z/a,\ t), & \text{если } z \geq a \\ (y-t,\ z \cdot a,\ t), & \text{если } z \leq 1/a \\ (y,\ z \cdot z,\ t/2), & \text{если } 1/a < z < a \end{cases}$$

Заметим, что при применении преобразования T некоторая величина как бы перетекает из одних переменных в другие, при этом равенство $a^y z^t = x$ остается неизменным.

Теперь можно выписать алгоритм вычисления логарифма:

```
вещ алгоритм логарифм(вх: вещ x, вещ a, вещ eps)
| дано: x > 0, a > 1, eps > 0
| надо: вычислить log_a x с точностью eps
начало алгоритма
| вещ y, z, t;
|
| // инициализация
| y := 0.0; z := x; t := 1.0;
| утверждение: a^y * z^t == x;
|
| цикл пока |t| >= eps или z <= 1.0/a или z >= a
| | инвариант: a^y * z^t == x;
| | если z >= a
| | | то
| | |   z := z/a; y := y + t;
| | иначе если z <= 1.0/a
| | | то
| | |   z := z*a; y := y - t;
| | иначе
| | |   z := z*z; t := t/2.0;
| | конец если
| конец цикла
|
| утверждение: |t| < eps  и
|              z > 1.0/a  и  z < a  и
|              a^y * z^t == x;
| ответ := y;
конец алгоритма
```

Расширенный алгоритм Евклида

Один из важнейших результатов элементарной теории чисел утверждает, что наибольший общий делитель двух целых чисел вы-

ражается в виде их линейной комбинации с целыми коэффициентами. Пусть m и n — два целых числа, хотя бы одно из которых не равно нулю. Тогда их наибольший общий делитель $d = \text{НОД}(m, n)$ выражается в виде

$$d = um + vn,$$

где u и v — некоторые целые числа. Результат этот очень важен для практики, т.к. позволяет вычислить обратный элемент к n в кольце вычетов по модулю m. Действительно, пусть числа m и n взаимно просты, т.е. $\text{НОД}(m, n) = 1$. Тогда

$$1 = um + vn,$$

откуда следует

$$\begin{aligned} vn &= 1 - um \qquad \Rightarrow \\ vn &\equiv 1 \pmod{m} \end{aligned}$$

Нахождение обратного элемента в кольце вычетов $\mathbf{Z}_m$ применяется во многих дискретных алгоритмах, например, в схеме кодирования с открытым ключом.

Для вычисления наибольшего общего делителя d и одновременно чисел u и v используется так называемый *расширенный алгоритм Евклида*. В обычном алгоритме Евклида пара чисел (a, b) в цикле заменяется на пару (b, r), где r — остаток от деления a на b, при этом наибольший общий делитель у обеих пар одинаковый. Начальные значения переменных a и b равны m и n соответственно. Алгоритм заканчивается, когда b становится равным нулю, при этом a будет содержать наибольший общий делитель.

Идея расширенного алгоритма Евклида заключается в том, что на любом шаге алгоритма хранятся коэффициенты, выражающие текущие числа a и b через исходные числа m и n. При замене пары (a, b) на пару (b, r) эти коэффициенты перевычисляются.

Итак, в алгоритме участвуют переменные a, b, u_1, v_1, u_2, v_2, для которых выполняется следующий инвариант цикла:

$$\begin{aligned} I(a, b, u_1, v_1, u_2, v_2): \quad & \text{НОД}(a, b) = \text{НОД}(m, n) \\ & a = u_1 m + v_1 n \\ & b = u_2 m + v_2 n \end{aligned}$$

Начальные значения этих переменных обеспечивают выполнение инварианта:

$$\begin{aligned} a &= m, \quad b = n, \\ u_1 &= 1, \quad v_1 = 0, \\ u_2 &= 0, \quad v_2 = 1. \end{aligned}$$

Условием завершения цикла, как и в обычном алгоритме Евклида, является равенство нулю переменной b:

$$Q(a, b, u_1, v_1, u_2, v_2): \quad b = 0.$$

Осталось написать тело цикла, сохраняющее инвариант и уменьшающее абсолютную величину переменной b. Это нетрудно сделать, исходя из инварианта цикла. В обычном алгоритме Евклида пара (a, b) заменяется на (b, r), где r — остаток от деления a на b.

$$a = qb + r, \quad |r| < |b|.$$

Здесь q равняется целой части частного от деления a на b. Заметим, что в программировании, в отличие от школьной математики, операция взятия целой части перестановочна с операцией изменения знака:

$$\textit{целая часть}(-x) = -\textit{целая часть}(x).$$

Например, $\textit{целая часть}(-3.7) = -3$. Это позволяет работать с отрицательными числами так же, как и с положительными, т.е. вообще не следить за знаком! Отметим также, что в большинстве языков программирования считается, что результат любой операции с целыми числами является целым числом, например, $8/3 = 2$.

Переменная q вычисляется как целая часть частного от деления a на b:

$$q = \textit{целая часть}(a/b).$$

Выразим остаток r в виде линейной комбинации a и b:

$$r = a - qb.$$

Используя инвариант цикла, можно выразить r через исходные числа m и n:

$$\begin{aligned} r &= a - qb = (u_1 m + v_1 n) - q(u_2 m + v_2 n) = \\ &= (u_1 - q u_2)m + (v_1 - q v_2)n. \end{aligned}$$

Через u_1', v_1', u_2', v_2' обозначаются новые значения переменных u_1, v_1, u_2, v_2. При замене $(a, b) \to (b, r)$ они вычисляются следующим образом:

$$\begin{aligned} u_1' &= u_2, & v_1' &= v_2, \\ u_2' &= u_1 - qu_2, & v_2' &= v_1 - qv_2 \end{aligned}$$

По завершению цикла ответ будет находиться в переменных a (НОД исходных чисел m и n), u_1, v_1 (коэффициенты выражения НОД через m и n).

Выпишем алгоритм:

```
алгоритм Расширенный алгоритм Евклида(
    вх: цел m, цел n,
    вых: цел d, цел u, цел v
)
| дано: целые числа m, n, хотя бы одно отлично от нуля;
| надо: вычислить d = НОД(m, n) и найти u, v такие, что
|                  d = u * m + v * n;
начало алгоритма
| цел a, b, q, r, u1, v1, u2, v2;
| цел t;        // вспомогательная переменная
| // инициализация
| a := m; b := n;
| u1 := 1; v1 := 0;
| u2 := 0; v2 := 1;
| утверждение: НОД(a, b) == НОД(m, n)  и
|               a == u1 * m + v1 * n    и
|               b == u2 * m + v2 * n;
|
| цикл пока b != 0
| | инвариант: НОД(a, b) == НОД(m, n)  и
| |            a == u1 * m + v1 * n    и
| |            b == u2 * m + v2 * n;
| | q := a / b; // целая часть частного от деления a на b
| | r := a % b; // остаток от деления a на b
| | a := b; b := r;    // заменяем пару (a, b) на (b, r)
| |
| | // Вычисляем новые значения переменных u1, u2
```

```
| | t := u2;           // запоминаем старое значение u2
| | u2 := u1 - q * u2; // вычисляем новое значение u2
| | u1 := t;           // новое значение u1 := старое
| |                    //                  значение u2
| | // Аналогично находим новые значения переменных v1, v2
| | t := v2;
| | v2 := v1 - q * v2;
| | v1 := t;
| конец цикла
|
| утверждение: b == 0                  и
|              НОД(a, b) == НОД(m, n) и
|              a == u1 * m + v1 * n;
| // Выдаем ответ
| d := a;
| u := u1; v := v1;
конец алгоритма
```

Нахождение корня функции методом деления отрезка пополам

Рассмотрим еще один пример использования схемы построения цикла с помощью инварианта, часто встречающийся в реальных программах. Пусть $y = f(x)$ — *непрерывная* функция от вещественного аргумента, принимающая вещественные значения. Пусть известно, что на заданном отрезке $[a, b]$ она принимает значения разных знаков. Из непрерывности функции f следует, что она имеет по крайней мере один корень на этом отрезке. Требуется вычислить корень функции f с заданной точностью ε.

Идея алгоритма состоит в том, чтобы поделить отрезок пополам и выбрать ту половину отрезка, на которой функция принимает значения разных знаков. Эта операция повторяется до тех пор, пока длина отрезка не станет меньше, чем ε.

Пусть концы текущего отрезка хранятся в переменных x_0, x_1. Инвариантом цикла является утверждение о том, что функция принимает значения разных знаков в точках x_0, x_1:

$$I(x_0, x_1): \quad f(x_0) \cdot f(x_1) \leq 0.$$

Начальные значения:

$$x_0 = a, \quad x_1 = b.$$

Условием завершения цикла является утверждение о том, что длина отрезка меньше ε:

$$Q(x_0, x_1): \quad |x_1 - x_0| < \varepsilon$$

(знак модуля используется потому, что в условии задачи не требуется выполнения неравенства $a < b$).

Выпишем алгоритм вычисления корня функции с заданной точностью:

```
вещ корень функции на отрезке(вх: вещ a, вещ b, вещ eps)
| дано: f(a) * f(b) <= 0,
|       eps > 0 -- очень маленькое число;
| надо: вычислить корень функции f на отрезке [a, b] с
|       точностью eps;
начало алгоритма
| вещ x0, x1, c;
|
| // инициализация
| x0 := a; x1 := b;
| утверждение: f(x0) * f (x1) <= 0;
|
| цикл пока |x1 - x0| >= eps
| | инвариант: f(x0) * f (x1) < 0;
| | c := (x0 + x1) / 2; // Середина отрезка [x0, x1]
| | если f(x0) * f(c) <= 0
| | | то
| | |   x1 := c
| | | иначе
| | |   утверждение: f(c) * f(x1) <= 0
| | |   x0 := c
| | конец если
| конец цикла
|
| утверждение: |x1 - x0| < eps  и
|              f(x0) * f (x1) <= 0;
```

```
| ответ := (x0 + x1) / 2;
конец алгоритма
```

Задачи по теме «Построение цикла с помощью инварианта»

1. По аналогии с алгоритмом быстрого возведения в степень, составить алгоритм быстрого вычисления произведения двух целых чисел, использующий лишь операции сложения, вычитания, удвоения и деления пополам. Пусть надо вычислить произведение чисел a и n. В качестве инварианта цикла использовать условие

$$b \cdot k + p = a \cdot n = \text{const},$$

где b, k, p — целочисленные переменные, которые меняются в теле цикла.

2. Определить, что вычисляется в результате данного фрагмента программы:

```
дано: целые числа a >= 0, b > 0

цел q, r, e, m;
q := 0; r := a; e := 1; m := b
утверждение: a - q*b == r  и
             m == e*b
цикл пока r >= b
| инвариант: a - q*b == r  и
|            m == e*b
| если 2*m <= r
| | то e := e*2; m := m*2;
| иначе если m > r
| | то e := e/2; m := m/2;
| иначе
| | утверждение: m <= r  и  r < 2*m
| | q := q + e; r := r - m;
| конец если
конец цикла
```

```
ответ := q;
```

Глава 2

Устройство компьютера

Компьютер — это универсальный исполнитель, который умеет управлять другими исполнителями и обладает собственной внутренней памятью. Запись алгоритма для компьютера называется программой. Все современные компьютеры построены по так называемой фон-Неймановской архитектуре: программа хранится в памяти компьютера, так же как и данные.

Компьютер построен из следующих составных частей:

процессор — это основа любого компьютера, его мозг. Процессор производит все вычисления и отдает команды всем остальным компонентам компьютера;

оперативная память также является обязательной составной частью любого компьютера. Оперативная память (RAM — Random Access Memory) хранит как программу, так и данные (т.е. значения переменных). Часть памяти может быть защищена от записи и хранится в специальной микросхеме (ПЗУ — постоянное запоминающее устройство или ROM – Read Only Memory). Обычно в ПЗУ лежит программа первоначальной загрузки и базовая система ввода-вывода (BIOS);

шина — это канал передачи команд и данных между всеми составными частями компьютера. В компьютере могут быть одна или несколько шин. Все устройства подключаются к шине *параллельно*, т.е. порядок подключения не важен, а количество

проводов не зависит от количества подключенных устройств. Порядок передачи команд и данных определяется *протоколом* работы шины, т.е. четко описанным набором соглашений, принятым, как правило, в виде международного стандарта. Каждое устройство подключается к шине с помощью *контроллера*, который осуществляет перевод с языка сигналов, передаваемых по шине, на язык команд конкретного устройства;

внешние устройства подключаются к шине компьютера. Наиболее распространенные внешние устройства — это жесткий диск, клавиатура, монитор, сетевая карта, модем и т.п. Ни одно из них не является обязательным, как показывает пример копьютера, управляющего автомобильным двигателем со впрыском топлива. Но какие-то внешние устройства всегда присутствуют, поскольку через них осуществляется связь компьютера с внешним миром.

Рассмотрим каждую из составляющих частей компьютера более подробно.

2.1. Оперативная память

Элементарной единицей памяти всех современных компьютеров является *байт*, состоящий из восьми двоичных разрядов. Каждый байт имеет свой адрес. В наиболее распространенной 32-разрядной архитектуре адреса байтов изменяются от 0 до $2^{32} - 1$ с шагом 1. Память, с логической точки зрения, можно рассматривать как массив байтов: можно прочесть или записать байт с заданным адресом. Содержимое байта трактуется либо как неотрицательное целое число в диапазоне от 0 до 255, либо как число со знаком в диапазоне от -128 до 127. (На самом деле байт — это элемент кольца вычетов по модулю 256, см. раздел 1.4.1.)

Однако, физически при работе с памятью по шине передаются не отдельные байты, а *машинные слова*. В 32-разрядной архитектуре машинное слово — это четыре подряд идущих байта, при этом адрес младшего байта кратен четырем. (В 64-разрядной архитектуре машинное слово состоит из восьми байтов.) Машинное слово — это наиболее естественный элемент данных для процессора. Машинное

слово содержит целое число, которое можно рассматривать либо как беззнаковое в диапазоне от 0 до $2^{32}-1$, либо как знаковое в диапазоне от -2^{31} до $2^{31}-1$. Адрес памяти также представляет собой машинное слово.

Принято нумеровать биты внутри машинного слова (как и внутри байта) справа налево, начиная с нуля и кончая 31. Младший бит имеет нулевой номер, старший, или знаковый, бит — номер 31 (см. раздел1.4.1). Младшие биты числа находятся в младших битах машинного слова.

Существуют два способа нумеровать байты внутри машинного слова. В соответствии с этим все процессоры разделяются на два типа:

Big Endian — байты внутри машинного слова нумеруются слева направо. Таковы процессоры Motorola, Power PC. Байты в архитектуре Big Endian удобно представлять записанными *слева направо*. При этом *старшие* биты целого числа располагаются в байте с *младшим* адресом.

Little Endian — байты внутри машинного слова нумеруются справа налево. Таковы процессоры Intel 80x86, Alpha, VAX и др. Байты в архитектуре Little Endian следует представлять записанными *справа налево*. При этом *старшие* биты целого числа располагаются в байте со *старшим* адресом.

Архитектура Big Endian была популярна в середине XX века. К концу 70-х годов программисты осознали, что Little Endian-архитектура гораздо удобнее. Например, один из аргументов в пользу Little Endian заключается в том, что целое число, занимающее машинное слово с адресом n, и байт с тем же адресом содержат одно и то же значение (конечно, если оно не превышает 255). В случае Big Endian это не так: например, если целое число с адресом n содержит число 17, то байт с адресом n содержит 0; или если целое число содержит отрицательное значение -77, то байт с адресом n содержит отрицательное значение -1. При небрежном программировании это порождает массу ошибок. Поэтому большинство современных процессоров построены по архитектуре Little Endian.

Тем не менее многие компьютерные протоколы ориентируются на Big Endian, поскольку они были приняты достаточно давно. Напри-

мер, все протоколы сети Internet передают данные в формате Big Endian, т.к. они были разработаны в 70-х годах XX века. На машинах с архитектурой Little Endian приходится переставлять байты внутри слова перед отправкой IP-пакета в сеть или при получении IP-пакета из сети.

2.2. Процессор

Процессор является основой любого компьютера. Это большая микросхема, содержащая внутри себя сотни тысяч или даже миллионы элементов. Современные процессоры чрезвычайно сложны и могут содержать несколько уровней построения и описания. Так, можно различать внешние команды процессора в том виде, в котором они используются в программах и записываются в оперативной памяти, и внутренний микрокод, применяемый для реализации внешних команд. Процессор может содержать внутри себя устройства, предназначенные для ускорения работы, — конвейер команд, устройство опережающей выборки из памяти, кеш-память и т.п.

Рассмотрим лишь самые общие принципы построения и работы процессора, которые одинаковы как для примитивных, так и для самых современных процессоров.

Любой процессор имеет устройство, выполняющее команды, и собственную внутреннюю память, реализованную внутри микросхемы процессора. Она называется *регистрами процессора*. Имеется 3 типа регистров:

общие регистры хранят целые числа или адреса. Размер общего регистра совпадает с размером машинного слова и в 32-разрядной архитектуре равен четырем байтам. Число общих регистров и их назначение зависит от конкретного процессора. В большинстве Ассемблеров к ним можно обращаться по именам R0, R1, R2, ... Среди общих регистров имеются регистры специального назначения: указатель стека SP (Stack Pointer), счетчик команд PC (Program Counter) и др.;

регистр флагов содержит биты, которые устанавливаются в единицу или в ноль в зависимости от результата выполнения последней команды. Так, бит Z устанавливается в единицу, если

результат равен нулю (Zero), бит N — если результат отрицательный (Negative), бит V — если произошло переполнение (oVerflow), бит C – если произошел перенос единицы из старшего или младшего разряда (Carry), например, при сложении двух целых чисел или при сдвиге.

Значения битов в регистре флагов используются в командах условных переходов;

плавающие регистры содержат вещественные числа. В простых процессорах аппаратная поддержка арифметики вещественных чисел может отсутствовать. В этом случае плавающих регистров нет, а операции с вещественными числами реализуются программным путем.

Команды, или инструкции, процессора состоят из кода операции и операндов. Команда может вообще не иметь операндов или иметь один, два, три операнда. Команды с числом операндов большим трех встречаются лишь в процессорах специального назначения (служащих, например, для обработки сигналов) и в обычных архитектурах не используются. Чаще всего применяются двухадресные и трехадресные архитектуры: к двухадресным относятся, к примеру, все процессоры серии Intel 80x86, к трехадресным — серии Motorola 68000. В двухадресной архитектуре команда сложения выглядит следующим образом:

```
add X, Y
```

что означает

```
X := X + Y,
```

т.е. один из аргументов команды является одновременно и ее результатом. Этот аргумент называется *получателем* (*destination*). Аргумент, который не меняется в результате выполнения команды, называется *источником* (*source*). Среди программистов нет единого мнения о том, в каком порядке записывать аргументы при использовании Ассемблера, т.е. в символической записи машинных команд. Например, в Ассемблере "masm" фирмы IBM для процессоров Intel 80x86 получатель всегда записывается первым, а источник вторым. Ассемблер "masm" используется в операционных системах MS DOS

и Windows. В Ассемблере “as”, который входит в состав компилятора “gcc” и используется в системах типа Unix (Linux и т.п.), получатель всегда является последним аргументом. Та же команда сложения записывается в “as” как

```
add Y, X
```

что означает сложить Y и X и результат записать в X.

В трехадресной архитектуре команда сложения имеет 3 операнда:

```
add X, Y, Z
```

Получателем в трехадресной архитектуре обычно является третий аргумент, т.е. в данном случае сумма X+Y записывается в Z.

Операндами команды могут быть регистры или элементы памяти. В действительности, конечно, процессор всегда сначала копирует слово из памяти в регистр, который может быть либо явно указан в команде, либо использоваться неявно. Операция всегда выполняется с содержимым регистров. После этого результат может быть записан в память либо оставлен в регистре. Например, при выполнении команды увеличения целого числа на единицу

```
inc X
```

в случае, когда операнд X является словом оперативной памяти, содержимое слова X сначала неявно копируется во внутренний регистр процессора, затем выполняется его увеличение на единицу, и после этого увеличенное значение записывается обратно в память.

Имеется несколько способов задания операнда, находящегося в оперативной памяти, они называются *режимами адресации*. Это

абсолютная адресация — когда в команде указывается константа, равная адресу аргумента;

косвенная адресация — когда в команде указывается регистр, содержащий адрес аргумента;

относительная адресация — адрес аргумента равен сумме содержимого регистра и константы, задающей смещение;

индексная адресация с масштабированием — адрес аргумента равен сумме содержимого *базового* регистра, константы, задающей смещение, а также содержимого *индексного* регистра, умноженного на *масштабирующий множитель*. Масштабирующий множитель может принимать значения 1, 2, 4, 8. Этот режим удобен для обращения к элементу массива.

Бывают и другие, более изощренные, режимы адресации, когда, например, адрес аргумента содержится в слове, адрес которого содержится в регистре (так называемая двойная косвенность).

2.2.1. CISC и RISC-процессоры

Существует два подхода к конструированию процессоров. Первый состоит в том, чтобы придумать как можно больше разных команд и предусмотреть как можно больше разных режимов адресации. Процессоры такого типа называются CISC-процессорами, от слов Complex Instruction Set Computers. Это, в частности, Intel 80x86 и Motorola 68000. Противоположный подход состоит в том, чтобы реализовать лишь минимальное множество команд и режимов адресации, процессоры такого типа называются RISC-процессорами, от слов Reduced Instruction Set Computers. Примеры RISC-процессоров: DEC Alpha, Power PC, Intel Itanium.

Казалось бы, CISC-процессоры должны иметь преимущество перед RISC-процессорами, но на самом деле все обстоит строго наоборот. Дело в том, что простота набора команд процессора облегчает его конструирование, в результате чего удается достичь следующих целей:

1) все команды выполняются исключительно быстро, причем за одинаковое время, т.е. за фиксированное число тактов работы процессора;

2) значительно поднимается тактовая частота процессора;

3) намного увеличивается количество регистров процессора и объем кеш-памяти;

4) удается добиться ортогональности режимов адресации, набора команд и набора регистров. Это означает, что нет каких-либо

выделенных регистров или режимов адресации: в любых (или почти любых) командах можно использовать произвольные регистры и режимы адресации независимо друг от друга. Следует отметить, что к памяти могут обращаться лишь команды загрузки слова из памяти в регистр и записи из регистра в память, а все арифметические команды работают только с регистрами;

5) простота команд позволяет эффективно организовать их выполнение в конвейере (pipeline), что значительно ускоряет работу программы.

Пункты 3 и 4 по достоинству оценят те, кому пришлось программировать на Ассемблере Intel 80x86, имеющем ряд ограничений на использование регистров и режимы адресации, к тому же и регистров в нем очень мало.

RISC-архитектуры обладают неоспоримыми преимуществами по сравнению с CISC-архитектурами — быстродействием, низкой стоимостью, удобством программирования и т.д. — и практически не имеют недостатков. Существование CISC-процессоров в большинстве случаев объясняется лишь традицией и требованием совместимости со старым программным обеспечением. Впрочем, существует и третий вариант — процессоры, которые по сути являются RISC-процессорами, но эмулируют внешнюю систему команд устаревших процессоров, например, современные процессоры Intel Pentium.

2.2.2. Алгоритм работы компьютера

Среди всех регистров процессора в любой архитектуре всегда имеется два выделенных регистра: это регистр PC, что означает Program Counter, по-русски его называют *счетчиком команд*, и регистр SP — Stack Pointer, т.е. *указатель стека*. Иногда регистр PC обозначают как IP, что означает Instruction Pointer, указатель инструкции. (Команды процессора часто называют инструкциями.)

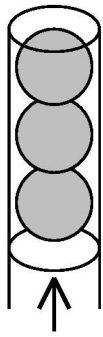
В фон-Неймановской архитектуре, по которой построены все современные компьютеры, программа, состоящая из машинных команд, содержится в оперативной памяти. Регистр PC всегда содержит адрес команды, которая будет выполняться на следующем шаге. Алгоритм работы процессора выглядит следующим образом:

```
цикл до бесконечности выполнять
| прочесть команду с адресом PC из оперативной памяти;
| увеличить содержимое PC на длину прочитанной команды;
| выполнить прочитанную команду;
конец цикла
```

В простейшем случае, когда выполняется линейный участок программы, команды выбираются из памяти и выполняются последовательно, а содержимое регистра PC монотонно возрастает. Выполнение команды, однако, может приводить к изменению регистра PC. Таким образом организуются безусловные и условные переходы в программе, нарушающие последовательный порядок выполнения команд. С помощью команд условных и безусловных переходов реализуются конструкции ветвления и цикла. Команда перехода представляет собой либо прибавление константы к содержимому PC (константа может быть положительной или отрицательной), либо загрузку в PC адреса элемента памяти со всеми возможными режимами адресации. Первый способ используется для реализации переходов внутри подпрограммы (внутри функции в терминах языка Си), второй — для перехода к подпрограмме. Впрочем, гораздо чаще в последнем случае используется команда *call* вызова подпрограммы, которая дополнительно запоминает точку возврата в регистре или в аппаратном стеке.

2.3. Аппаратный стек

Стек — это запоминающее устройство, из которого элементы извлекаются в порядке, обратном их помещению в стек. Стек можно представить как стопку листов бумаги, на каждом из которых записан один из сохраняемых элементов. На *вершине* стека находится последний запомненный элемент.



Стек изображается также в виде расположенной вертикально трубки с открытым верхним концом и подпружиненным дном, в которую можно помещать элементы через ее верхний конец. Команда добавления элемента в стек по-английски называется *push* — протолкнуть. Действительно, последний элемент как бы проталкивает предыдущие элементы вниз на одну позицию. Команда извлечения из стека называется *pop* — выбросить, выстрелить.

Аппаратный стек реализуется на базе оперативной памяти. Элементы стека расположены в оперативной памяти, каждый из них занимает одно слово. Регистр SP в любой момент времени хранит адрес элемента в вершине стека. Стек растет в сторону уменьшения адресов: элемент, расположенный непосредственно под вершиной стека, имеет адрес SP + 4 (при условии, что размер слова равен четырем байтам), следующий SP + 8 и т.д.

Оперативная память

адрес	содержимое	
0		
4		
8		
...	...	
SP	элементы	← вершина стека
SP + 4	стека	
SP + 8		
...	...	
$2^{32} - 4$		

Поскольку регистр SP содержит адрес машинного слова, его значение всегда кратно четырем. При помещении элемента x в стек значение SP сначала уменьшается на 4, затем x записывается в слово оперативной памяти с адресом SP. При извлечении элемента из стека сначала слово с адресом SP копируется в выходную переменную x, затем значение SP, т.е. адрес вершины стека, увеличивается на 4. Обычно команда добавления в стек обозначается словом *push*,

команда извлечения из стека — словом *pop*:

$$\begin{aligned} \text{push } x \quad &\sim \quad \text{SP} := \text{SP} - 4; \\ &\qquad m[\text{SP}] := x; \\ \text{pop } x \quad &\sim \quad x := m[\text{SP}]; \\ &\qquad \text{SP} := \text{SP} + 4; \end{aligned}$$

Здесь через $m[\text{SP}]$ обозначается содержимое слова памяти с адресом SP (m — сокращение от memory).

2.3.1. Команды вызова подпрограммы call и возврата return

Одно из главных назначений аппаратного стека — поддержка вызовов подпрограмм. При вызове подпрограммы надо сохранить адрес возврата, чтобы подпрограмма могла по окончанию своей работы вернуть управление вызвавшей ее программе. В старых архитектурах, в которых аппаратный стек отсутствовал (например, в компьютерах IBM 360/370), точки возврата сохранялись в фиксированных ячейках памяти для каждой подпрограммы. Это делало невозможной *рекурсию*, т.е. повторный вызов той же подпрограммы непосредственно из ее текста или через цепочку промежуточных вызовов, поскольку при повторном вызове старое содержимое ячейки, хранившей адрес возврата, терялось.

Во всех современных архитектурах точка возврата сохраняется в аппаратном стеке, что делает возможным рекурсию, а также параллельное выполнение нескольких легковесных процессов (нитей). Для вызова подпрограммы f служит команда call, которая осуществляет переход к подпрограмме f (т.е. присваивает регистру PC адрес f) и одновременно помещает старое содержимое регистра PC в стек:

$$\begin{aligned} \text{call } f \quad &\sim \quad \text{push PC}; \\ &\qquad \text{PC} := f; \end{aligned}$$

В момент выполнения любой команды регистр PC содержит адрес следующей команды, т.е. фактически адрес возврата из подпрограммы f. Таким образом, команда call сохраняет в стеке точку возврата и осуществляет переход к подпрограмме f.

Для возврата из подпрограммы используется команда return. Она извлекает из стека адрес возврата и помещает его в регистр PC:

return ~ pop PC;

2.3.2. Аппаратный стек и локальные переменные подпрограммы

Поскольку аппаратный стек располагается в оперативной памяти, в нем можно размещать обычные переменные программы. Размещение локальных переменных в стеке обладает рядом преимуществ по сравнению со статическим размещением переменных в фиксированных ячейках оперативной памяти. Как уже говорилось выше, это позволяет организовывать рекурсию. Кроме того, в современных архитектурах принципиальное значение имеет поддержка параллельных процессов, работающих над общими статическими переменными. Это так называемые легковесные процессы, или нити (Thread), работающие параллельно в рамках одной программы. На использовании нитей, например, основана работа всех графических приложений в системе Microsoft Windows 32: одна нить обрабатывает сообщения графической системы (нажатия на клавиатуру и кнопки мыши, перерисовка окон, выборка команд из меню и т.п.), другие нити занимаются вычислениями, сетевым обменом, анимацией и т.п.

Различные нити работают параллельно над общими статическими данными, совершая таким образом некоторую совместную работу. При этом одна и та же подпрограмма может вызываться из разных нитей. В отличие от статических переменных, которые являются общими для всех нитей, для каждой нити выделяется свой отдельный стек. При использовании нитей очень важно, чтобы локальные переменные подпрограммы располагались в стеке. Иначе было бы невозможно параллельно вызывать одну и ту же подпрограмму из разных нитей: повторный вызов подпрограммы, уже работающей в рамках другой нити, разрушил бы статический набор локальных переменных этой подпрограммы. А при использовании стека наборы локальных данных одной и той же подпрограммы, вызываемой из разных нитей, различны, поскольку они располагаются в разных стеках. Таким образом, разные нити работают с разными наборами локальных переменных, не мешая друг другу.

Рассмотрим более подробно, как размещаются локальные переменные подпрограммы в стеке, на примере языка Си. В Си подпрограммы называются функциями. Функция может иметь аргументы и локальные переменные, т.е. переменные, существующие только в процессе выполнения функции. Рассмотрим для примера функцию f, зависящую от двух входных аргументов x и y целого типа, в которой используются три локальные переменные a, b и c также целого типа. Функция возвращает целое значение.

```
int f(int x, int y) {
    int a, b, c;
    ...
}
```

Пусть в некотором месте программы вызывается функция f с аргументами $x = 222$, $y = 333$:

```
z = f(222, 333);
```

Вызывающая программа помещает фактические значения аргументов x и y функции f в стек, при этом на вершине стека лежит первый аргумент функции, под ним — второй аргумент. Вызов функции транслируется в следующие команды:

```
push   333
push   222
call   f
```

Обратите внимание, что в стек сначала помещается второй аргумент функции, затем первый, в результате на вершине стека оказывается первый аргумент. При выполнении инструкции вызова call в стек помещается также адрес возврата.

В момент начала работы функции f стек имеет следующий вид:

адрес возврата	← SP
222	
333	
...	

На вершине стека лежит адрес возврата, под ним — фактическое значение аргумента x, затем фактическое значение аргумента y.

Перед началом работы функция f должна захватить в стеке область памяти под свои локальные переменные a, b, c. В языке Си принято следующее соглашение: адрес блока локальных переменных функции в момент ее работы помещается в специальный регистр процессора, который называется FP, от англ. Frame Pointer — *указатель кадра*. (В процессоре Intel 80386 роль указателя кадра выполняет регистр EBP.) В первую очередь функция f сохраняет в стеке предыдущее значение регистра FP. Затем значение указателя стека копируется в регистр FP. После этого функция f захватывает в стеке область памяти размером в 3 машинных слова под свои локальные переменные a, b, c. Для этого функция f просто уменьшает значение регистра SP на 12 (три машинных слова равны двенадцати байтам). Таким образом, начало функции f состоит из следующих команд:

```
push  FP
FP  :=  SP
SP  :=  SP – 12
```

После захвата кадра локальных переменных стек выглядит следующим образом.

c	← SP
b	
a	
старое значение FP	← FP
адрес возврата	
$x = 222$	
$y = 333$	
...	

Аргументы и локальные переменные функции f адресуются относительно регистра FP. Так, аргумент x имеет адрес FP+8, аргумент y — адрес FP+12. Переменная a имеет адрес FP-4, переменная b — адрес FP-8, переменная c — адрес FP-12.

По окончании работы функция f сначала увеличивает указатель стека на 12, удаляя таким образом из стека свои локальные переменные a, b, c. Затем старое значение FP извлекается из стека и помещается в FP (таким образом, регистр FP восстанавливает свое значение до вызова функции f). После этого осуществляется возврат в вызывающую программу: адрес возврата снимается со стека

и управление передается по адресу возврата. Результат функции f передается через нулевой регистр.

```
R0 := результат функции
SP := SP + 12
pop  FP
return
```

Вызывающая программа удаляет из стека фактические значения аргументов x и y, помещенные в стек перед вызовом функции f.

2.4. RTL: машинно-независимый Ассемблер

Каждый процессор имеет свои специфические команды, наборы регистров и режимы адресации, поэтому программу на Ассемблере невозможно перенести с одной аппаратной платформы на другую. Для того чтобы не зависеть от конкретного процессора, часто используют язык описания команд RTL, от англ. Register Transfer Language — язык перемещения регистров. Фактически RTL представляет собой Ассемблер, не зависящий от конкретного процессора. Многие компиляторы, например, gcc, не переводят программу с языка высокого уровня сразу на язык машинных команд, а сначала транслируют ее на язык RTL. Затем на уровне RTL выполняется оптимизация кода, которая составляет 99% работы компилятора. И лишь на последнем этапе программа с языка RTL переводится на язык команд конкретного процессора. Поскольку RTL максимально приближен к Ассемблеру, трансляция из RTL в конкретный Ассемблер не представляет никакого труда.

Такой подход позволяет сделать компилятор с языка высокого уровня практически независимым от конкретной архитектуры. Зависим лишь модуль, осуществляющий перевод с RTL в Ассемблер, но его реализация требует минимальных усилий.

Мы будем использовать RTL для записи примеров несложных программ в кодах вместо какого-либо конкретного Ассемблера.

В RTL имеется неограниченное число регистров общего назначения

R0, R1, R2, R3, ...

и несколько выделенных регистров:

1) счетчик команд PC;

2) указатель стека SP;

3) регистр флагов CC0 (от слов Conditional Codes), иногда добавляют также дополнительные регистры флагов CC1, CC2, ...;

4) указатель кадра FP.

Слово памяти с адресом a обозначается в RTL через $m[a]$. Выражение a может быть константой, регистром или суммой регистра и константы, что соответствует абсолютному, косвенному и относительному режимам адресации. Примеры:

$$m[1000],\ m[\mathrm{R0}],\ m[\mathrm{FP}-4].$$

Байт с адресом a обозначается через $m_b[a]$, короткое (двухбайтовое) слово — через $m_s[a]$.

Арифметические команды, такие, как сложение или умножение, записываются в RTL в естественном виде, например, команда сложения двух регистров R0 и R1, помещающая результат в R2, записывается в RTL в виде

$$\mathrm{R2} := \mathrm{R0} + \mathrm{R1}$$

Команды перехода записываются следующим образом: фрагмент программы, на который осуществляется переход, отмечается меткой. Метка ставится между командами, т.е. строка с меткой всегда пустая (это позволяет добавлять в текст новые команды, не трогая строку с меткой). Сам переход выполняется с помощью команды goto, после которой указывается метка перехода, например,

```
L:
    . . .
    goto L;
```

Ветвление в RTL реализуется с помощью команд условного перехода, которые в зависимости от состояния различных битов или их

комбинаций в *регистре флагов* CC0 либо осуществляют переход на указанную метку, либо ничего на делают. Например, команда

```
if (eq) goto L;
```

осуществляет переход на метку L в случае, когда результат предыдущей команды равен нулю (eq — от слова equal), т.е. в регистре CC0 установлен бит z (от слова zero). Большинство арифметических команд автоматически устанавливают биты-признаки результата в регистре флагов CC0. Очень часто требуется просто сравнить два числа, никуда не записывая результат. Команда сравнения присутствуют в системе команд любого процессора, чаще всего она называется cmp (от слова compare). Логически команду сравнения следует понимать как вычитание двух чисел, при этом результат как бы помещается в регистр флагов. На самом деле, в регистре флагов от результата остаются лишь биты-признаки (равен ли он нулю, больше нуля и т.п.). В RTL команда сравнения двух регистров R0 и R1 записывается следующим образом:

```
CC0 := R0 - R1;
```

Результат как бы помещается в регистр флагов CC0.

В командах условного перехода, таких как

```
if (eq) goto L;
```

можно использовать следующие условия:

eq	результат равен нулю (equal)
ne	результат не равен нулю (not equal)
g	результат больше нуля (greater)
l	результат меньше нуля (less)
ge	результат больше или равен нулю (greater or equal)
le	результат меньше или равен нулю (less or equal)

Перечисленные сравнения используются для чисел со знаком. Для неотрицательных чисел (например, в случае сравнения адресов памяти) вместо слов “больше” и “меньше” используются слова “выше”

и “ниже” (above и below):

a	первое число выше второго (above)
b	первое число ниже второго (below)
ae	первое число выше или равно второму (above or equal)
be	первое число ниже или равно второму (below or equal)

Приведем простой пример реализации конструкции “если”:

```
если R0 == R1
| то R2 := 77;
конец если
```

На RTL этот фрагмент реализуется так:

```
    CC0 := R0 - R1;
    if (ne) goto L;
    R2 := 77;
L:
```

Отметим, что в команде условного перехода используется отрицание условия после слова “если”, т.е. фактически условие *обхода* фрагмента кода после слова “то”.

2.4.1. Примеры программ на RTL и Ассемблере Intel 80x86

Рассмотрим несколько простых примеров программ на “виртуальном Ассемблере” RTL и на конкретном Ассемблере для процессора Intel 80x86.

Вычисление наибольшего общего делителя

Реализуем функцию, вычисляющую наибольший общий делитель двух целых чисел. Мы уже записывали алгоритм вычисления НОД на псевдокоде. На языке Си эта функция выглядит следующим образом:

```
int gcd(int x, int y) { // цел алг. gcd(цел x, цел y)
    int a, b, r;        // | цел a, b, r;
    a = x; b = y;       // | a := x; b := y;
    while (b != 0) {    // | цикл пока b != 0
        r = a % b;      // | | r := a % b;
        a = b;          // | | a := b;
        b = r;          // | | b := r;
    }                   // | конец цикла
    return a;           // | ответ := a;
}                       // конец алгоритма
```

Вместо НОД мы назвали функцию "gcd" (от слов greatest common divizor), поскольку в языке Си русские буквы в именах функций и переменных использовать нельзя. Запишем эту программу на языке RTL. Переменные a, b, r мы будем хранить в регистрах R0, R1, R2.

```
                    // Вход в функцию:
    push FP;        // сохраним значение FP в стеке;
    FP := SP;       // определим новое значение FP;
    push R1;        // сохраним значения регистров R1
    push R2;        //                          и R2
                    //
    R0 := m[FP+8];  // a := x;
    R1 := m[FP+12]; // b := y;
L1:                 // метка начала цикла
    CC0 := R1 - 0;  //   сравнить b с нулем
    if (eq) goto L2; //  если результат равен нулю,
                    //       то перейти на метку L2
    R2 := R0 % R1;  //   r := a % b;
    R0 := R1;       //   a := b;
    R1 := R2;       //   b := r;
    goto L1         //   перейти на метку L1
L2:                 // метка конца цикла
                    // ответ уже содержится в R0
                    // выход из функции:
    pop R2;         // восстановим значения R2
    pop R1;         //                    и R1
    pop FP;         // восстановим значение FP
    return;         // вернемся в вызывающую программу
```

Эта программу можно переписать на конкретный Ассемблер, например, на Ассемблер "Masm" фирмы Microsoft для процессоров Intel 80x86. Первое, что надо сделать при переводе с RTL на Ассемблер — это распределить регистры, т.е. задать отображение виртуальных регистров R0, R1, ... на конкретные регистры данного процессора. У процессоров серии Intel 80x86 есть всего 8 общих регистров: это регистры

EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.

Процессор Intel сконструирован таким образом, что каждый регистр выполняет в определенных командах свою особую роль (Intel 80x86 — это CISC-процессор; в RISC-процессорах все регистры равноправны). В частности, команда деления всегда использует в качестве делимого длинное восьмибайтовое целое число, содержащееся в паре регистров (EDX, EAX), где старшие байты в регистре EDX. В результате выполнения команды деления вычисляется как частное, так и остаток от деления: частное помещается в регистр EAX, остаток — в регистр EDX.

В данной программе на языке RTL остаток от деления помещается в регистр R2. Поэтому регистр R2 удобно отобразить на регистр EDX, это позволит избежать лишних пересылок результата из одного регистра в другой. Итак, зафиксируем следующее распределение регистров:

$$R0 \rightarrow EAX$$
$$R1 \rightarrow EBX$$
$$R2 \rightarrow EDX$$
$$FP \rightarrow EBP$$
$$SP \rightarrow ESP$$

После того как распределены регистры, остается только переписать каждую строку RTL программы на конкретный Ассемблер. Для этого необходимо знать ограниченный набор команд, реализующих операции языка RTL в конкретном Ассемблере. Например, в нашем случае операция пересылки из одного регистра в другой или из памяти в регистр реализуется командой mov, операция деления реализуется командой div и т.д. Программа на языке Ассемблера Intel 80386 записывается следующим образом:

```
.386
.model flat, stdcall
.code

gcd:                        ; Вход в функцию:
    push    EBP             ; сохраним старое значение EBP
    mov     EBP, ESP        ; определим новое значение EBP
    push    EBX             ; сохраним значения EBX
    push    EDX             ;                   и EDX.
                            ;
    mov     EAX, [EBP+8]    ; EAX := x
    mov     EBX, [EBP+12]   ; EBX := y
L1:                         ; метка начала цикла
    cmp     EBX, 0          ;   сравнить EBX с нулем
    je      L2              ;   если результат равен нулю,
                            ;       то перейти на метку L2
    mov     EDX, 0          ;
    div     EBX             ;   EDX := EAX % EBX
    mov     EAX, EBX        ;   EAX := EBX
    mov     EBX, EDX        ;   EBX := EDX
    jmp     L1              ;   перейти на метку L1
L2:                         ; метка конца цикла
                            ; ответ уже содержится в EAX
                            ; выход из функции:
    pop EDX                 ; восстановим значения EDX
    pop EBX                 ;                     и EBX
    pop EBP                 ; восстановим значение EBP
    ret                     ; возврат из функции

public gcd
end
```

Суммирование массива

Рассмотрим еще один простой пример программы на Ассемблере. Требуется вычислить сумму элементов целочисленного массива заданной длины. Прототип этой функции на языке Си выглядит следующим образом:

```
int sum(int a[], int n);
```

Функции передается (как обычно, через аппаратный стек) адрес начала целочисленного массива a и его длина n. На RTL функция sum записывается следующим образом:

```
                    // Вход в функцию:
    push FP;        // сохраним старое значение FP;
    FP := SP;       // определим новое значение FP;
    push R1;        // сохраним значения регистров R1,
    push R2;        //                              R2
    push R3;        //                            и R3.
                    //
    R0 := 0;        // обнулим сумму
    R1 := m[FP+8];  // R1 := a  (адрес начала массива)
    R2 := m[FP+12]; // R2 := n  (число эл-тов массива)
L1:                 // метка начала цикла
    CC0 := R2 - 0;  //   сравнить R2 с нулем
    if (le) goto L2; //  если результат <= 0,
                    //       то перейти на метку L2
    R3 := m[R1];    //   R3 := очередной элемент массива
    R0 := R0 + R3;  //   прибавим очередной эл-т к сумме
    R1 := R1 + 4;   //   увеличим адрес очер. эл-та на 4
    R2 := R2 - 1;   //   уменьшим счетчик необр. эл-тов
    goto L1         //   перейти на метку L1
L2:                 // метка конца цикла
                    // ответ уже содержится в R0
                    // выход из функции:
    pop R3;         // восстановим значения R3,
    pop R2;         //                       R2
    pop R1;         //                     и R1
    pop FP;         // восстановим старое значение FP
    return;         // вернемся в вызывающую программу
```

В этой программе адрес очередного элемента массива содержится в регистре R1. Сумма просмотренных элементов массива накапливается в регистре R0. Регистр R2 содержит число еще не обработанных элементов массива. В начале программы в регистр R1 записывается адрес начала массива, а в R2 — число элементов массива. В теле

цикла очередной элемент массива читается из памяти и помещается в регистр R3, затем содержимое R3 прибавляется к сумме R0. После каждого выполнения тела цикла адрес очередного элемента увеличивается на 4 (т.к. целое число занимает 4 байта), а количество необработанных элементов уменьшается на единицу. Цикл продолжается, пока содержимое регистра R2 (т.е. число необработанных элементов) больше нуля.

Для переписывания программы на Ассемблер Intel 80386 зафиксируем следующее распределение виртуальных регистров:

$$R0 \rightarrow EAX$$
$$R1 \rightarrow EBX$$
$$R2 \rightarrow ECX$$
$$R3 \rightarrow EDX$$
$$FP \rightarrow EBP$$
$$SP \rightarrow ESP$$

Программа переписывается таким образом:

```
.386
.model flat, stdcall
.code

sum:                    ; Вход в функцию:
   push    EBP          ; сохраним старое значение EBP
   mov     EBP, ESP     ; определим новое значение EBP
   push    EBX          ; сохраним значения регистров EBX,
   push    ECX          ;                               ECX
   push    EDX          ;                             и EDX.
                        ;
   mov     EAX, 0       ; EAX := 0
   mov     EBX, [EBP+8] ; EAX := a
   mov     ECX, [EBP+12]; ECX := n
L1:                     ; метка начала цикла
   cmp     ECX, 0       ;  сравнить ECX с нулем
   jle     L2           ;  если результат <= 0,
                        ;      то перейти на метку L2
   mov     EDX, [EBX]   ;  EDX := очередной эл-т массива
   add     EAX, EDX     ;  EAX := EAX+EDX
```

```
   add     EBX, 4          ;  EBX := EBX+4 (адрес след. эл-та)
   dec     ECX             ;  ECX := ECX-1 (счетчик)
   jmp     L1              ;  перейти на метку L1
L2:                        ; метка конца цикла
                           ; ответ содержится в регистре EAX
                           ; выход из функции:
   pop EDX                 ; восстановим значения EDX,
   pop ECX                 ;                        ECX
   pop EBX                 ;                      и EBX.
   pop EBP                 ; восстановим значение EBP
   ret                     ; вернемся в вызывающую программу

public sum
end
```

Отметим, что мы использовали команду уменьшения значения регистра на единицу dec (от слова decrement) для реализации следующей строки RTL:

```
R2 := R2 - 1; // уменьшим счетчик необр. эл-тов
```

В Ассемблере Intel 80386 она записывается как

```
dec ECX      ; ECX := ECX-1
```

Команда увеличения регистра на единицу обычно записывается как inc (от слова increment). Эти команды, как правило, присутствуют в наборе инструкций любого процессора.

2.4.2. Задачи по теме «Программирование на Ассемблере»

Во всех задачах требуется написать на RTL или на Ассемблере Intel 80386 функцию, которая вызывается из программы на Си. Функция должна возвращать целочисленный ответ в регистре R0 (в регистре EAX в случае Ассемблера Intel 80386). Аргументы передаются функции через стек в соответствии с соглашениями языка Си.

1. Функции передается адрес массива целых чисел и его длина. Вычислить значение максимального элемента массива.

2. Вычислить целую часть квадратного корня из заданного целого неотрицательного числа методом деления отрезка пополам (см. раздел 1.5.2).

3. Вычислить целую часть логарифма по основанию 2 от заданного целого положительного числа.

4. Возвести целое число в заданную целую неотрицательную степень, используя алгоритм быстрого возведения в степень (см. раздел 1.5.2).

2.5. Внешние устройства и аппаратные прерывания

Согласно рассмотренному ранее алгоритму работы компьютера, процессор в любой момент времени выполняет команду, адрес которой находится в регистре PC. После чтения команды из оперативной памяти содержимое PC увеличивается на длину прочитанной команды. Затем команда выполняется, читается следующая команда, и все это повторяется в бесконечном цикле. Однако предусмотрена возможность нарушения этого бесконечного цикла. Она продиктована необходимостью реагировать на события, связанные с внешними устройствами компьютера.

Например, рассмотрим ситуацию, когда нажата клавиша на клавиатуре компьютера. Процессор должен временно прервать выполнение текущей задачи и отреагировать на нажатие клавиши. Затем он, возможно, возобновит выполнение прерванной задачи. Прерывание от клавиатуры относится к классу так называемых *аппаратных*, или *асинхронных*, прерываний. Бывают также и синхронные прерывания, которые происходят не в результате внешних событий, а при выполнении команд процессора. Например, синхронное прерывание происходит при попытке выполнить деление на ноль.

Инициатором аппаратного прерывания всегда является какое-либо внешнее устройство. Все внешние устройства подключены к компьютеру *параллельно* с помощью *шины*. Шина представляет собой, попросту говоря, набор проводов, соединяющих различные компоненты компьютера. Порядок обмена данными по шине определяет-

ся *протоколом* работы шины. В частности, шина содержит несколько линий, отвечающих за аппаратные прерывания. Когда внешнее устройство определяет наступление некоторого события, требующего немедленной обработки, оно выставляет на шине сигнал аппаратного прерывания. Как правило, это сигнал о наличии прерывания плюс индентификатор прерывания, т.е. целое число, которое устанавливается в виде двоичного кода на соответствующих линиях шины. Процессор по идентификатору прерывания определяет, с каким внешним устройством связано данное прерывание.

Получив по шине сигнал прерывания, процессор прерывает текущую работу и переключается на обработку прерывания. Текущее состояние процессора запоминается в аппаратном стеке. (Состояние процессора определяется набором его самых важных регистров, включающим, например, регистр PC и регистр флагов; конкретный набор запоминаемых в стеке регистров зависит от конструкции процессора.) Затем процессор переходит к выполнению специальной программы–обработчика прерывания. Эта программа определяется по номеру прерывания, выставленному внешним устройством на шине.

Во время выполнения обработчика прерывания другие прерывания запрещены. Программа обработки прерывания должна быть короткой и быстрой, чтобы по возможности не нарушить нормальную работы компьютера. Так, программа обработки прерывания от клавиатуры должна прочесть код нажатой или отпущенной клавиши, для этого процессор читает данные из портов ввода-вывода, соответствующих клавиатуре. (Чтение и запись в порты ввода-вывода также происходят через шину и определяются протоколом работы шины.) Затем команда, соответствующая нажатой или отпущенной клавише, ставится в очередь запросов к драйверу клавиатуры. На этом программа-обработчик прерывания от клавиатуры завершает свою работу. По завершению обработчика прерывания процессор восстанавливает свое состояние, которое было ранее запомнено в аппаратном стеке. После восстановления процессор продолжает работу с того места, на котором она была прервана.

Реально клавиатурная команда будет обработана, когда до нее дойдет черед, т.е. только после выполнения всех команд, поступивших от клавиатуры ранее и сохраненных в очереди клавиатурного

драйвера, а также поступивших ранее команд от других внешних устройств.

2.6. Виртуальная память и поддержка параллельных задач

Все современные операционные системы поддерживают параллельное выполнение нескольких процессов на одном компьютере. Компьютер может иметь один или несколько процессоров, однако, даже в многопроцессорных системах количество выполняемых задач обычно превышает число процессоров. Поэтому современные процессоры обязательно реализуют механизм быстрого переключения с одной задачи на другую. Среди всех этих механизмов наиболее важным является поддержка *виртуальной памяти*.

2.6.1. Страничная организация памяти

Выполняемые параллельно на одном и том же компьютере различные задачи оперируют одними и теми же адресами памяти. Адрес памяти, который указывается в конкретной команде процессора, называется *виртуальным*. Для поддержки многозадачности операционная система должна отображать одни и те же виртуальные адреса у разных задач на различные *физические* адреса памяти.

Для отображения виртуальной памяти на физическую используется механизм *страничной организации памяти*. Вся физическая память делится на страницы, обычный размер страницы — 4096 байтов (4 килобайта). Для каждой задачи выделяется специальная область памяти, в которой записаны адреса физических страниц, соответствующих виртуальным страницам. Адрес этой области содержится в специальном регистре процессора. Когда программа обращается к некоторому виртуальному адресу, то он делится на 4096, таким образом определяется номер виртуальной страницы. Затем адрес физической страницы, соответствующий данному номеру, извлекается из специальной области памяти, хранящей физические адреса виртуальных страниц. После этого к вычисленному таким образом адресу физической страницы прибавляется смещение внутри страницы, т.е. остаток от деления виртуального адреса на 4096 (на размер

страницы).

Таким образом, разные задачи работают с одними и теми же *виртуальными* адресами, но с разными *физическими* адресами, не мешая друг другу.

Отметим еще один очень важный аспект виртуальной памяти. Объем физической памяти может быть меньше, чем объем виртуальной памяти. В этом случае используется механизм *своппинга*, т.е. вытеснения страниц физической памяти на диск. Считается, что объем пространства на диске компьютера практически бесконечен. Поэтому при нехватке физической памяти страницу виртуальной памяти можно разместить на диске. При обращении к этой странице операционная система сначала загружает ее с диска в физическую память. При этом, возможно, какая-то другая страница вытесняется на диск, чтобы освободить место. Отсюда и слово своппинг (swapping), что в переводе означает "обмен". Загрузка виртуальной страницы с диска происходит в результате синхронного прерывания в связи с отсутствием физической страницы в памяти.

Своппинг может замедлить выполнение программы в миллионы раз, это крайне нежелательное явление. Следует писать программы так, чтобы им всегда хватало физической памяти.

2.6.2. Переключение между процессами и нитями

В многозадачной операционной системе процессор периодически переключается между различными задачами. В момент переключения выполнение текущей задачи приостанавливается, ее состояние сохраняется ядром операционной системы, и система переключается на выполнение другой задачи. Такие переключения происходят регулярно по прерываниям от таймера. *Таймер* — это внешнее устройство, входящее в конструкцию любого компьютера. Таймер может быть реализован внутри микросхемы процессора или отдельно от нее. Продолжительность кванта времени, в котором не происходит переключение задач, обычно измеряется сотыми или тысячными долями секунды и зависит от операционной системы.

В современных операционных системах различают понятия процесса и нити (thread). В рамках одного процесса могут существовать несколько нитей. Все нити одного процесса разделяют одно и то же виртуальное адресное пространство, соответствующее статическим и

глобальным переменным программы. Однако стек, стековая (локальная) память и наборы значений регистров у каждой нити свои.

Каждая нить выполняется по отдельности, поэтому нити иногда называют *легковесными процессами* (light-weight process). Переключение между нитями происходит аналогично переключению между процессами. Нити очень удобны в программировании, поскольку разные нити параллельно выполняют совместную работу над одними и теми же глобальными переменными, расположенными в статической памяти процесса. Это позволяет избежать сложных механизмов передачи данных между различными процессами, которые приходилось использовать в старых операционных системах до изобретения нитей. На использовании нитей основано, например, программирование графических задач в операционных системах типа Windows. Здесь одна нить отвечает за обработку действий пользователя и поддержку оконного интерфейса (нажатий на мышь, клавиатуру, перерисовку окон и т.п.). Другие нити могут выполнять вычислительную работу и сетевой обмен, поддерживать анимацию и т.п. Выполнять вычислительную работу внутри нити, отвечающей за графический интерфейс, нельзя, потому что длительные вычисления приводят к визуальному подвисанию программы и к замедленным реакциям на действия пользователя.

Для синхронизации нитей и процессов, а также исключения одновременного обращения к критическим данным операционная система предоставляет программисту специальные *объекты синхронизации* — семафоры, критические секции, события, мьютексы и др. Идея заключается в том, что для каждого критического набора данных выделяется специальный объект, который может в любой момент времени принадлежать только одной нити. Так, на железных дорогах в старой Англии, когда существовал только один путь, используемый в обоих направлениях, поезд не мог выехать на этот путь, пока машинист не получал в свои руки специальное маркерное кольцо, соответствующее данному перегону. Это исключало встречное движение и столкновение поездов.

Для защиты критических данных используется объект синхронизации типа *мьютекс*, от англ. MUTual EXclusive — взаимно исключающий. Нить перед обращением к критическим данным пытается захватить мьютекс, соответствующий этим данным. Если он уже

захвачен, то операционная система приостанавливает нить до тех пор, пока объект не будет освобожден. После этого мьютекс передается нити, нить пробуждается и выполняет необходимые действия. По завершении критических действий нить сама должна освободить мьютекс, подобно тому как машинист поезда должен сам оставить маркерное кольцо на станции после проезда участка пути, "защищенного" этим кольцом.

Глава 3

Основы языка Си

В настоящее время язык Си и объектно-ориентированные языки его группы (прежде всего C++, а также Java и C#) являются основными в практическом программировании. Достоинство языка Си — это, прежде всего, его простота и лаконичность. Язык Си легко учится. Главные понятия языка Си, такие, как статические и локальные переменные, массивы, указатели, функции и т.д., максимально приближены к архитектуре реальных компьютеров. Так, указатель — это просто адрес памяти, массив — непрерывная область памяти, локальные переменные — это переменные, расположенные в аппаратном стеке, статические — в статической памяти. Программист, пишущий на Си, всегда достаточно точно представляет себе, как созданная им программа будет работать на любой конкретной архитектуре. Другими словами, язык Си предоставляет программисту полный контроль над компьютером.

Первоначально язык Си задумывался как заменитель Ассемблера для написания операционных систем. Поскольку Си — это язык высокого уровня, не зависящий от конкретной архитектуры, текст операционной системы оказывался легко переносимым с одной платформы на другую. Первой операционной системой, написанной практически целиком на Си, была система Unix. В настоящее время почти все используемые операционные системы написаны на Си. Кроме того, средства программирования, которые операционная система предоставляет разработчикам прикладных программ (так называемый API — Application Program Interface), — это наборы системных

функций на языке Си.

Тем не менее, область применения языка Си не ограничилась разработкой операционных систем. Язык Си оказался очень удобен в программах обработки текстов и изображений, в научных и инженерных расчетах. Объектно-ориентированные языки на основе Си отлично подходят для программирования в оконных средах.

В данном разделе будут приведены лишь основные понятия языка Си (и частично C++). Это не заменяет чтения полного учебника по Си или C++, например, книг [6] и [8].

Мы будем использовать компилятор C++ вместо Си. Дело в том, что язык Си почти целиком входит в C++, т.е. нормальная программа, написанная на Си, является корректной C++ программой. Слово "нормальная" означает, что она не содержит неудачных конструкций, оставшихся от ранних версий Си и не используемых в настоящее время. Компилятор C++ предпочтительнее, чем компилятор Си, т.к. он имеет более строгий контроль ошибок. Кроме того, некоторые конструкции C++, не связанные с объектно-ориентированным программированием, очень удобны и фактически являются улучшением языка Си. Это, прежде всего, комментарии //, возможность описывать локальные переменные в любой точке программы, а не только в начале блока, и также задание констант без использования оператора #define препроцесора. Мы будем использовать эти возможности C++, оставаясь по существу в рамках языка Си.

3.1. Структура Си-программы

Любая достаточно большая программа на Си (программисты используют термин *проект*) состоит из файлов. Файлы транслируются Си-компилятором независимо друг от друга и затем объединяются программой-построителем задач, в результате чего создается файл с программой, готовой к выполнению. Файлы, содержащие тексты Си-программы, называются *исходными*.

В языке Си исходные файлы бывают двух типов:

- заголовочные, или h-файлы;
- файлы реализации, или Си-файлы.

Имена заголовочных файлов имеют расширение ".h". Имена файлов реализации имеют расширения ".c" для языка Си и ".cpp", ".cxx" или ".cc" для языка C++.

> К сожалению, в отличие от языка Си, программисты не сумели договориться о едином расширении имен для файлов, содержащих программы на C++. Мы будем использовать расширение ".h" для заголовочных файлов и расширение ".cpp" для файлов реализации.

Заголовочные файлы содержат только описания. Прежде всего, это прототипы функций. Прототип функции описывает имя функции, тип возвращаемого значения, число и типы ее аргументов. Сам текст функции в h-файле не содержится. Также в h-файлах описываются имена и типы внешних переменных, константы, новые типы, структуры и т.п. В общем, h-файлы содержат лишь *интерфейсы*, т.е. информацию, необходимую для использования программ, уже написанных другими программистами (или тем же программистом раньше). Заголовочные файлы лишь сообщают информацию о других программах. При трансляции заголовочных файлов, как правило, никакие объекты не создаются. Например, в заголовочном файле нельзя *определить* глобальную переменную. Строка описания

```
int x;
```

определяющая целочисленную переменную x, является ошибкой. Вместо этого следует использовать описание

```
extern int x;
```

означающее, что переменная x определена где-то в файле реализации (в каком — неизвестно). Слово extern (внешняя) лишь *сообщает информацию* о внешней переменной, но не определяет эту переменную.

Файлы реализации, или Си-файлы, содержат тексты функций и определения глобальных переменных. Говоря упрощенно, Си-файлы содержат сами программы, а h-файлы — лишь информацию о программах.

Представление исходных текстов в виде заголовочных файлов и файлов реализации необходимо для создания больших проектов, имеющих модульную структуру. Заголовочные файлы служат для передачи информации между модулями. Файлы реализации — это

отдельные модули, которые разрабатываются и транслируются независимо друг от друга и объединяются при создании выполняемой программы.

Файлы реализации могут подключать описания, содержащиеся в заголовочных файлах. Сами заголовочные файлы также могут использовать другие заголовочные файлы. Заголовочный файл подключается с помощью директивы препроцессора #include. Например, описания стандартых функций ввода-вывода включаются с помощью строки

```
#include <stdio.h>
```

(stdio — от слов standard input/output). Имя h-файла записывается в угловых скобках, если этот h-файл является частью стандартной Си-библиотеки и расположен в одном из системных каталогов. Имена h-файлов, созданных самим программистом в рамках разрабатываемого проекта и расположенных в текущем каталоге, указываются в двойных кавычках, например,

```
#include "abcd.h"
```

Препроцессор — это программа предварительной обработки текста непосредственно перед трансляцией. Команды препроцессора называются *директивами*. Директивы препроцессора содержат символ диез # в начале строки. Препроцессор используется в основном для подключения h-файлов. В Си также очень часто используется директива #define для задания символических имен констант. Так, строка

```
#define PI 3.14159265
```

задает символическое имя PI для константы 3.14159265. После этого имя PI можно использовать вместо числового значения. Препроцессор находит все вхождения слова PI в текст и заменяет их на константу.

Таким образом, препроцессор осуществляет подмену одного текста другим. Необходимость использования препроцессора всегда свидетельствует о недостаточной выразительности языка. Так, в любом Ассемблере средства препроцессирования используются довольно интенсивно. В Си по возможности следует избегать чрезмерного увлечения командами препроцессора — это затрудняет понимание текста программы и зачастую ведет к трудно исправляемым ошибкам. В C++ можно обойтись без использования директив #define для задания констант. Например, в C++ константу PI можно задать с помощью нормального описания

```
const double PI = 3.14159265;
```

Это является одним из аргументов в пользу применения компилятора C++ вместо Си даже при трансляции программ, не содержащих конструкции класса.

3.2. Функции

Функция является основной структурной единицей языка Си. В других языках функции иногда называют подпрограммами. Функция — это фрагмент программы, который может вызываться из других программ. Функция обычно выполняет алгоритм, который описывается и реализуется отдельно от других алгоритмов. При вызове функции передаются аргументы, которые могут быть использованы в теле функции. В результате своей работы функция возвращает значение некоторого типа. Например, функция sin, прототип которой задается строкой

```
double sin(double x);
```

имеет один аргумент x типа double (вещественное число). Результат функции также имеет тип double. При вызове фукция sin вычисляет синус числа, переданного ей в качестве фактического аргумента, и возвращает вычисленное значение в вызывающую программу.

Вызов функции происходит в результате использования ее имени в выражении. За именем функции следуют круглые скобки, внутри которых перечисляются фактические значения ее аргументов. Даже если аргументов нет, круглые скобки с пустым списком аргументов обязательно должны присутствовать!

После вызова функции значение, возвращенное в результате ее выполнения, используется в выражении (имя функции как бы заменяется возвращенным значением). Примеры:

```
x = sin(1.0);
```

Здесь в результата вызова функции sin вычисляется синус числа 1.0, затем вычисленное значение записывается в переменную x при выполнения оператора присваивания "=". Другой пример:

```
f();
```

Вызывается функция f, не имеющая параметров. Значение, возвращенное в результате выполнения функции f, не используется.

Программа на Си состоит из функций. Работа программы всегда начинается с функции с именем *main*. Рассмотрим минимальный пример Си-программы.

3.2.1. Программа "Hello, World!"

Приведенная ниже программа печатает фразу "Hello, World!" на экране терминала.

```
#include <stdio.h>

int main() {
    printf("Hello, World\n");
    return 0;
}
```

Первая строка подключает заголовочный файл с описаниями стандартных функций ввода-вывода Си-библиотеки. В частности, в этом файле описан прототип функции *printf* (печать по формату), используемой для вывода информации в стандартный поток вывода (по умолчанию он назначен на терминал). Выполнение программы начинается с функции *main*. Функция *main* возвращает по окончании работы целое число, которое трактуется операционной системой как код завершения задания. Число ноль обычно означает успешное выполнение задачи, но вообще-то программист волен по своему усмотрению определять коды завершения. Во многих книгах приводятся примеры функций *main*, которые ничего не возвращают, — строго говоря, это ошибка (на которую, к сожалению, многие компиляторы никак не реагируют).

Тело любой функции заключается в фигурные скобки. В теле функции *main* вызывается функция *printf*. В данном случае ее единственным аргументом является строка, которая выводится в стандартный поток вывода. Строковые константы в Си заключаются в двойные апострофы. Строка заканчивается символом перевода курсора в начало следующей строки \n (читается как "new line", новая строка). Желательно любую печать завершать этим символом, иначе

при следующей печати новая строка будет дописана в конец предыдущей.

Строка

```
return 0;
```

завершает выполнение функции *main* и возвращает нулевой результат ее выполнения. Операционная система трактует нулевой результат как признак успешного завершения программы.

Для выполнения данной программы надо сначала ввести ее текст в файл "hello.cpp", используя любой текстовый редактор. Затем надо скомпилировать и собрать готовую программу. Конкретные команды зависят от операционной системы и установленного Си-компилятора. В системе Unix с компилятором gcc из пакета GNU это делается с помощью команды

```
g++ hello.cpp
```

В результате создается выполняемый файл с именем "a.out". Для запуска программы следует выполнить команду

```
./a.out
```

Если необходимо, чтобы в результате компиляции и сборки создавался выполняемый файл с именем "hello", то надо выполнить следующую команду:

```
g++ -o hello hello.cpp
```

Здесь в командной строке используется ключ "-o hello" (от слова "output"), задающий имя "hello" для выходного файла. В этом случае программа запускается с помощью команды

```
./hello
```

Заметим, что в системе Unix имена выполняемых файлов обычно не имеют никакого расширения. В системах MS DOS и MS Windows выполняемые файлы имеют расширение ".exe".

3.3. Типы переменных

При рассмотрении типов переменных в Си и C++ следует различать понятия базового типа и конструкции, позволяющей строить новые типы на основе уже построенных. Базовых типов совсем немного — это целые и вещественные числа, которые могут различаться по диапазону возможных значений (или по длине в байтах) и, в случае языка C++, логический тип. К конструкциям относятся массив, указатель и структура, а также класс в C++.

3.3.1. Базовые типы

В языке Си используются всего два базовых типа: целые и вещественные числа. Кроме того, имеется фиктивный тип void ("пустота"), который применяется либо для функции, не возвращающей никакого значения, либо для описания указателя общего типа (когда неизвестна информации о типе объекта, на который ссылается указатель).

В C++ добавлен логический тип.

Целочисленные типы

Целочисленные типы различаются по длине в байтах и по наличию знака. Их четыре — char, short, int и long. Кроме того, к описанию можно добавлять модификаторы unsigned или signed для беззнаковых (неотрицательных) или знаковых целых чисел.

Тип int

Самый естественный целочисленный тип — это тип int, от слова integer — целое число. Тип int всегда соответствует размеру машинного слова или адреса. Все действия с элементами типа int производятся максимально быстро. Всегда следует выбирать именно тип int, если использование других целочисленных типов не диктуется явно спецификой решаемой задачи. Параметры большинства стандартных функций, работающих с целыми числами или символами, имеют тип int. Целочисленные типы были подробно рассмотрены в разделе 1.4.1. Подчеркнем еще раз, что целочисленные переменные

хранят на самом деле не целые числа, а элементы кольца вычетов по модулю m, где m — степень двойки.

В современных архитектурах элемент типа int занимает 4 байта, т.е. $m = 2^{32}$. Элементы типа int трактуются в Си как числа со знаком. Минимальное отрицательное число равно $-2^{31} = -2147483648$, максимальное положительное равно $2^{31} - 1 = 2147483647$.

При описании переменной сначала указывается базовый тип, затем — имя переменной или список имен, разделенных запятыми, например,

```
int x;
int y, z, t;
```

При описании переменных можно присваивать им начальные значения:

```
int maxind = 1000;
int a = 5, b = 7;
```

Кроме типа int, существуют еще три целочисленных типа: char, short и long.

Тип char

Тип char представляет целые числа в диапазоне от -128 до 127. Элементы типа char занимают один байт памяти. Слово "char" является сокращением от character, что в переводе означает "символ". Действительно, традиционно символы представляются их целочисленными кодами, а код символа занимает один байт (см. раздел 1.4.3). Тем не менее, подчеркнем, что элементы типа char — это именно целые числа, с ними можно выполнять все арифметические операции. С математической точки зрения, элементы типа char — это элементы кольца вычетов $\mathbf{Z}_{256}$. Стандарт Си не устанавливает, трактуются ли элементы типа char как знаковые или беззнаковые числа, но большинство Си-компиляторов считают char знаковым типом. Примеры описаний переменных типа char:

```
char c;
char eof = (-1);
char letterA = 'A';
```

В последнем случае значение переменной “letterA” инициализируется кодом латинской буквы 'A', т.е. целым числом 65. В Си символьные константы записываются в одинарных апострофах и означают коды соответствующих символов в кодировке ASCII. Рассмотрим следующий пример:

```
char c = 0;
char d = '0';
```

Здесь переменная c инициализируется нулевым значением, а переменная d — значением 48, поскольку символ '0' имеет код 48.

Типы short и long

Слова short и long означают в Си короткое и длинное целое число со знаком. Стандарт Си не устанавливает конкретных размеров для типов short и long. В самой распространенной в настоящее время 32-разрядной архитектуре переменная типа short занимает 2 байта (диапазон значений — от -32768 до 32767), а тип long совпадает с типом int, размер его равен четырем байтам. Примеры описаний:

```
short s = 30000;
long x = 100000;
int y = 100000;
```

В 32-разрядной архитектуре переменные x и y имеют один и тот же тип.

Модификатор unsigned

Типы int, short и long представляют целые числа со знаком. Для типа char стандарт Си не устанавливает явно наличие знака, однако большинство компиляторов трактуют элементы типа char как целые числа со знаком в диапазоне от -128 до 127. Если необходимо трактовать целые числа как неотрицательные, или беззнаковые, следует добавить модификатор unsigned при описании переменных. Примеры:

```
unsigned char c = 255;
unsigned short s = 65535;
```

```
unsigned int i = 1000000000;
unsigned j = 1;
```

При описании типа "unsigned int" слово "int" можно опускать, что и сделано в последнем примере.

Следует по возможности избегать беззнаковых типов, поскольку арифметика беззнаковых чисел не на всех компьютерах реализована одинаково и из-за этого при переносе программы с одной платформы на другую могут возникнуть проблемы. По этой причине в языке Java беззнаковые числа запрещены.

Имеется также модификатор signed (знаковый). Его имеет смысл использовать на тех платформах, в которых тип char является беззнаковым. Пример описания:

```
signed char d = (-1);
```

Вещественные типы

Вещественных типов два: длинное вещественное число double (переводится как "двойная точность") и короткое вещественное число float (переводится как "плавающее"). Вещественные типы были подробно рассмотрены в разделе 1.4.2. Вещественное число типа double занимает 8 байтов, типа float — 4 байта.

Тип double является основным для компьютера. Тип float — это, скорее, атавизм, оставшийся от ранних версий языка Си. Компьютер умеет производить арифметические действия только с элементами типа double, элементы типа float приходится сначала преобразовывать к double. Точность, которую обеспечивает тип float, низка и не достаточна для большинства практических задач. Все стандартные функции математической библиотеки работают только с типом double. Рекомендуем вам никогда не использовать тип float!

Примеры описаний вещественных переменных:

```
double x, y, z;
double a = 1.5, b = 1e+6, c = 1.5e-3;
```

В последних двух случаях использовалось задание вещественных констант в экспоненциальной форме (см. раздел 1.4.2).

Логический тип

В языке Си специального логического типа нет, вместо него используются переменные целого типа. Значению "истина" соответствует любое ненулевое целое число, значению "ложь" — ноль. Например, в Си допустим такой фрагмент программы:

```
int b;
double s;
. . .
if (b) {
    s = 1.0;
}
```

Здесь целочисленная переменная b используется в качестве условного выражения в операторе if ("если"). Если значение b отлично от нуля, то выполняется тело оператора if, т.е. переменной s присваивается значение 1.0; если значение b равно нулю, то тело оператора if не выполняется.

На самом деле, приведенный пример представляет собой дурной стиль программирования. Гораздо яснее выглядит следующий фрагмент, эквивалентный приведенному выше:

```
if (b != 0) {
    s = 1.0;
}
```

В более строгом языке Java второй фрагмент корректен, а первый нет.

Язык C++ вводит логический тип bool в явном виде (отметим, что этот тип появился в C++ далеко не сразу!). Переменные типа bool принимают два значения: false и true (истина и ложь). Слова false и true являются ключевыми словами языка C++.

Примеры описания логических переменных в C++:

```
bool a, b;
bool c = false, d = true;
```

Оператор sizeof

Переменная одного и того же типа на разных платформах может занимать различное число байтов памяти. Язык Си предоставляет программисту возможность получить размер элемента данного типа или размер переменной в байтах, для этого служит оператор sizeof. Аргумент sizeof указывается в круглых скобках, он может быть типом или переменной. Рассмотрим несколько примеров. Пусть определены следующие переменные:

```
int i; char c; short s; long l;
double d; float f; bool b;
```

Тогда приведенные ниже выражения в 32-разрядной архитектуре имеют следующие значения:

размер переменной	размер типа	значение
sizeof(i)	sizeof(int)	4
sizeof(c)	sizeof(char)	1
sizeof(s)	sizeof(short)	2
sizeof(l)	sizeof(long)	4
sizeof(d)	sizeof(double)	8
sizeof(f)	sizeof(float)	4
sizeof(b)	sizeof(bool)	1

Тип void

Слово void означает “пустота”. Тип void в Си обозначает отсутствие чего-либо там, где обычно предполагается описание типа. Например, функция, не возвращающая никакого значения, в Си описывается как возвращающая значение типа void:

```
void f(int x);
```

Другое применение ключевого слова void состоит в описании указателя общего типа, когда заранее не известен тип объекта, на который он будет ссылаться.

3.3.2. Конструирование новых типов

Для создания новых типов в Си можно использовать конструкции массива, указателя и структуры.

Массивы

Описание массива в Си состоит из имени базового типа, названия массива и его размера, который указывается в квадратных скобках. Размер массива обязательно должен быть целочисленной константой или константным выражением. Примеры:

```
int a[10];
char c[256];
double d[1000];
```

В первой строке описан массив целых чисел из 10 элементов. Подчеркнем, что нумерация в Си всегда начинается с нуля, так что индексы элементов массива изменяются в пределах от 0 до 9. Во второй строке описан массив символов из 256 элементов (индексы в пределах $0 \ldots 255$), в третьей — массив вещественных чисел из 1000 элементов (индексы в пределах $0 \ldots 999$). Для доступа к элементу массива указывается имя массива и индекс элемента в квадратных скобках, например,

```
a[0], c[255], d[123].
```

Оператор sizeof возвращает размер всего массива в байтах, а не в элементах массива. В данном примере

$\text{sizeof(a)} = 10 \cdot \text{sizeof(int)} = 40,$
$\text{sizeof(c)} = 256 \cdot \text{sizeof(char)} = 256,$
$\text{sizeof(d)} = 1000 \cdot \text{sizeof(double)} = 8000.$

Указатели

Указатели — это переменные, которые хранят адреса объектов. Указатели — фамильная принадлежность языка Си. В неявном виде указатели присутствовали и в других языках программирования, но в Си они используются гораздо чаще, а работа с указателями организована максимально просто.

При описании указателя надо задать тип объектов, адреса которых будут содержаться в нем. Перед именем указателя при описании ставится звездочка, чтобы отличить его от обычной переменной. Примеры описаний указателей:

```
int *a, *b, c, d;
char *e;
void *f;
```

В первой строке описаны указатели a и b на тип int и простые переменный c и d типа int (c и d — не указатели!).

С указателями возможны следующие два действия:

1) присвоить указателю адрес некоторой переменной. Для этого используется операция взятия адреса, которая обозначается амперсендом &. Например, строка

```
a = &c;
```

указателю a присваивает значение адреса переменной c;

2) получить объект, адрес которого содержится в указателе; для этого используется операция звездочка '*', которая записывается перед указателем. (Заметим, что звездочкой обозначается также операция умножения.) Например, строка

```
d = *a;
```

присваивает переменной d значение целочисленной переменной, адрес которой содержится в a. Так как ранее указателю a был присвоен адрес переменной c, то в результате переменной d присваивается значение c, т.е. данная строка эквивалентна следующей:

```
d = c;
```

Ниже будут рассмотрены также арифметические операции с указателями, которые в языке Си чрезвычайно важны.

Сложные описания

Конструкции массива и указателя при описании типа можно применять многократно в произвольном порядке. Кроме того, можно описывать прототип функции. Таким образом можно строить сложные описания вроде «массив указателей», «указатель на указатель», «указатель на массив», «функция, возвращающая значение типа указатель», «указатель на функцию» и т.д. Правила здесь таковы:

- для группировки можно использовать круглые скобки, например, описание

```
int *(x[10]);
```

означает «массив из 10 элементов типа указатель на int»;

- при отсутствии скобок приоритеты конструкций описания распределены следующим образом:
  - операция * определения указателя имеет самый низкий приоритет. Например, описание

```
int *x[10];
```

означает «массив из 10 элементов типа указатель на int». Здесь к имени переменной x сначала применяется операция определения массива [] (квадратные скобки), поскольку она имеет более высокий приоритет, чем звездочка. Затем к полученному массиву применяется операция определения указателя. В результате получается «массив указателей», а не указатель на массив! Если нам нужно определить указатель на массив, то следует использовать круглые скобки при описании:

```
int (*x)[10];
```

Здесь к имени x сначала применяется операция * определения указателя;

  - операции определения массива [] (квадратные скобки после имени) и определения функции (круглые скобки после имени) имеют одинаковый приоритет, более высокий, чем звездочка. Примеры:

```
int f();
```

Описан прототип функции f без аргументов, возвращающей значение типа int.

```
int (*f())[10];
```

Описан прототип функции *f* без аргументов, возвращающей значение типа указатель на массив из 10 элементов типа int;

- последний пример уже не является очевидным. Общий алгоритм разбора сложного описания можно охарактеризовать как «чтение изнутри». Сначала находим описываемое имя. Затем определяем, какая операция применяется к имени первой. Если нет круглых скобок для группировки, то это либо определение указателя (звездочка слева от имени), либо определение массива (квадратные скобки справа от имени), либо определение функции (круглые скобки справа от имени). Таким образом получается первый шаг сложного описания. Затем находим следующую операцию описания, которая применяется к уже выделенной части сложного описания, и повторяем это до тех пор, пока не исчерпаем все описание. Проиллюстрируем этот алгоритм на примере:

```
void (*a[100])(int x);
```

Описывается переменная `a`. К ней сначала применяется операция описания массива из 100 элементов, далее — определение указателя, далее — функция от одного целочисленного аргумента `x` типа `int`, наконец — определение возвращаемого типа `int`. Описание читается следующим образом:

1) `a` — это
2) массив из `100` элементов типа
3) указатель на
4) функцию с одним аргументом `x` типа `int`, возвращающую значение типа
5) `void`.

Ниже расставлены номера операций в порядке их применения в описании переменной `a`:

```
void (*  a  [100])(int x);
5)    3) 1) 2)    4)
```

Строки

Специального типа данных «строка» в Си нет. Строки представляются массивами символов (а символы — их числовыми кодами, см. раздел 1.4.3). Последним символом массива, представляющего строку, должен быть символ с нулевым кодом. Пример:

```
char str[10];
str[0] = 'e'; str[1] = '2';
str[2] = 'e'; str[3] = '4';
str[4] = 0;
```

Описан массив str из 10 символов, который может представлять строку длиной не более 9, поскольку один элемент должен быть зарезервирован для терминирующего нуля. Далее в массив str записывается строка `"e2e4"`. Строка терминируется нулевым символом. Всего запись строки использует 5 первых элементов массива str с индексами $0 \ldots 4$. Последние 5 элементов массива не используются. Массив можно инициализировать непосредственно при описании, например

```
char t[] = "abc";
```

Здесь мы не указываем в квадратных скобках размер массива t, компилятор его вычисляет сам. После операции присваивания записана строковая константа `"abc"`, которая заносится в массив t. В результате компилятор создает массив t из четырех элементов, поскольку на строку отводится 4 байта, включая терминирующий ноль.

Строковые константы заключаются в Си в двойные апострофы, в отличие от символьных, которые заключаются в одинарные. Значением строковой константы является адрес ее первого символа. Когда компилятор встречает строковую константу в программе, он записывает ее текст в область статической памяти, обычно защищенную от изменения, и использует этот адрес. Например, в результате следующего описания

```
const char *s = "abcd";
```

создается указатель s, а также строка символов `"abcd"`, строка помещается в область статической памяти, защищенную от изменения, а в указатель s помещается адрес начала строки. Строка содержит 5 элементов: коды символов `abcd` и теминирующий нулевой байт.

Модификатор const

Константы в Си можно задавать двумя способами:

с помощью директивы #define препроцессора. Например, строка

```
#define MILLENIUM 1000
```

задает символическое имя MILLENIUM для константы 1000. Препроцессор всюду в тексте заменяет это имя на константу 1000, используя текстовую подстановку. Это не очень хороший способ, поскольку при таком задании отсутствует контроль типов;

с помощью модификатора const. При описании любой переменной можно добавить модификатор типа const. Например, вместо #define можно использовать следующее описание:

```
const int MILLENIUM = 1000;
```

Модификатор const означает, что переменная MILLENIUM является константой, т.е. менять ее значение нельзя. Попытка присвоить новое значение константе приведет к ошибке компиляции:

```
MILLENIUM = 100; // Ошибка: константу
                 //         нельзя изменять
```

При описании указателя модификатор const, записанный до звездочки, означает, что описан указатель на константный объект, т.е. на объект, менять который нельзя или запрещено. Например, в строке

```
const char *p;
```

описан указатель на константную строку (массив символов, менять который запрещено).

Указатели на константные объекты используются в Си чрезвычайно часто. Причина состоит в том, что константный указатель позволяет прочесть объект и при этом гарантирует, что объект не будет испорчен в результате ошибки программирования, т.к. константный указатель не дает возможности изменить объект.

Константный указатель ссылается на константный объект, однако, содержимое самого указателя может изменяться. Например, следующий фрагмент вполне корректен:

```
const char *str = "e2e4";
. . .
str = "c7c5";
```

Здесь константный указатель str сначала содержит адрес константной строки “e2e4”. Затем в него записывается адрес другой константной строки “c7c5”.

В Си можно также описать указатель, значение которого не может быть изменено; для этого модификатор const указывается *после звездочки*. Например, фрагмент кода

```
int i;
int * const p = &i;
```

навечно записывает в указатель p адрес переменной i, перенаправить указатель p на другую переменную уже нельзя. Строка

```
p = &n;
```

является ошибкой, т.к. указатель p — константа, а константе нельзя присвоить новое значение. Указатели, значения которых изменять нельзя, используются в Си значительно реже, в основном при заполнении константных таблиц.

Модификатор volatile

Слово volatile в переводе означает “изменчивый, непостоянный”. В Си к описанию переменной следует добавлять слово *volatile*, если ее значение может изменяться не в результате выполнения программы, а из-за каких-либо внешних событий. Например, переменная может измениться при выполнении программы-обработчика аппаратного прерывания (см. раздел 2.5). Другой причиной «внезапного» изменения значения переменной может быть переключение между нитями при параллельном программировании (см. 2.6.2) и модификация переменной в параллельной нити.

Необходимо обязательно сообщать компилятору о таких изменчивых переменных. Дело в том, что процессор выполняет все действия

с регистрами, а не с элементами памяти. Оптимизирующий компилятор держит значения большинства переменных в регистрах, сводя к минимуму обращения к памяти. Непостоянная переменная может изменить свое значение в памяти, но программа будет по-прежнему использовать значение в регистре, которое осталось прежним. Из-за этого выполнение программы нарушится. Модификатор *volatile* запрещает даже временно помещать переменную в регистр процессора.

Пример описания переменной:

```
volatile int inputPort;
```

Здесь мы описываем целочисленную переменную *inputPort* и сообщаем компилятору, что ее значение может внезапно меняться в результате каких-либо внешних событий. Этим мы запрещаем компилятору помещать переменную в регистр процессора в целях оптимизации программы.

Оператор typedef

В языке Си можно задать имя типа, если его описание достаточно громоздко и его не хочется повторять много раз. В дальнейшем можно использовать имя типа при описании переменных. Для определения типа применяется оператор *typedef*. Синтаксически оператор *typedef* аналогичен обычному описанию переменной, к которому в самом начале добавлено слово *typedef*. При этом вместо переменной определяется имя нового типа. Сравните следующее описание переменной "real" и определение нового типа "Real":

```
double real;          // Описание переменной real
typedef double Real; // Определение нового типа Real,
                      // эквивалентного типу double.
```

Мы как бы описываем переменную, добавляя к описанию слово *typedef*. При этом описываемое имя становится именем нового типа. Его можно использовать затем для задания переменных:

```
Real x, y, z;
```

Чаще всего определение типов с помощью *typedef* используют, когда описание типа достаточно громоздко. Оператор *typedef* позволяет задать его только один раз, что облегчает исправление программы при необходимости. Например, следующая строка определяет тип *callback* как указатель на функцию с одним целым параметром, возвращающую значение логического типа:

```
typedef bool (*callback)(int);
```

Строка, описывающая три переменные p, q, r,

```
callback p, q, r;
```

эквивалентна строке

```
bool (*p)(int), (*q)(int), (*r)(int);
```

но первая строка, конечно, понятнее и нагляднее.

Еще одна цель использования оператора *typedef* состоит в том, чтобы сделать текст программы менее зависимым от особенностей конкретной архитектуры (разрядности процессора, конкретного Си-компилятора и т.п.). Например, в старых Си-компиляторах, которые использовались для 16-разрядных процессоров Intel 80286, существовали так называемые близкие (*near*) и далекие (*far*) указатели. В эталонном языке Си ключевых слов *near* и *far* нет, они использовались лишь в Си-компиляторах для Intel 80286 как расширение языка. Поэтому, чтобы тексты программ не зависели от компилятора, в системных h-файлах с помощью оператора *typedef* определялись имена для типов указателей, а в текстах программ использовались не типы эталонного языка Си, а введенные имена типов. Например, тип "далекий указатель на константную строку" в соответствии с соглашениями фирмы Microsoft называется LPCTSTR (Long Pointer to Constant Text STRing). При использовании 16-разрядного компилятора он определяется в системных h-файлах как

```
typedef const char far *LPCTSTR;
```

в 32-разрядной архитектуре он определяется без ключевого слова far (поскольку в ней все указатели «далекие»):

```
typedef const char *LPCTSTR;
```

Во всех программах указатели на константные строки описываются как имеющие тип LPCTSTR:

```
LPCTSTR s;
```

благодаря этому программы Microsoft можно использовать как в 16-разрядной, так и в 32-разрядной архитектуре.

3.4. Выражения

Выражения в Си составляются из переменных или констант, к которым применяются различные операции. Для указания порядка операций можно использовать круглые скобки.

Отметим, что, помимо обычных операций, таких, как сложение или умножение, в Си существует ряд операций, несколько непривычных для начинающих. Например, запятая и знак равенства (оператор присваивания) являются операциями в Си; помимо операции сложения $+$, есть еще операция «увеличить на» $+=$ и операция увеличения на единицу $++$. Зачастую они позволяют писать эстетически красивые, но не очень понятные для начинающих программы.

Впрочем, эти непривычные операции можно не использовать, заменяя их традиционными.

3.4.1. Оператор присваивания

Оператор присваивания является основой любого алгоритмического языка (см. раздел 1.3). В Си он записывается с помощью символа равенства, например, строка

```
x = 100;
```

означает присвоение переменной x значения 100. Для сравнения двух значений используется двойное равенство $==$, например, строка

```
bool f = (2 + 2 == 5);
```

присваивает логической переменной f значение false (поскольку $2+2$ не равно пяти, логическое выражение в скобках ложно).

Непривычным для начинающих может быть то, что оператор присваивания "=" в Си — бинарная операция, такая же, как, например,

сложение или умножение. Значением операции присваивания = является значение, которое присваивается переменной, стоящей в левой части. Это позволяет использовать знак присваивания внутри выражения, например,

```
x = (y = sin(z)) + 1.0;
```

Здесь в скобках стоит выражение `y = sin(z)`, в результате вычисления которого переменной y присваивается значение $\sin z$. Значением этого выражения является значение, присвоенное переменной y, т.е. $\sin z$. К этому значению затем прибавляется единица, т.е. в результате переменной x присваивается значение $\sin z + 1$.

Выражения, подобные приведенному в этом примере, иногда используются, когда необходимо запомнить значение *подвыражения* (в данном случае `sin(z)`) в некоторой переменной (в данном случае y), чтобы затем не вычислять его повторно. Еще один пример:

```
n = (k = 3) + 2;
```

В результате переменной k присваивается значение 3, а переменной n — значение 5. Конечно, в нормальных программах такие выражения не встречаются.

3.4.2. Арифметические операции

К четырем обычным арифметическим операциям сложения +, вычитания −, умножения * и деления / в Си добавлена операция нахождения остатка от деления первого целого числа на второе, которая обозначается символом процента %. Приоритет у операции вычисления остатка % такой же, как и у деления или умножения. Отметим, что операция % перестановочна с операцией изменения знака (унарным минусом), например, в результате выполнения двух строк

```
x = -(5 % 3);
y = (-5) % 3;
```

обеим переменным x и y присваивается отрицательное значение -2.

3.4.3. Операции увеличения и уменьшения

В Си добавлены операции увеличения и уменьшения на единицу, которые, к примеру, очень удобно применять к счетчикам. Операция увеличения записывается с помощью двух знаков сложения $++$, операция уменьшения — с помощью двух минусов $--$. Например, операция $++$, примененная к целочисленной переменной i, увеличивает ее значение на единицу:

$$++i; \quad \text{эквивалентно} \quad i = i + 1;$$

Операции увеличения и уменьшения на единицу можно применять только к *дискретным* типам — целочисленным переменным различного вида и указателям. Операцию нельзя применять к вещественным переменным! Например, следующий фрагмент программы является ошибочным:

```
double x;
. . .
++x;  // Ошибка! Операция ++ неприменима
      //          к вещ. переменной
```

Операция $++$ увеличивает значение переменной на «минимальный атом». Так как для вещественных переменных такого «атомарного» значения нет, операции увеличения и уменьшения для них запрещены.

Для *указателей* операция $++$ увеличивает значение переменной на *размер одного элемента* того типа, на который ссылается указатель. Для указателя «атомом» является один элемент заданного типа, поэтому размер одного элемента и является шагом изменения значения указателя. Это очень естественно, т.к. после увеличения указатель будет содержать адрес *следующего* элемента данного типа, а после уменьшения — адрес *предыдущего* элемента. Пример:

```
double a[100];
double *p = &(a[15]); // в p записывается адрес
                      // элемента массива a[15]
++p; // в p будет адрес элемента a[16]
     // (адрес увеличивается на sizeof(double) == 8)
```

Описаны массив a вещественных чисел типа double и указатель p на элементы типа double. При описании указателя p в него заносится начальное значение, равное адресу элемента $a[15]$ массива a. После выполнения операции увеличения $++$ в переменной p будет содержаться адрес следующего элемента $a[16]$. Физически содержимое переменной p увеличивается на размер одного элемента типа double, т.е. на 8.

Операции увеличения $++$ и уменьшения $--$ на единицу имеют *префиксную* и *суффиксную* формы. В префиксной форме операция записывается перед переменной, как в приведенных выше примерах. В суффиксной форме операция записывается после переменной:

```
++x;    // Префиксная форма
x--;    // Суффиксная форма
```

Разница между префиксной и суффиксной формами проявляется только при вычислении сложных выражений. Если используется *префиксная* форма операции $++$, то *сначала переменная увеличивается*, и только после этого ее новое значение используется в выражении. При использовании *суффиксной* формы значение переменной *сначала используется* в выражении и только затем увеличивается. Примеры:

```
int x = 5, y = 5, a, b;
a = (++x) + 2; // переменной a присваивается значение 8
b = (y++) + 2; // переменной b присваивается значение 7
```

С логической точки зрения, префиксная операция более естественна (при использовании суффиксной формы надо сперва вычислить сложное выражение и только затем вернуться к увеличению переменной, т.е. операция $++$ выполняется не в момент ее использования, а как бы откладывается на потом). Забегая вперед, отметим, что это различие весьма существенно при программировании на C++ в случае переопределения операторов увеличения для классов. Тем не менее, в большинстве книг по Си суффиксная форма используется чаще (скорее всего, эта традиция, связаная с эстетикой текста).

Дадим два совета (возможно, не бесспорные) по использованию операций $++$ и $--$:

- никогда не применяйте эти операции в сложных выражениях! Ничего, кроме путаницы, это не дает. Например, вместо фрагмента

```
double *p, x, y;
. . .
y = *p++ + x;
```

лучше использовать фрагмент

```
double *p, x, y;
. . .
y = *p + x;
++p;
```

С точки зрения компилятора, они абсолютно эквивалентны, но второй фрагмент проще и понятнее (и, значит, вероятность ошибки программирования меньше);

- всегда отдавайте предпочтение префиксной форме операций ++ и --. Например, вместо фрагмента

```
int x, y;
. . .
x++; y--; // Используется суффиксная форма
```

лучше использовать фрагмент

```
int x, y;
. . .
++x; --y; // Лучше применять префиксную форму
```

3.4.4. Операции «увеличить на», «домножить на» и т.п.

В большинстве алгоритмов при выполнении операции сложения чаще всего переменная-результат операции совпадает с первым аргументом:

```
x = x + y;
```

Здесь складываются значения двух переменных x и y, результат помещается в первую переменную x. Таким образом, значение переменной x увеличивается на значение y. Подобные фрагменты встречаются в программах гораздо чаще, чем фрагменты вида

```
x = y + z;
```

где аргументы и результат различны. Рассмотрим, например, фрагмент программы, вычисляющий сумму элементов массива вещественных чисел (забегая вперед, мы используем в нем конструкцию цикла "пока"):

```
double a[100];
double s;
int i;
. . .
s = 0.0;
i = 0;
while (i < 100) {
    s = s + a[i];
    ++i;
}
```

Здесь сумма элементов массива накапливается в переменной s. В строке

```
s = s + a[i];
```

к сумме s прибавляется очередной элемент массива $a[i]$, т.е. значение s увеличивается на $a[i]$. В Си существует сокращенная запись операции увеличения:

```
s += a[i];
```

Оператор += читается как "увеличить на". Строка

```
x += y;     // Увеличить значение x на y
```

эквивалентна в Си строке

```
x = x + y; // x присвоить значение x + y,
```

но короче и нагляднее.

Оператор вида $\circ=$ существует для любой операции $\circ$, допустимой в Си. Например, для арифметических операций $+$, $-$, *, /, % можно использовать операции

+=	увеличить на,
−=	уменьшить на,
*=	домножить на,
/=	поделить на,
%=	поделить с остатком на;

к примеру, строка

```
x *= 2.0;
```

удваивает значение вещественной переменной x.

Операторы вида $\circ=$ можно использовать даже для операций $\circ$, которые записываются двумя символами. Например, операции логического умножения и сложения (см. раздел 1.4.4) записываются в Си как && (двойной амперсенд) и || (двойная вертикальная черта). Соответственно, логические операторы "домножить на" и "увеличить на" записываются в виде &&= и ||=, например,

```
bool x, y;
x &&= y;    // эквивалентно x = x && y;
x ||= y;    // эквивалентно x = x || y;
```

3.4.5. Логические операции

Логические операции и выражения были подробно рассмотрены в разделе 1.4.4. В Си используются следующие обозначения для логических операций:

||	логическое "или" (логическое сложение)
&&	логическое "и" (логическое умножение)
!	логическое "не" (логическое отрицание)

Логические константы "истина" и "ложь" обозначаются через true и false (это ключевые слова языка). Примеры логических выражений:

```
bool a, b, c, d;
int x, y;

a = b || c;              // логическое "или"
d = b && c;              // логическое "и"
a = !b;                  // логическое "не"
a = (x == y);            // сравнение в правой части
a = false;               // ложь
b = true;                // истина
c = (x > 0 && y != 1); // с истинно, когда
                         // оба сравнения истинны
```

Самый высокий приоритет у операции логического отрицания, затем следует логическое умножение, самый низкий приоритет у логического сложения.

Чрезвычайно важной особенностью операций логического сложения и умножения является так называемое «сокращенное вычисление» результата. А именно, при вычислении результата операции логического сложения или умножения всегда сначала вычисляется значение первого аргумента. Если оно истинно в случае логического сложения или ложно в случае логического умножения, то второй аргумент операции не вычисляется вовсе! Результат операции полагается истинным в случае логического сложения или ложным в случае логического умножения. Подробно это рассмотрено в разделе 1.4.4.

3.4.6. Операции сравнения

Операция сравнения сравнивает два выражения. В результате вырабатывается логическое значение — *true* или *false* (истина или ложь) в зависимости от значений выражений. Примеры:

```
bool res;
int x, y;
res = (x == y); // true, если x равно y, иначе false
res = (x == x); // всегда true
res = (2 < 1);  // всегда false
```

Операции сравнения в Си обозначаются следующим образом:

==	равно,	!=	не равно,
>	больше,	>=	больше или равно,
<	меньше,	<=	меньше или равно.

3.4.7. Побитовые логические операции

Кроме обычных логических операций, в Си имеются побитовые логические операции, которые выполняются независимо для каждого отдельного бита операндов. Побитовые операции имеют следующие обозначения:

- `&` побитовое логическое сложение («или»)
- `|` побитовое логическое умножение («и»)
- `~` побитовое логическое отрицание («не»)
- `^` побитовое сложение по модулю 2 (исключающее «или»)

(Необходимо помнить, что логические операции умножения и сложения записываются с помощью двойных знаков `&&` или `||`, а побитовые — с помощью одинарных.)

Ни в коем случае не используйте побитовые операции в качестве логических условий, это может приводить к непредсказуемым ошибкам!

В основном побитовые операции применяются для манипуляций с битовыми масками. Например, пусть целое число x описывает набор признаков некоторого объекта, состоящий из четырех признаков. Назовем их условно A, B, C, D. Пусть за признак A отвечает нулевой бит слова x (биты в двоичном представлении числа нумеруются справа налево, начиная с нуля). Если бит равен единице (программисты говорят «бит установлен»), то считается, что объект обладает признаком A. За признаки B, C, D отвечают биты с номерами 1, 2, 3. Общепринятая практика состоит в том, чтобы определить константы, отвечающие за соответствующие признаки (их обычно называют *масками*):

```
const int MASK_A = 1;
const int MASK_B = 2;
const int MASK_C = 4;
const int MASK_D = 8;
```

Эти константы содержат единицу в соответствующем бите и нули в остальных битах. Для того чтобы проверить, установлен ли в слове x бит, соответствующий, к примеру, признаку D, используется операция побитового логического умножения. Число x умножается на константу MASK_D; если результат отличен от нуля, то бит установлен, т.е. объект обладает признаком D, если нет, то не обладает. Такая проверка реализуется следующим фрагментом:

```
if ((x & MASK_D) != 0) {
    // Бит D установлен в слове x, т.е.
    // объект обладает признаком D
    . . .
} else {
    // Объект не обладает признаком D
    . . .
}
```

При побитовом логическом умножении константа MASK_D обнуляет все биты слова x, кроме бита D, т.е. как бы вырезает бит D из x. В двоичном представлении это выглядит примерно так:

```
x:           0101110110...10*101
MASK_D:      0000000000...001000
-------------------------------
x & MASK_D:  0000000000...00*000
```

Звездочкой здесь обозначено произвольное значение бита D слова x.

Для установки бита D в слове x используется операция побитового логического сложения:

```
x = (x | MASK_D);   // Установить бит D в слове x
```

Чаще это записывается с использованием операции |= типа "увеличить на" (см. раздел 3.4.4):

```
x |= MASK_D;    // Установить бит D в слове x
```

В двоичном виде это выглядит так:

```
x:           0101110110...10*101
MASK_D:      0000000000...001000
-------------------------------
x | MASK_D:  0101110110...101101
```

Операция побитового отрицания "~" инвертирует биты слова:

```
x:  0101110110...101101
~x: 1010001001...010010
```

Для очистки (т.е. установки в ноль) бита D используется комбинация операций побитового отрицания и побитового логического умножения:

```
x = (x & ~MASK_D);  // Очистить бит D в слове x
```

или, применяя операцию "&=" типа "домножить на":

```
x &= ~MASK_D;   // Очистить бит D в слове x
```

Здесь сначала инвертируется маска, соответствующая биту D,

```
MASK_D:  0000000000...001000
~MASK_D: 1111111111...110111
```

в результате получаются единицы во всех битах, кроме бита D. Затем слово x побитно домножается на инвертированную маску:

```
x:            0101110110...10*101
~MASK_D:      1111111111...110111
x & ~MASK_D: 0101110110...100101
```

В результате в слове x бит D обнуляется, а остальные биты остаются неизменными.

Приоритеты побитовых операций в Си выбраны достаточно странно (они такие же, как у соответствующих логических операций), это иногда приводит к неожиданным ошибкам. Например, если не заключить в скобки операцию побитового умножения в приведенном выше примере, то получится ошибочный результат: строка

```
if (x & MASK_D != 0) {
```

эквивалентна строке

```
if ((x & 1) != 0) {
```

т.е. проверяется бит A, а вовсе не D! Дело в том, что приоритет операции сравнения `!=` выше, чем операции побитового умножения `&`, т.е. в приведенной строке скобки неявно расставлены так:

```
if (x & (MASK_D != 0)) {
```

Выражение (MASK_D != 0) истинно и, таким образом, равно единице, поэтому строка эквивалентна

```
if (x & 1) {
```

что, в свою очередь, эквивалентно более канонической записи:

```
if ((x & 1) != 0) {
```

Чтобы избежать подобных ошибок, всегда заключайте все побитовые операции в скобки.

Побитовую операцию ^ называют сложением по модулю 2, а также «исключающим или». Часто для нее используется аббревиатура XOR, от eXclusive OR. «Таблица сложения» для этой операции выглядит следующим образом:

```
0 ^ 0 = 0,      0 ^ 1 = 1,
1 ^ 0 = 1,      1 ^ 1 = 0.
```

Пусть x — произвольное целое число, m — маска, т.е. число, в котором интересующие программиста биты установлены в единицу, остальные в ноль. В результате выполнения операции XOR

```
x = (x ^ m);
```

или, в более удобной записи,

```
x ^= m;
```

биты в слове x, соответствующие установленным в единицу битам маски m, изменяются на противоположные (инвертируются). Биты слова x, соответствующие нулевым битам маски, не меняют своих значений. Пример:

```
x:      101101...1001011110
m:      000000...0011111100
x ^ m:  101101...1010100010
```

Операция XOR обладает замечательным свойством: если дважды прибавить к слову x произвольную маску m, то в результате получается исходное значение x:

```
((x ^ m) ^ m) == x
```

Прибавление к слову x маски m можно трактовать как шифрование x, ведь в результате биты x, соответсвующие единичным битам маски m, инвертируются. Если маска достаточно случайная, то в результате x тоже принимает случайное значение. Процедура расшифровки в данном случае совпадает с процедурой шифрования и состоит в повторном прибавлении маски m.

3.4.8. Операции сдвига

Оперции сдвига применяются к целочисленным переменным: двоичный код числа сдвигается вправо или влево на указанное количество позиций. Сдвиг вправо обозначается двумя символами «больше» >>, сдвиг влево — двумя символами «меньше» <<. Примеры:

```
int x, y;
. . .
x = (y >> 3);   // Сдвиг на 3 позиции вправо
y = (y << 2);   // Сдвиг на 2 позиции влево
```

При сдвиге влево на k позиций младшие k разрядов результата устанавливаются в ноль. Сдвиг влево на k позиций эквивалентен умножению на число 2^k. Сдвиг вправо более сложен, он по-разному определяется для беззнаковых и знаковых чисел. При сдвиге вправо беззнакового числа на k позиций освободившиеся k старших разрядов устанавливаются в ноль. Например, в двоичной записи имеем:

```
unsigned x;
x      = 110111000...10110011
x >> 3 = 000110111000...10110
```

Сдвиг вправо на k позиций соответствует целочисленному делению на число 2^k.

При сдвиге вправо чисел со знаком происходит так называемое «расширение знакового разряда». Именно, если число неотрицательно, т.е. старший, или знаковый, разряд числа равен нулю, то происходит обычный сдвиг, как и в случае беззнаковых чисел. Если же число отрицательное, т.е. его старший разряд равен единице, то

освободившиеся в результате сдвига k старших разрядов устанавливаются в единицу. Число, таким образом, остается отрицательным. При $k = 1$ это соответствует делению на 2 только для отрицательных чисел, не равных -1. Для числа -1, все биты двоичного кода которого равны единице, сдвиг вправо не приводит к его изменению. Пример (используется двоичная запись):

```
int x;
x      = 110111000...10110011
x >> 3 = 111110111000...10110
```

В программах лучше не полагаться на эту особенность сдвига вправо для знаковых чисел и использовать конструкции, которые заведомо одинаково работают для знаковых и беззнаковых чисел. Например, следующий фрагмент кода выделяет из целого числа составляющие его байты и записывает их в целочисленные переменные $x0$, $x1$, $x2$, $x3$, младший байт в $x0$, старший в $x3$. При этом байты трактуются как неотрицательные числа. Фрагмент выполняется одинаково для знаковых и беззнаковых чисел:

```
int x;
int x0, x1, x2, x3;
. . .
x0 = (x & 255);
x1 = ((x >> 8) & 255);
x2 = ((x >> 16) & 255);
x3 = ((x >> 24) & 255);
```

Здесь число 255 играет роль маски, см. раздел 3.4.7. При побитовом умножении на эту маску из целого числа «вырезается» его младший байт, поскольку маска 255 содержит единицы в младших восьми разрядах. Чтобы получить байт числа x с номером n, $n = 0, 1, 2, 3$, мы сначала сдвигаем двоичный код x вправо на $8n$ разрядов, таким образом, байт с номером n становится младшим. Затем с помощью побитового умножения вырезается младший байт.

3.4.9. Арифметика указателей

С указателями можно выполнять следующие операции:

- сложение указателя и целого числа, результат — указатель;
- увеличение или уменьшение переменной типа указатель, что эквивалентно прибавлению или вычитанию единицы;
- вычитание двух указателей, результат — целое число.

Прибавление к указателю p целого числа n означает увеличение адреса, который содержится в переменной p, на суммарный размер n элементов того типа, на который ссылается указатель. Указатель как бы сдвигается на n элементов вправо, если считать, что индексы элементов массива возрастают слева направо. Аналогично вычитание целого числа n из указателя означает сдвиг указателя влево на n элементов. Пример:

```
int *p, *q;
int a[100];
p = &(a[5]); // записываем в p адрес 5-го
             //     элемента массива a
p += 7;      // p будет содержать адрес 12-го эл-та
q = &(a[10]);
--q;         // q содержит адрес элемента a[9]
```

Значение указателя при прибавлении к нему целого числа n увеличивается на произведение n на количество байтов, занимаемое одним элементом того типа, на который ссылается указатель. В программировании это называют *масштабированием*.

Разность двух указателей — это количество элементов данного типа, которое умещается между двумя адресами. Результатом вычитания указателей является целое число. Физически оно вычисляется как разность значений двух адресов, деленная на размер одного элемента заданного типа. Операции сложения указателя с целым числом и разности двух указателей взаимно обратны:

```
int *p, *q;
int a[100];
int n;
p = &(a[5]);
q = &(a[12]);
n = q - p;       // n == 7
q = p + n;       // q == &(a[12])
```

Подчеркнем, что указатели нельзя складывать! В отличие от разности указателей, операция сложения указателей (т.е. сложения адресов памяти) абсолютно бессмысленна.

```
int *p, *q, *r;
int a[100];
p = &(a[5]);
q = &(a[12]);
r = p + q;  // Ошибка! Указатели нельзя складывать.
```

3.4.10. Связь между указателями и массивами

В языке Си имя массива `a` является указателем на его первый элемент, т.е. выражения `a` и `&(a[0])` эквивалентны. Учитывая арифметику указателей, получаем эквивалентность следующих выражений:

$$\texttt{a[i]} \quad \sim \quad \texttt{*(a+i)}$$

Действительно, при прибавлении к `a` целого числа `i` происходит сдвиг на `i` элементов вправо. Поскольку имя массива является адресом его начального элемента, получается адрес `i`-го элемента массива `a`. Применяя операцию звездочка `*`, получаем сам элемент `a[i]`. Точно так же эквивалентны выражения

$$\texttt{\&(a[i])} \quad \sim \quad \texttt{a+i} \quad \text{(адрес эл-та } \texttt{a[i]}\text{)}.$$

Эта особенность арифметики указателей позволяет вообще не использовать квадратные скобки, т.е. обращение к элементу массива; вместо этого можно использовать указатели и операцию звездочка `*`.

Обратно, пусть `p` — указатель. Синтаксис языка Си позволяет трактовать его как адрес начала массива и применять к нему операцию доступа к элементу массива с заданным индексом. Эквивалентны следующие выражения:

$$\texttt{p[i]} \quad \sim \quad \texttt{*(p+i)}$$

Таким образом, выбор между массивами и указателями — это выбор между двумя эквивалентными способами записи программ. Указатели, возможно, нравятся системным программистам, которые

привыкли к работе с адресами объектов. Массивы больше отвечают традиционному стилю. В объектно-ориентированных языках, таких как Java или C#, указателей либо нет вовсе, либо их разрешено использовать лишь в специфических ситуациях. Массивы же присутствуют в подавляющем большинстве алгоритмических языков.

Для иллюстрации работы с массивами и с указателями приведем два фрагмента программы, суммирующие элементы массива.

```
double a[100], s;
int i;
. . .
s = 0.0;
i = 0;

while (i < 100) {
    s += a[i];
    ++i;
}
```

```
double a[100], s;
double *p, *q;
. . .
s = 0.0;
p = a;  // адрес начала массива
q = a + 100; // адрес за концом
while (p < q) {
    s += *p;
    ++p;
}
```

3.4.11. Операция приведения типа

Операция приведения типа (type cast) является одной из самых важных в Си. Без знакомства с синтаксисом этой операции (весьма непривычного для начинающих) и сознательного ее использования написать на Си что-нибудь более или менее полезное невозможно.

Операция приведения типа используется, когда значение одного типа преобразуется к другому типу, в том случае, если существует некоторый разумный способ такого преобразования. Операция обозначается именем типа, заключенным в круглые скобки; она записывается *перед* ее единственным аргументом. Рассмотрим два примера. Пусть требуется преобразовать целое число к вещественному типу. Как известно, целые и вещественные числа по-разному представляются в компьютере, см. раздел 3.3.1. Тем не менее, существует однозначный способ преобразования целого числа типа *int* к вещественному типу *double*. В первом примере значение целой переменной n приводится к вещественному типу и присваивается вещественной переменной x:

```
double x;
```

```
int n;
. . .
x = (double) n; // Операция приведения к типу double
```

В данном случае никакой потери информации не происходит, поэтому такое приведение допустимо и по умолчанию:

```
x = n; // Эквивалентно x = (double) n;
```

Во втором примере вещественное значение преобразуется к целому типу. При этом дробная часть вещественного числа отбрасывается, а знак числа сохраняется:

```
double x, y;
int n, k;
. . .
x = 3.7;
y = (-1.5);
n = (int) x;     // n присваивается значение 3
k = (int) y;     // k присваивается значение -1
```

В результате выполнения операции приведения вещественного числа к целому типу происходит отбрасывание дробной части числа, т.е. потеря информации. Поэтому, если использовать операцию приведения типа *неявно* (т.е. в результате простого присваивания целой переменной вещественного значения), например,

```
double x; int n;
. . .
n = x; // неявное приведение вещественного к целому
```

то компилятор обязательно выдаст предупреждение (или даже ошибку, если компилятор строгий). Поэтому так писать ни в коем случае нельзя! Когда используется *явное* приведение типа, компилятору сообщается, что это не случайная ошибка, а намеренное приведение вещественного значения к целому типу, при котором дробная часть отбрасывается. При этом компилятор никаких предупреждений не выдает.

Операция приведения типа чаще всего используется для преобразования *указателей*. Например, стандартная функция захвата динамической памяти *malloc* возвращает указатель общего типа `void*`

(см. раздел 3.7.3). Значение указателя обобщенного типа нельзя присвоить указателю на конкретный тип (язык C++ запрещает такие присвоения, Си-компиляторы иногда разрешают преобразования указателей по умолчанию, выдавая предупреждения, — но в любом случае это дурной стиль!). Для преобразования указателей разного типа нужно использовать операцию приведения типа в явном виде. В следующем примере в динамической памяти захватывается участок размером в 400 байт, его адрес присваивается указателю на массив из 100 целых чисел:

```
int *a; // Описываем указатель на массив типа int
. . .
// Захватываем участок памяти размером в 400 байт
// (поскольку sizeof(int) == 4), приводим указатель
// на него от типа void* к типу int* и присваиваем
// приведенное значение указателю a:
a = (int*) malloc(100 * sizeof(int));
```

Отметим, что допустимо неявное преобразование любого указателя к указателю обобщенного типа `void*`. Обратное, как указано выше, считается грубой ошибкой в C++ и дурным стилем (возможно, сопровождаемым предупреждением компилятора) в Си:

```
int *a;        // Указатель на целое число
void *p;       // Указатель обобщенного типа
. . .
a = p;         // Ошибка! В C++ запрещено неявное
               // приведение типа от void* к int*
a = (int*) p; // Корректно: явное приведение типа

p = a;  // Корректно: любой указатель можно
        // неявно привести к обобщенному
```

3.5. Управляющие конструкции

Управляющие конструкции позволяют организовывать циклы и ветвления в программах. В Си всего несколько конструкций, причем половину из них можно не использовать (они реализуются через остальные).

3.5.1. Фигурные скобки

Фигурные скобки позволяют объединить несколько элементарных операторов в один составной оператор, или блок. Во всех синтаксических конструкциях составной оператор можно использовать вместо простого.

В Си в начало блока можно помещать описания локальных переменных. Локальные переменные, описанные внутри блока, создаются при входе в блок и уничтожаются при выходе из него.

В C++ локальные переменные можно описывать где угодно, а не только в начале блока. Тем не менее, они, так же как и в Си, автоматически уничтожаются при выходе из блока.

Приведем фрагмент программы, обменивающий значения двух вещественных переменных:

```
double x, y;
. . .
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

Здесь, чтобы обменять значения двух переменных x и y, мы сначала запоминаем значение x во вспомогательной переменной *tmp*. Затем в x записывается значение y, а в y — сохраненное в *tmp* предыдущее значение x. Поскольку переменная *tmp* нужна только внутри этого фрагмента, мы заключили его в блок и описали переменную *tmp* внутри этого блока. По выходу из блока память, занятая переменной *tmp*, будет освобождена.

3.5.2. Оператор if

Оператор if («если») позволяет организовать ветвление в программе. Он имеет две формы: оператор «если» и оператор «если. . . иначе». Оператор «если» имеет вид

```
if (условие)
    действие;
```

оператор «если... иначе» имеет вид

```
if (условие)
    действие1;
else
    действие2;
```

В качестве условия можно использовать любое выражение логического или целого типа. Напомним, что при использовании целочисленного выражения значению "истина" соответствует любое ненулевое значение. При выполнении оператора «если» сначала вычисляется условное выражение после if. Если оно истинно, то выполняется действие, если ложно, то ничего не происходит. Например, в следующем фрагменте в переменную m записывается максимальное из значений переменных x и y:

```
double x, y, m;
. . .
m = x;
if (y > x)
    m = y;
```

При выполнении оператора «если... иначе» в случае, когда условие истинно, выполняется действие, записанное после if; в противном случае выполняется действие после else. Например, предыдущий фрагмент переписывается следующим образом:

```
double x, y, m;
. . .
if (x > y)
    m = x;
else
    m = y;
```

Когда надо выполнить несколько действий в зависимости от истинности условия, следует использовать фигурные скобки, объединяя несколько операторов в блок, например,

```
double x, y, d;
. . .
```

```
if (d > 1.0) {
    x /= d;
    y /= d;
}
```

Здесь переменные x и y делятся на d только в том случае, когда значение d больше единицы.

Фигурные скобки можно использовать даже, когда после if или else стоит только один оператор. Они улучшают структуру текста программы и облегчают ее возможную модификацию. Пример:

```
double x, y;
. . .
if (x != 0.0) {
    y = 1.0;
}
```

Если нужно будет добавить еще одно действие, выполняемое при условии "x отлично от нуля", то мы просто добавим строку внутри фигурных скобок.

3.5.3. Выбор из нескольких возможностей: if... else if...

Несколько условных операторов типа «если... иначе» можно записывать последовательно (т.е. действие после else может снова представлять собой условный оператор). В результате реализуется *выбор из нескольких возможностей*. Конструкция выбора используется в программировании очень часто. Пример: дана вещественная переменная x, требуется записать в вещественную переменную y значение функции $\mathrm{sign}(x)$:

$$\mathrm{sign}(x) = \begin{cases} -1, & \text{при } x < 0, \\ 1, & \text{при } x > 0, \\ 0, & \text{при } x = 0 \end{cases}$$

Это делается с использованием конструкции выбора:

```
double x, s;
. . .
```

```
if (x < 0.0) {
    s = (-1.0);
}
else if (x > 0.0) {
    s = 1.0;
}
else {
    s = 0.0;
}
```

При выполнении этого фрагмента сперва проверяется условие `x < 0.0`. Если оно истинно, то выполняется оператор `s = (-1.0);` иначе проверяется второе условие `x < 0.0`. В случае его истинности выполняется оператор `s = 1.0`, иначе выполняется оператор `s = 0.0`. Фигурные скобки здесь добавлены для улучшения структурности текста программы.

В любом случае, в результате выполнения конструкции выбора исполняется лишь один из операторов (возможно, составных). Условия проверяются последовательно сверху вниз. Как только находится истинное условие, то производится соответствующее действие и выбор заканчивается.

3.5.4. Пример: решение квадратного уравнения

Рассмотрим простой пример, в котором применяется конструкция «если... иначе»: требуется решить квадратное уравнение

$$ax^2 + bx + c = 0$$

Программа должна ввести с клавиатуры терминала числа a, b, c и затем напечатать ответ. После ввода надо проверить корректность введенных чисел — коэффициент a должен быть отличен от нуля (иначе уравнение перестает быть квадратным, тогда формула решения квадратного уравнения неприменима). В зависимости от знака дискриминанта уравнение может не иметь решений. Программа должна напечатать либо сообщение об отсутствии решений, либо два корня уравнения (возможно, совпадающие в случае нулевого дискриминанта).

Для печати на экран терминала и ввода информации с клавиатуры используются функции ввода-вывода из стандартной библиотеки Си. Отметим, что функции стандартного ввода-вывода не являются частью языка Си: Си не содержит средств ввода-вывода. Однако, любой компилятор обычно предоставляет набор библиотек, в который входит стандартный ввод-вывод. Описания функций ввода-вывода содержатся в заголовочном файле stdio.h, который подключается с помощью строки

```
#include <stdio.h>
```

Мы используем две функции: функцию printf вывода по формату и функцию scanfввода по формату. У обеих этих функций число аргументов переменное, первым аргументом всегда является форматная строка. В случае функции printf обычные символы форматной строки просто выводятся на экран терминала. Например, в рассмотренном ранее примере “Hello, World!” текст выводился на экран с помощью строки прграммы

```
printf("Hello, World!\n'');
```

(Здесь '\n' — символ конца строки, т.е. перевода курсора в начало следующей строки.) Единственным аргументом функции printf в данном случае служит форматная строка.

Кроме обычных символов, форматная строка может включать *символы формата*, которые при выводе заменяются *значениями* остальных аргументов функции printf, начиная со второго аргумента. Для каждого типа данных Си имеются свои форматы. Формат начинается с символа процента '%'. После процента идет необязательный числовой аргумент, управляющий представлением данных. Наконец, далее идет одна или несколько букв, задающих тип выводимых на печать данных. Для вывода чисел можно использовать следующие форматы:

%d вывод целого числа типа int (d — от decimal)
%lf вывод вещ. числа типа double (lf — от long float)

Например, для печати целого числа n можно использовать строку

```
printf("n = %d\n", n);
```

Здесь формат "%d" будет заменен на значение переменной n. Пусть, к примеру, $n = 15$. Тогда при выполнении функции printf будет напечатана строка

```
n = 15
```

При печати вещественного числа компьютер сам решает, сколько знаков после десятичной точки следует напечатать. Если нужно повлиять на представление числа, следует использовать необязательную часть формата. Например, формат

```
%.3lf
```

применяется для печати значения вещественного числа в форме с тремя цифрами после десятичной точки. Пусть значение вещественной переменной x равно единице. Тогда при выполнении функции

```
printf("ответ = %.3lf\n", x);
```

будет напечатана строка

```
ответ = 1.000
```

При вызове функции форматного ввода scanf форматная строка должна содержать *только форматы*. Этим функция scanf отличается от printf. Вместо *значений* печатаемых переменных или выражений, как в функции printf, функция scanf должна содержать *указатели* на вводимые переменные! Для начинающих это постоянный источник ошибок. Необходимо запомнить: функции scanf нужно передавать *адреса* переменных, в которые надо записать введенные значения. Если вместо адресов переменных передать их значения, то функция scanf все равно проинтерпретирует полученные значения как адреса, что при выполнении вызовет попытку записи по некорректным адресам памяти и, скорее всего, приведет к ошибке типа Segmentation fault. Пример: пусть нужно ввести значения трех вещественных переменных a, b, c. Тогда следует использовать фрагмент

```
scanf("%lf%lf%lf", &a, &b, &c);
```

Ошибка, которую часто совершают начинающие: передача функции scanf значений переменных вместо адресов:

```
    scanf("%lf%lf%lf", a, b, c); // Ошибка! Передаются
                                 // значения вместо указателей
```

Помимо стандартной библиотеки ввода-вывода, в Си-программах широко используется стандартная библиотека математических функций. Ее описания содержатся в стандартном заголовочном файле `math.h`, который подключается строкой

```
#include <math.h>
```

Стандартная математическая библиотека содержит математические функции sin, cos, exp, log (натуральный логарифм), fabs (абсолютная величина вещ. числа) и многие другие. Нам необходима функция sqrt, вычисляющая квадратный корень вещественного числа.

Итак, приведем полный текст программы, решающей квадратное уравнение; он содержится в файле “squareEq.cpp”.

```
#include <stdio.h>  // Описания стандартного ввода-вывода
#include <math.h>   // Описания математической библиотеки

int main() {
    double a, b, c; // Коэффициенты уравнения
    double d;       // Дискриминант
    double x1, x2;  // Корни уравнения

    printf("Введите коэффициенты a, b, c:\n");
    scanf("%lf%lf%lf", &a, &b, &c);

    if (a == 0.0) {
        printf("Коэффициент a должен быть ненулевым.\n");
        return 1;   // Возвращаем код некорректного
    }               //                 завершения

    d = b*b - 4.0*a*c;  // Вычисляем дискриминант
    if (d < 0.0) {
        printf("Решений нет.\n");
    }
    else {
        d = sqrt(d); // Квадр. корень из дискриминанта
```

```
        x1 = (-b + d) / (2.0 * a); // Первый корень ур-я
        x2 = (-b - d) / (2.0 * a); // Второй корень ур-я

        // Печатаем ответ
        printf(
            "Решения уравнения: x1 = %lf, x2 = %lf\n",
            x1, x2
        );
    }
    return 0; // Возвращаем код успешного завершения
}
```

Приведем пример выполнения программы:

```
Введите коэффициенты a, b, c:
1 2 -3
Решения уравнения: x1 = 1.000000, x2 = -3.000000
```

Здесь первая и третья строчки напечатаны компьютером, вторая строчка напечатана человеком (ввод чисел заканчивается клавишей перевода строки *Enter*).

3.5.5. Цикл while

Конструкция цикла “пока” соответствует циклу while в Си:

```
while (условие)
    действие;
```

Цикл while называют *циклом с предусловием*, поскольку условие проверяется *перед* выполнением тела цикла.

Цикл while выполняется следующим образом: сначала проверяется *условие*. Если оно истинно, то выполняется *действие*. Затем снова проверяется *условие*; если оно истинно, то снова повторяется *действие*, и так до бесконечности. Цикл завершается, когда *условие* становится ложным. Пример:

```
int n, p;
. . .
p = 1;
```

```
while (2*p <= n)
    p *= 2;
```

В результате выполнения этого фрагмента в переменной p будет вычислена максимальная степень двойки, не превосходящая целого положительного числа n.

Если *условие* ложно с самого начала, то *действие* не выполняется ни разу. Это очень облегчает программирование и делает программу более надежной, поскольку исключительные ситуации автоматически правильно обрабатываются. Так, приведенный выше фрагмент работает корректно при $n = 1$ (цикл не выполняется ни разу).

При ошибке программирования цикл может никогда не кончиться. Чтобы избежать этого, следует составлять программу таким образом, чтобы некоторая ограниченная величина, от которой прямо или косвенно зависит условие в заголовке цикла, монотонно убывала или возрастала после каждого выполнения тела цикла. Это обеспечивает завершение цикла. В приведенном выше фрагменте такой величиной является значение p, которое возрастает вдвое после каждого выполнения тела цикла.

Тело цикла может состоять из одного или нескольких операторов. В последнем случае их надо заключить в фигурные скобки. Советуем заключать тело цикла в фигурные скобки даже в том случае, когда оно состоит всего из одного оператора, — это делает текст программы более наглядным и облегчает его возможную модификацию. Например, приведенный выше фрагмент лучше было бы записать так:

```
int n, p;
. . .
p = 1;
while (2*p <= n) {
    p *= 2;
}
```

Сознательное применение цикла “пока” всегда связано с явной формулировкой инварианта цикла, см. раздел 1.5.2.

Рассмотрим построение цикла “пока” на примере программы вычисления квадратного корня методом деления отрезка пополам.

3.5.6. Пример: вычисление квадратного корня методом деления отрезка пополам

Метод вычисления корня функции с помощью деления отрезка пополам в общем случае уже был рассмотрен в разделе 1.5.2. Пусть надо найти квадратный корень из неотрицательного вещественного числа a с заданной точностью ε. Задача сводится к нахождению корня функции

$$y = x^2 - a$$

на отрезке $[0, b]$, где $b = \max(1, a)$. На этом отрезке функция имеет ровно один корень, поскольку она монотонно возрастает и на концах отрезка принимает значения разных знаков (или нулевое значение при $a = 0$ или $a = 1$).

Идея алгоритма состоит в том, что отрезок делится пополам и выбирается та половина, на которой функция принимает значения разных знаков. Эта операция повторяется до тех пор, пока длина отрезка не станет меньше, чем ε. Концы текущего отрезка содержатся в переменных $x0$, $x1$. В данном случае функция монотонно возрастает при $x \geq 0$. Инвариантом цикла является утверждение о том, что функция принимает отрицательное или нулевое значение в точке $x0$ и положительное или нулевое значение в точке $x1$. Цикл рано или поздно завершается, поскольку после каждого выполнения тела цикла длина отрезка $[x0, x1]$ уменьшается в два раза.

Приведем полный текст программы:

```
#include <stdio.h> // Описания стандартного ввода-вывода

int main() {
    double a;   // Число, из которого извлекается корень
    double x, x0, x1;      // [x0, x1] - текущий отрезок
    double y;              // Значение ф-ции в точке x
    double eps = 0.000001; // Точность вычисления корня

    printf("Введите число a:\n");
    scanf("%lf", &a);

    if (a < 0.0) {
        printf("Число должно быть неотрицательным.\n");
```

```
        return 1; // Возвращаем код
    }             //     некорректного завершения

    // Задаем концы отрезка
    x0 = 0.0;
    x1 = a;
    if (a < 1.0) {
        x1 = 1.0;
    }

    // Утверждение: x0 * x0 - a <= 0,
    //              x1 * x1 - a >= 0

    while (x1 - x0 > eps) {
        // Инвариант: x0 * x0 - a <= 0,
        //            x1 * x1 - a >= 0
        x = (x0 + x1) / 2.0; // середина отрезка [x0,x1]
        y = x * x - a;       // значение ф-ции в точке x

        if (y >= 0.0) {
            x1 = x; // выбираем левую половину отрезка
        }
        else {
            x0 = x; // выбираем правую половину отрезка
        }
    }

    // Утверждение: x0 * x0 - a <= 0,
    //              x1 * x1 - a >= 0,
    //              x1 - x0 <= eps
    x = (x0 + x1) / 2.0; // Корень := середина отрезка

    // Печатаем ответ
    printf("Квадратный корень = %lf\n", x);

    return 0; // Возвращаем код успешного завершения
}
```

Отметим, что существует более быстрый способ вычисления квадратного корня числа — метод итераций Ньютона, или метод касательных к графику функции, но здесь мы его не рассматриваем.

3.5.7. Выход из цикла break, переход на конец цикла continue

Если необходимо прервать выполнение цикла, следует использовать оператор

```
break;
```

Оператор break применяется внутри тела цикла, заключенного в фигурные скобки. Пример: требуется найти корень целочисленной функции f(x), определенной для целочисленных аргументов.

```
int f(int x);   // Описание прототипа функции
. . .
int x;
. . .
// Ищем корень функции f(x)
x = 0;
while (true) {
    if (f(x) == 0) {
        break;  // Нашли корень
    }
    // Переходим к следующему целому значению x
    //    в порядке 0, -1, 1, -2, 2, -3, 3, ...
    if (x >= 0) {
        x = (-x - 1);
    }
    else {
        x = (-x);
    }
}
// Утверждение: f(x) == 0
```

Здесь используется бесконечный цикл “while (true)”. Выход из цикла осуществляется с помощью оператора “break”.

Иногда требуется пропустить выполнение тела цикла при каких-либо значениях изменяющихся в цикле переменных, переходя к следующему набору значений и очередной итерации. Для этого используется оператор

```
continue;
```

Оператор continue, так же, как и break, используется лишь в том случае, когда тело цикла состоит более чем из одного оператора и заключено в фигурные скобки. Его следует понимать как переход на фигурную скобку, закрывающую тело цикла. Пример: пусть задана $n+1$ точка на вещественной прямой x_i, $i = 0, 1, \ldots, n$; точки x_i будут называться узлами интерполяции. Элементарный *интерполяционный многочлен Лагранжа* $L_k(x)$ — это многочлен степени n, который принимает нулевые значения во всех узлах x_i, кроме x_k. В k-ом узле x_k многочлен $L_k(x)$ принимает значение 1. Многочлен $L_k(x)$ вычисляется по следующей формуле:

$$L_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{(x - x_i)}{(x_k - x_i)}$$

Пусть требуется вычислить значение элементарного интерполяционного многочлена $L_k(x)$ в заданной точке $x = t$. Это делается с помощью следующего фрагмента программы:

```
double x[100]; // Узлы интерполяции (не более 100)
int n;    // Количество узлов интерполяции
int k;    // Номер узла
double t; // Точка, в которой вычисляется значение
double L; // Значение многочлена L_k(x) в точке t
int i;
. . .
L = 1.0;  // Начальное значение произведения
i = 0;
while (i <= n) {
    if (i == k) {
        ++i;      // К следующему узлу
        continue; // Пропустить k-й множитель
    }
```

```
        // Вычисляем произведение
        L *= (t - x[i]) / (x[k] - x[i]);
        ++i; // К следующему узлу
    }
    // Ответ в переменной L
```

Здесь оператор continue используется для того, чтобы пропустить вычисление произведения при $i = k$.

3.5.8. Оператор перехода на метку goto

Оператор перехода goto позволяет изменить естественный порядок выполнения программы и осуществить переход на другой участок программы, обозначенный меткой. Переход может осуществляться только внутри функции, т.е. оператор goto не может ни выйти из функции, ни войти внутрь другой функции. Оператор goto выглядит следующим образом:

```
L: ...;
. . .
goto L;
```

В качестве метки можно использовать любое имя, допустимое в Си (т.е. последовательность букв, цифр и знаков подчеркивания "_", начинающуюся не с цифры). Метка может стоять до или после оператора goto. Метка выделяется символом двоеточия ":". Лучше после него сразу ставить точку с запятой ";", помечая таким образом *пустой оператор* — это общепринятая программистская практика, согласно которой метки ставятся *между операторами*, а не на операторах.

Не следует увлекаться использованием оператора goto — это всегда запутывает программу. Большинство программистов считают применение оператора goto *дурным стилем* программирования. Вместо goto при необходимости можно использовать операторы выхода из цикла break и пропуска итерации цикла continue (см. раздел 3.5.7). Единственная ситуация, в которой использование goto оправдано, — это выход из нескольких вложенных друг в друга циклов:

```
while (...) {
    . . .
    while (...) {
        . . .
        if (...) {
            goto LExit; // Выход из двух
                        // вложенных циклов
        }
        . . .
    }
}
LExit: ;
```

В объектно-ориентированном языке Java, синтаксис которого построен на основе языка Си, использование оператора goto запрещено. Вместо него для выхода из нескольких вложенных друг в друга циклов применяется форма оператора break с меткой. Меткой помечается цикл, из которого надо выйти:

```
Loop1:
while (...) {
    . . .
    while (...) {
        . . .
        break Loop1; // Выход из цикла,
                     // помеченного меткой Loop1
        . . .
    }
    . . .
}
```

3.5.9. Цикл for

Популярный в других языках программирования *арифметический цикл* (см. с. 41) в языке Си реализуется с помощью цикла for. Он выглядит следующим образом:

```
for (инициализация; условие продолжения; итератор)
    тело цикла;
```

Инициализация выполняется один раз перед первой проверкой условия продолжения и первым выполнением тела цикла. *Условие про-*

должения проверяется *перед* каждым выполнением тела цикла. Если условие истинно, то выполняется *тело цикла*, иначе цикл завершается. *Итератор* выполняется *после* каждого выполнения тела цикла (перед следующей проверкой условия продолжения).

Поскольку условие продолжения проверяется перед выполнением тела цикла, цикл for является, подобно циклу while, циклом *с предусловием*. Если условие продолжения не выполняется изначально, то тело цикла не выполняется ни разу, а это хорошо как с точки зрения надежности программы, так и с точки зрения простоты и эстетики (поскольку не нужно отдельно рассматривать исключительные случаи).

Рассмотрим пример суммирования массива с использованием цикла for:

```
double a[100]; // Массив a содержит не более 100 эл-тов
int n;         // Реальная длина массива a (n <= 100)
double sum;    // Переменная для суммы эл-тов массива
int i;         // Переменная цикла
. . .
sum = 0.0;
for (i = 0; i < n; ++i) {
    sum += a[i]; // Увеличиваем сумму на a[i]
}
```

Здесь целочисленная переменная i используется в качестве *переменной цикла*. В операторе инициализации переменной i присваивается значение 0. Условием продолжения цикла является условие $i < n$. Итератор $++i$ увеличивает переменную i на единицу. Таким образом, переменная i последовательно принимает значения 0, 1, 2,..., $n - 1$. Для каждого значения i выполняется тело цикла.

В большинстве других языков программирования арифметический цикл жестко связан с использованием переменной цикла, которая должна принимать значения из арифметической прогрессии. В Си это не так, здесь инициализация, условие продолжения и итератор могут быть произвольными выражениями, что обеспечивает гораздо бóльшую гибкость программы. Конструкцию цикла for можно

реализовать с помощью цикла while:

```
for (
    инициализация;
    условие;
    итератор
) {
    тело цикла;
}
```

$\sim$

```
инициализация;
while (условие) {
    тело цикла;
    итератор;
}
```

Например, фрагмент с суммированием массива реализуется с использованием цикла while следующим образом:

```
for (i=0; i < n; ++i) {
    sum += a[i];
}
```

$\sim$

```
i = 0;
while (i < n) {
    sum += a[i];
    ++i;
}
```

В принципе, конструкция цикла for не нужна: она реализуется с помощью цикла while, он проще и понятнее. Однако большинство программистов продолжают использовать цикл for. Связано это, скорее всего, с традицией и привычками, поскольку в более ранних языках программирования, например, в первых версиях Фортрана, арифметический цикл был основным, а цикл while приходилось реализовывать с помощью операторов if и goto.

3.5.10. Операция «запятая» и цикл for

В цикле for

```
for (инициализация; условие продолжения; итератор)
    тело цикла;
```

в качестве *инициализации* и *итератора* можно использовать любые выражения, в частности, операцию присваивания = и операцию увеличения значения переменной на единицу ++. Как быть, если необходимо выполнить несколько действий при инициализации или в итераторе? Можно, конечно, использовать цикл while, но любители цикла for поступают другим образом. Для этого язык Си предоставляет операцию «запятая», которая позволяет объединить несколько

выражений в одно. У операции «запятая» два аргумента, которые вычисляются последовательно слева направо. Результатом операции является последнее вычисленное, т.е. правое, значение. Пример:

```
int x, y, z;
x = 5;
z = (y = x + 10, ++x);  // y = 15, x = 6, z = 6
```

Здесь при вычислении выражения в скобках сначала вычисляется первое подвыражение $y = x + 10$, в результате которого в y записывается значение 15, значение первого подвыражения также равно 15. Затем вычисляется стоящее после запятой второе подвыражение $++x$, в результате чего значение x увеличивается и становится равным 6, значение второго подвыражения также равно 6. Значением операции «запятая» является значение второго подвыражения, т.е. 6. В результате значение 6 присваивается переменной z.

Наличие операции «запятая» отражает эстетскую сторону первоначального варианта языка Си 70-х годов XX века: в нем почти любая запись имела какой-то смысл. Позже программисты пришли к пониманию того, что надежность программы важнее краткости и изящества, и приняли более строгий ANSI-стандарт языка Си 1989 г., который несколько ограничил свободу творчества в области Си-программ.

Тем не менее, операцию «запятая» по-прежнему можно использовать в заголовке цикла for, когда нужно выполнить несколько действий при инициализации или в итераторе. Например, фрагмент суммирования массива

```
sum = 0.0;
for (i = 0; i < n; ++i) {
    sum += a[i];
}
```

можно переписать следующим «эстетским» образом:

```
for (sum = 0.0, i = 0; i < n; sum += a[i], ++i);
```

Здесь тело цикла вообще пустое, все действия вынесены в заголовок цикла! Лучше избегать такого стиля программирования: он ничего не добавляет в смысле эффективности готовой программы, но делает текст менее понятным и, таким образом, увеличивает вероятность ошибок.

3.5.11. Конструкции, которые лучше не использовать

В программировании предпочтительнее избегать решений эстетически красивых, но не очень понятных. На первый план в последнее время вышло требование надежности программы, поэтому из нескольких решений лучше выбирать более простое, по возможности сводящее к минимуму вероятность ошибок. Это предполагает также некоторое самоограничение свободы программиста.

Перечисленные ниже конструкции существуют в языке Си начиная с самых ранних версий. Тем не менее, без них можно обойтись, заменяя их другими конструкциями, которые потенциально более надежны.

Цикл do... while

Цикл do... while имеет вид

```
do
    действие;
while (условие);
```

Действие лучше всегда обрамлять фигурными скобками, даже когда оно состоит только из одного оператора, например,

```
do {
    x *= 2;
} while (x < n);
```

Цикл do... while является циклом *с постусловием*. Сначала выполняется тело цикла и только после этого проверяется *условие продолжения* цикла. Если условие истинно, то тело цикла повторяется, и так до бесконечности, пока условие не станет ложным. Таким образом, тело цикла выполняется всегда, даже если условие ложно с самого начала. Это является потенциальным источником ошибок. Лучше всегда использовать цикл *с предусловием* while (прежде чем прыгнуть, лучше сначала посмотреть, куда прыгаешь!).

Приведем пример ошибочного использования цикла do... while. Пусть переменная n содержит целое положительное число. Надо записать в целочисленную переменную p максимальную степень двойки, не превосходящую n. Ранее этот фрагмент уже был реализован с помощью цикла while (раздел 3.5.5):

```
int n, p;
. . .
p = 1;
while (2*p <= n) {
    p *= 2;
}
```

Попытка использовать цикл do... while может привести к ошибке:

```
int n, p;
. . .
p = 1;
do {
    p *= 2;
} while (2*p < n);
```

Программа работает неверно при $n = 1$ (в переменную p записывается двойка вместо единицы), поскольку тело цикла do... while всегда выполняется один раз независимо от истинности условия, которое проверяется лишь *после* выполнения тела цикла. Такого рода ошибки в «крайних» ситуациях наиболее опасны в программировании: программа правильно работает почти во всех ситуациях, кроме нескольких исключений. Но известно, что большинство катастроф происходит как раз в результате исключительного стечения обстоятельств!

Оператор switch (вычисляемый goto)

Оператор switch имеет следующий вид:

```
switch (выражение) {
    case значение_1:
        фрагмент_1;
    case значение_2:
        фрагмент_2;
    case значение_3:
        фрагмент_3;
    . . .
    default:        // Необязательный фрагмент
```

```
        фрагмент_N;
}
```

Выражение должно быть дискретного типа (целое число или указатель). *Значения* должны быть *константами* того же типа, что и выражение в заголовке. Оператор switch работает следующим образом:

1) сначала вычисляется значение в*ыражения* в заголовке switch;
2) затем осуществляется переход на метку *"case L:"*, где константа L совпадает с вычисленным значением в*ыражения* в заголовке;
3) если такого значения нет среди меток внутри тела switch, то
   - если есть метка "default:", то осуществляется переход на нее;
   - если метка "default:" отсутствует, то ничего не происходит.

Подчеркнем, что после перехода на метку *case L:* текст программы выполняется последовательно. Например, при выполнении фрагмента программы

```
int n, k;
n = 2;
switch (n) {
    case 1:
        k = 2;
    case 2:
        k = 4;
    case 3:
        k = 8;
}
```

переменной k будет присвоено значение 8, а не 4. Дело в том, что при переходе на метку "case 2:" будут выполнена сначала строка

```
        k = 4;
```

и затем строка

```
        k = 8;
```

что делает приведенный фрагмент совершенно бессмысленным (оптимизирующий компилятор вообще исключит строки “k = 2;” и “k = 4;” из кода готовой программы!). Чтобы исправить этот фрагмент, следует использовать оператор

```
break;
```

Так же, как и в случае цикла, оператор break приводит к выходу из фигурных скобок, обрамляющих тело оператора switch. Приведенный фрагмент надо переписать следующим образом:

```
int n, k;
n = 2;
switch (n) {
    case 1:
        k = 2;
        break;
    case 2:
        k = 4;
        break;
    case 3:
        k = 8;
        break;
}
```

В результате выполнения этого фрагмента переменной k будет присвоено значение 4. Если бы значение n равнялось 1, то k было бы присвоено значение 2, если n равнялось бы 3, то 8. Если n не равно ни 1, ни 2, ни 3, то ничего не происходит.

Оператор switch иногда совершенно необоснованно называют оператором выбора. На самом деле, для выбора следует использовать конструкцию if... else if..., см. раздел 3.5.3. Например, приведенный фрагмент лучше реализовать следующим образом:

```
if (n == 1) {
    k = 2;
}
else if (n == 2) {
    k = 4;
}
```

```
else if (n == 3) {
    k = 8;
}
```

Оператор switch по сути своей является оператором перехода goto с вычисляемой меткой. Ему присущи многие недостатки goto, например, проблемы с инициализацией локальных переменных при входе в блок. Кроме того, switch не позволяет записывать условия в виде логических выражений, что ограничивает сферу его применения. Рекомендуется никогда не использовать оператор switch: выбор в стиле if... else if... во всех отношениях лучше!

3.6. Представление программы в виде функций

Кратко функции уже были рассмотрены в разделе 3.2. Умение разделить большую программу на функции и правильно организовать взаимодействие между функциями является основой искусства программирования. В этом разделе мы обсудим вопросы взаимодействия функций более подробно.

3.6.1. Прототипы функций

Перед использованием или реализацией функции необходимо описать ее *прототип*. Прототип функции сообщает информацию об имени функции, типе возвращаемого значения, количестве и типах ее аргументов. Пример:

```
int gcd(int x, int y);
```

Описан прототип функции gcd, возвращающей целое значение, с двумя целыми аргументами. Имена аргументов x и y здесь являются лишь комментариями, не несущими никакой информации для компилятора. Их можно опускать, например, описание

```
int gcd(int, int);
```

является вполне допустимым.

Описания прототипов функций обычно выносятся в заголовочные файлы, см. раздел 3.1. Для коротких программ, которые помещаются в одном файле, описания прототипов располагают в начале программы. Рассмотрим пример такой короткой программы.

3.6.2. Пример: вычисление наибольшего общего делителя

Программа вводит с клавиатуры терминала два целых числа, затем вычисляет и печатает их наибольший общий делитель. Непосредственно вычисление наибольшего общего делителя реализовано в виде отдельной функции

```
int gcd(int x, int y);
```

(gcd — от слов greatest common divizor). Основная функция main лишь вводит исходные данные, вызывает функцию gcd и печатает ответ. Описание прототипа функции gcd располагается в начале текста программы, затем следует функция main и конце — реализация функции gcd. Приведем полный текст программы:

```
#include <stdio.h> // Описания стандартного ввода-вывода

int gcd(int x, int y); // Описание прототипа функции

int main() {
    int x, y, d;
    printf("Введите два числа:\n");
    scanf("%d%d", &x, &y);
    d = gcd(x, y);
    printf("НОД = %d\n", d);
    return 0;
}

int gcd(int x, int y) { // Реализация функции gcd
    while (y != 0) {
        // Инвариант: НОД(x, y) не меняется
        int r = x % y;  // Заменяем пару (x, y) на
        x = y;          // пару (y, r), где r --
```

```
        y = r;          // остаток от деления x на y
    }
    // Утверждение: y == 0
    return x;   // НОД(x, 0) = x
}
```

Стоит отметить, что реализация функции gcd располагается в конце текста программы. Можно было бы расположить реализацию функции в начале текста и при этом сэкономить на описании прототипа. Это, однако, *дурной стиль*! Лучше всегда, не задумываясь, описывать прототипы всех функций в начале текста, ведь функции могут вызывать друг друга, и правильно упорядочить их (чтобы вызываемая функция была реализована раньше вызывающей) во многих случаях невозможно. К тому же предпочтительнее, чтобы основная функция main, с которой начинается выполнение программы, была бы реализована раньше функций, которые из нее вызываются. Это соответствует *технологии «сверху вниз»* разработки программы: основная задача решается сразу на первом шаге путем сведения ее к одной или нескольким вспомогательным задачам, которые решаются на следующих шагах.

3.6.3. Передача параметров функциям

В языке Си функциям передаются *значения* фактических параметров. При вызове функции значения параметров *копируются* в аппаратный стек, см. раздел 2.3. Следует четко понимать, что изменение формальных параметров в теле функции не приводит к изменению переменных вызывающей программы, передаваемых функции при ее вызове, — ведь функция работает не с самими этими переменными, а с копиями их значений! Рассмотрим, например, следующий фрагмент программы:

```
void f(int x);  // Описание прототипа функции

int main() {
    . . .
    int x = 5;
    f(x);
    // Значение x по-прежнему равно 5
```

```
    . . .
}

void f(int x) {
    . . .
    x = 0;  // Изменение формального параметра
    . . .   // не приводит к изменению фактического
            // параметра в вызывающей программе
}
```

Здесь в функции *main* вызывается функция f, которой передается значение переменной x, равное пяти. Несмотря на то, что в теле функции f формальному параметру x присваивается значение 0, значение переменной x в функции *main* не меняется.

Если необходимо, чтобы функция могла изменить значения переменных вызывающей программы, надо передавать ей указатели на эти переменные. Тогда функция может записать любую информацию по переданным адресам. В Си таким образом реализуются выходные и входно-выходные параметры функций. Подробно этот прием уже рассматривался в разделе 3.5.4, где был дан короткий обзор функций printf и scanf из стандартной библиотеки ввода-вывода языка Си. Напомним, что функции ввода scanf надо передавать адреса вводимых переменных, а не их значения.

3.6.4. Пример: расширенный алгоритм Евклида

Вернемся к примеру с расширенным алгоритмом Евклида, подробно рассмотренному в разделе 1.5.2. Напомним, что наибольший общий делитель двух целых чисел выражается в виде их линейной комбинации с целыми коэффициентами. Пусть x и y — два целых числа, хотя бы одно из которых не равно нулю. Тогда их наибольший общий делитель $d = \text{НОД}(x, y)$ выражается в виде

$$d = ux + vy,$$

где u и v — некоторые целые числа. Алгоритм вычисления чисел d, u, v по заданным x и y называется расширенным алгоритмом Евклида. Мы уже выписывали его на псевдокоде, используя схему построения цикла с помощью инварианта.

Оформим расширенный алгоритм Евклида в виде функции на Си. Назовем ее extGCD (от англ. Extended Greatest Common Divizor). У этой функции два входных аргумента x, y и три выходных аргумента d, u, v. В случае выходных аргументов надо передавать функции указатели на переменные. Итак, функция имеет следующий прототип:

```
void extGCD(int x, int y, int *d, int *u, int *v);
```

При вызове функция вычисляет наибольший общий делитель от двух переданных целых значений x и y и коэффициенты его представления через x и y. Ответ записывается по переданным адресам d, u, v.

Приведем полный текст программы. Функция main вводит исходные данные (числа x и y), вызывает функцию extGCD и печатает ответ. Функция extGCD использует схему построения цикла с помощью инварианта для реализации расширенного алгоритма Евклида.

```
#include <stdio.h> // Описания стандартного ввода-вывода

// Прототип функции extGCD (расш. алгоритм Евклида)
void extGCD(int x, int y, int *d, int *u, int *v);

int main() {
    int x, y, d, u, v;
    printf("Введите два числа:\n");
    scanf("%d%d", &x, &y);
    if (x == 0 && y == 0) {
        printf("Должно быть хотя бы одно ненулевое.\n");
        return 1; // Вернуть код некорректного завершения
    }

    // Вызываем раширенный алгоритм Евклида
    extGCD(x, y, &d, &u, &v);

    // Печатаем ответ
    printf("НОД = %d, u = %d, v = %d\n", d, u, v);

    return 0;   // Вернуть код успешного завершения
```

```
}

void extGCD(int x, int y, int *d, int *u, int *v) {
    int a, b, q, r, u1, v1, u2, v2;
    int t; // вспомогательная переменная

    // инициализация
    a = x; b = y;
    u1 = 1; v1 = 0;
    u2 = 0; v2 = 1;

    // утверждение: НОД(a, b) == НОД(x, y)  &&
    //              a == u1 * x + v1 * y    &&
    //              b == u2 * x + v2 * y;

    while (b != 0) {
        // инвариант: НОД(a, b) == НОД(x, y)  &&
        //            a == u1 * x + v1 * y    &&
        //            b == u2 * x + v2 * y;
        q = a / b; // целая часть частного a / b
        r = a % b; // остаток от деления a на b
        a = b; b = r; // заменяем пару (a, b) на (b, r)

        // Вычисляем новые значения переменных u1, u2
        t = u2;         // запоминаем старое значение u2
        u2 = u1 - q * u2; // вычисляем новое значение u2
        u1 = t;           // новое u1 := старое u2

        // Аналогично вычисляем новые значения v1, v2
        t = v2;
        v2 = v1 - q * v2;
        v1 = t;
    }

    // утверждение: b == 0                 &&
    //              НОД(a, b) == НОД(m, n) &&
    //              a == u1 * m + v1 * n;
```

```
    // Выдаем ответ
    *d = a;
    *u = u1; *v = v1;
}
```

Пример работы программы:

```
Введите два числа:
187 51
НОД = 17, u = -1, v = 4
```

Здесь первая и третья строка напечатаны компьютером, вторая введена человеком.

3.7. Работа с памятью

В традиционных языках программирования, таких как Си, Фортран, Паскаль, существуют три вида памяти: статическая, стековая и динамическая. Конечно, с физической точки зрения никаких различных видов памяти нет: оперативная память — это массив байтов, каждый байт имеет адрес, начиная с нуля. Когда говорится о видах памяти, имеются в виду способы организации работы с ней, включая выделение и освобождение памяти, а также методы доступа.

3.7.1. Статическая память

Статическая память выделяется еще до начала работы программы, на стадии компиляции и сборки. Статические переменные имеют фиксированный адрес, известный до запуска программы и не изменяющийся в процессе ее работы. Статические переменные создаются и инициализируются до входа в функцию *main*, с которой начинается выполнение программы.

Существует два типа статических переменных:

глобальные переменные — это переменные, определенные *вне функций*, в описании которых отсутствует слово *static*. Обычно *описания* глобальных переменных, включающие слово *extern*,

выносятся в заголовочные файлы (h-файлы). Слово extern означает, что переменная описывается, но не создается в данной точке программы. *Определения* глобальных переменных, т.е. описания без слова extern, помещаются в файлы реализации (c-файлы или cpp-файлы). Пример: глобальная переменная *maxind* описывается дважды:

— в h-файле с помощью строки

```
extern int maxind;
```

это описание сообщает о наличии такой переменной, но не создает эту переменную!

— в cpp-файле с помощью строки

```
int maxind = 1000;
```

это описание *создает* переменную *maxind* и присваивает ей начальное значение 1000. Заметим, что стандарт языка не требует обязательного присвоения начальных значений глобальным переменным, но, тем не менее, это лучше делать всегда, иначе в переменной будет содержаться непредсказуемое значение (мусор, как говорят программисты). Инициализация всех глобальных переменных при их определении — это правило хорошего стиля.

Глобальные переменные называются так потому, что они доступны в любой точке программы во всех ее файлах. Поэтому имена глобальных переменных должны быть достаточно длинными, чтобы избежать случайного совпадения имен двух разных переменных. Например, имена x или n для глобальной переменной не подходят;

статические переменные — это переменные, в описании которых присутствует слово *static*. Как правило, статические переменные описываются *вне функций*. Такие статические переменные во всем подобны глобальным, с одним исключением: область видимости статической переменной ограничена одним файлом, внутри которого она определена, — и, более того, ее можно использовать только после ее описания, т.е. ниже по тексту. По

этой причине описания статических переменных обычно выносятся в начало файла. В отличие от глобальных переменных, статические переменные *никогда* не описываются в h-файлах (модификаторы *extern* и *static* конфликтуют между собой).

Совет: используйте статические переменные, если нужно, чтобы они были доступны только для функций, описанных внутри *одного и того же файла*. По возможности не применяйте в таких ситуациях глобальные переменные, это позволит избежать конфликтов имен при реализации больших проектов, состоящих из сотен файлов.

Статическую переменную можно описать и внутри функции, хотя обычно так никто не делает. Переменная размещается не в стеке, а в статической памяти, т.е. ее нельзя использовать при рекурсии, а ее значение сохраняется между различными входами в функцию. Область видимости такой переменной ограничена телом функции, в которой она определена. В остальном она подобна статической или глобальной переменной.

Заметим, что ключевое слово static в языке Си используется для двух различных целей:

— как указание типа памяти: переменная располагается в статической памяти, а не в стеке;

— как способ ограничить область видимости переменной рамками одного файла (в случае описания переменной вне функции).

Слово static может присутствовать и в заголовке функции. При этом оно используется только для того, чтобы ограничить область видимости имени фунции рамками одного файла. Пример:

```
static int gcd(int x, int y);  // Прототип ф-ции
. . .
static int gcd(int x, int y) { // Реализация
    . . .
}
```

Совет: используйте модификатор static в заголовке функции, если известно, что функция будет вызываться лишь внутри одного файла. Слово static должно присутствовать как в описании

прототипа функции, так и в заголовке функции при ее реализации.

3.7.2. Стековая, или локальная, память

Локальные, или стековые, переменные — это переменные, описанные *внутри функции*. Память для таких переменных выделяется в аппаратном стеке, см. раздел 2.3.2. Память выделяется в момент входа в функцию или блок и освобождается в момент выхода из функции или блока. При этом захват и освобождение памяти происходят практически мгновенно, т.к. компьютер только изменяет регистр, содержащий адрес вершины стека.

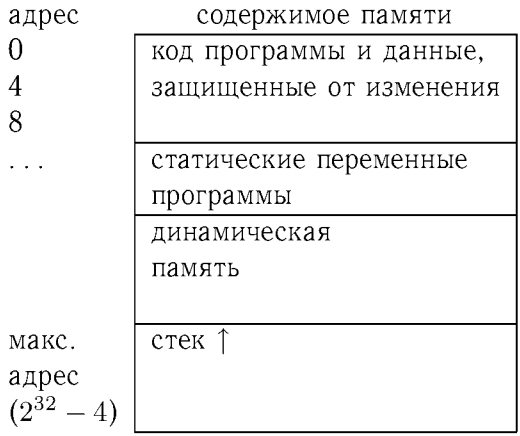
Локальные переменные можно использовать при рекурсии, поскольку при повторном входе в функцию в стеке создается новый набор локальных переменных, а предыдущий набор не разрушается. По этой же причине локальные переменные безопасны при использовании нитей в параллельном программировании (см. раздел 2.6.2). Программисты называют такое свойство функции *реентерабельностью*, от англ. re-enter able — возможность повторного входа. Это очень важное качество с точки зрения надежности и безопасности программы! Программа, работающая со статическими переменными, этим свойством не обладает, поэтому для защиты статических переменных приходится использовать механизмы синхронизации (см. 2.6.2), а логика программы резко усложняется. Всегда следует избегать использования глобальных и статических переменных, если можно обойтись локальными.

Недостатки локальных переменных являются продожением их достоинств. Локальные переменные создаются при входе в функцию и исчезают после выхода из нее, поэтому их нельзя использовать в качестве данных, разделяемых между несколькими функциями. К тому же, размер аппаратного стека не бесконечен, стек может в один прекрасный момент переполниться (например, при глубокой рекурсии), что приведет к катастрофическому завершению программы. Поэтому локальные переменные не должны иметь большого размера. В частности, нельзя использовать большие массивы в качестве локальных переменных.

3.7.3. Динамическая память, или куча

Помимо статической и стековой памяти, существует еще практически неограниченный ресурс памяти, которая называется *динамическая*, или *куча* (heap). Программа может захватывать участки динамической памяти нужного размера. После использования ранее захваченный участок динамической памяти следует освободить.

Под динамическую память отводится пространство виртуальной памяти процесса между статической памятью и стеком. (Механизм виртуальной памяти был рассмотрен в разделе 2.6.) Обычно стек располагается в старших адресах виртуальной памяти и растет в сторону уменьшения адресов (см. раздел 2.3). Программа и константные данные размещаются в младших адресах, выше располагаются статические переменные. Пространство выше статических переменных и ниже стека занимает динамическая память:



Структура динамической памяти автоматически поддерживается исполняющей системой языка Си или C++. Динамическая память состоит из захваченных и свободных сегментов, каждому из которых предшествует описатель сегмента. При выполнении запроса на захват памяти исполняющая система производит поиск свободного сегмента достаточного размера и захватывает в нем отрезок требуемой длины. При освобождении сегмента памяти он помечается как свободный, при необходимости несколько подряд идущих свободных сегментов объединяются.

В языке Си для захвата и освобождения динамической памяти применяются стандартные функции *malloc* и *free*, описания их прототипов содержатся в стандартном заголовочном файле "stdlib.h". (Имя *malloc* является сокращением от memory allocate — "захват памяти".) Прототипы этих функций выглядят следующим образом:

```
void *malloc(size_t n); // Захватить участок памяти
                        // размером в n байт
void free(void *p); // Освободить участок
                    // памяти с адресом p
```

Здесь n — это размер захватываемого участка в байтах, *size_t* — имя одного из целочисленных типов, определяющих максимальный размер захватываемого участка. Тип *size_t* задается в стандартном заголовочном файле "stdlib.h" с помощью оператора *typedef* (см. с. 117). Это обеспечивает независимость текста Си-программы от используемой архитектуры. В 32-разрядной архитектуре тип *size_t* определяется как беззнаковое целое число:

```
typedef unsigned int size_t;
```

Функция *malloc* возвращает адрес захваченного участка памяти или ноль в случае неудачи (когда нет свободного участка достаточно большого размера). Функция *free* освобождает участок памяти с заданным адресом. Для задания адреса используется указатель общего типа `void*`. После вызова функции *malloc* его необходимо привести к указателю на конкретный тип, используя операцию приведения типа, см. раздел 3.4.11. Например, в следующем примере захватывается участок динамической памяти размером в 4000 байтов, его адрес присваивается указателю на массив из 1000 целых чисел:

```
int *a;      // Указатель на массив целых чисел
. . .
a = (int *) malloc(1000 * sizeof(int));
```

Выражение в аргументе функции `malloc` равно 4000, поскольку размер целого числа `sizeof(int)` равен четырем байтам. Для преобразования указателя используется операция приведения типа `(int *)` от указателя обобщенного типа к указателю на целое число.

3.7.4. Пример: печать n первых простых чисел

Рассмотрим пример, использующий захват динамической памяти. Требуется ввести целое число n и напечатать n первых простых чисел. (Простое число — это число, у которого нет нетривиальных делителей.) Используем следующий алгоритм: последовательно проверяем все нечетные числа, начиная с тройки (двойку рассматриваем отдельно). Делим очередное число на все простые числа, найденные на предыдущих шагах алгоритма и не превосходящие квадратного корня из проверяемого числа. Если оно не делится ни на одно из этих простых чисел, то само является простым; оно печатается и добавляется в массив найденных простых.

Поскольку требуемое количество простых чисел n до начала работы программы неизвестно, невозможно создать массив для их хранения в статической памяти. Выход состоит в том, чтобы захватывать пространство под массив в динамической памяти уже после ввода числа n. Вот полный текст программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int n;  // Требуемое количество простых чисел
    int k;  // Текущее количество найденных простых чисел
    int *a; // Указатель на массив найденных простых
    int p;  // Очередное проверяемое число
    int r;  // Целая часть квадратного корня из p
    int i;  // Индекс простого делителя
    bool prime; // Признак простоты

    printf("Введите число простых: ");
    scanf("%d", &n);
    if (n <= 0)   // Некорректное значение =>
        return 1; // завершаем работу с кодом ошибки

    // Захватываем память под массив простых чисел
    a = (int *) malloc(n * sizeof(int));
```

```
a[0] = 2; k = 1;      // Добавляем двойку в массив
printf("%d ", a[0]); // и печатаем ее

p = 3;
while (k < n) {

    // Проверяем число p на простоту
    r = (int)(                // Целая часть корня
        sqrt((double) p) + 0.001
    );
    i = 0;
    prime = true;
    while (i < k && a[i] <= r) {
        if (p % a[i] == 0) { // p делится на a[i]
            prime = false;   // => p не простое,
            break;           // выходим из цикла
        }
        ++i; // К следующему простому делителю
    }

    if (prime) {  // Если нашли простое число,
        a[k] = p; // то добавляем его в массив
        ++k;      // Увеличиваем число простых
        printf("%d ", p); // Печатаем простое число
        if (k % 5 == 0) { // Переход на новую строку
            printf("\n"); // после каждых пяти чисел
        }
    }

    p += 2; // К следующему нечетному числу
}

if (k % 5 != 0) {
    printf("\n"); // Перевести строку
}
```

```
    // Освобождаем динамическую память
    free(a);
    return 0;
}
```

Пример работы данной программы:

```
Введите число простых: 50
2 3 5 7 11
13 17 19 23 29
31 37 41 43 47
53 59 61 67 71
73 79 83 89 97
101 103 107 109 113
127 131 137 139 149
151 157 163 167 173
179 181 191 193 197
199 211 223 227 229
```

3.7.5. Операторы new и delete языка C++

В языке C++ для захвата и освобождения динамической памяти используются операторы *new* и *delete*. Они являются частью языка C++, в отличие от функций *malloc* и *free*, входящих в библиотеку стандартных функций Си.

Пусть T — некоторый тип языка Си или C++, p — указатель на объект типа T. Тогда для захвата памяти размером в один элемент типа T используется оператор *new*:

```
T *p;
p = new T;
```

Например, для захвата восьми байтов под вещественное число типа *double* используется фрагмент

```
double *p;
p = new double;
```

При использовании *new*, в отличие от*malloc*, не нужно приводить указатель от типа `void*` к нужному типу: оператор *new* возвращает указатель на тип, записанный после слова *new*. Сравните два

эквивалентных фрагмента на Си и C++:

```
double *p;                                    | double *p;
p = (double*) malloc(sizeof(double));         | p = new double;
```

Конечно, второй фрагмент гораздо короче и нагляднее.

Оператор *new* удобен еще и тем, что можно присвоить начальное значение объекту, созданному в динамической памяти (т.е. выполнить инициализацию объекта). Для этого начальное значение записывается в круглых скобках после имени типа, следующего за словом *new*. Например, в приведенной ниже строке захватывается память под вещественное число, которому присваивается начальное значение 1.5:

```
double *p = new double(1.5);
```

Этот фрагмент эквивалентен фрагменту

```
double *p = new double;
*p = 1.5;
```

С помощью оператора *new* можно захватывать память под *массив* элементов заданного типа. Для этого в квадратных скобках указывается длина захватываемого массива, которая может представляться любым целочисленным выражением. Например, в следующем фрагменте в динамической создается памяти для хранения вещественной матрицы размера $m \times n$:

```
double *a;
int m = 100, n = 101;
a = new double[m * n];
```

Такую форму оператора *new* иногда называют *векторной*.

Оператор *delete* освобождает память, захваченную ранее с помощью оператора *new*, например,

```
double *p = new double(1.5); // Захват и инициализация
. . .
delete p; // Освобождение памяти
```

Если память под массив была захвачена с помощью *векторной* формы оператора *new*, то для ее освобождения следует использовать *векторную* форму оператора *delete*, в которой после слова *delete* записываются пустые квадратные скобки:

```
double *a = new double[100]; // Захватываем массив
. . .
delete[] a; // Освобождаем массив
```

Для массивов, состоящих из элементов базовых типов Си, при освобождении памяти можно использовать и обычную форму оператора *delete*. Единственное отличие векторной формы: при освобождении массива элементов класса, в котором определен *деструктор*, т.е. завершающее действие перед уничтожением объекта, этот деструктор вызывается для каждого элемента уничтожаемого массива. Поскольку для базовых типов деструкторы не определены, векторная и обычная формы оператора *delete* для них эквивалентны.

Приятная особенность оператора *delete* состоит в том, что при освобождении нулевого указателя ничего не происходит. Например, следующий фрагмент вполне корректен:

```
double *a = 0;  // Нулевой указатель
bool b;
. . .
if (b) {
    a = new double[1000];
    . . .
}
. . .
delete[] a;
```

Здесь в указатель *a* вначале записывается нулевой адрес. Затем, если справедливо некоторое условие, захватывается память под массив. Таким образом, при выполнении оператора *delete* указатель *a* содержит либо нулевое значение, либо адрес массива. В первом случае оператор *delete* ничего не делает, во втором освобождает память, занятую массивом. Такая технология применяется практически всеми программистами на C++: всегда инициализировать указатели на динамическую память нулевыми значениями и в результате не иметь никаких проблем при освобождении памяти.

Попытка освобождения нулевого указателя с помощью стандартной функции *free* может привести к аварийному завершению программы (это зависит от используемой Си-библиотеки: нормальная работа не гарантируется стандартом ANSI).

3.8. Структуры

Структура — это конструкция, которая позволяет объединить несколько переменных с разными типами и именами в один «составной» объект. Она позволяет строить новые типы данных языка Си. В других языках программирования структуры называют записями или кортежами.

Описание структуры выглядит следующим образом:

```
struct имя_структуры {
    описания полей структуры
};
```

Здесь *имя_структуры* — это любое имя, соответствующее синтаксису языка Си, *описания полей структуры* — любая последовательность описаний переменных, имена и типы этих переменных могут быть произвольными. Эти переменные называются *полями структуры*. Заканчивается описание структуры закрывающей фигурной скобкой. За закрывающей фигурной скобкой в описании структуры обязательно следует точка с запятой, в отличие от конструкции составного оператора, не следует забывать об этом! Для чего здесь нужна точка с запятой, будет объяснено ниже в разделе 3.8.3.

Рассмотрим пример: опишем вектор в трехмерном пространстве, который задается тремя вещественными координатами x, y, z:

```
struct R3Vector {
    double x;
    double y;
    double z;
};
```

Таким образом, вводится новый тип "`struct R3Vector`"; объект этого типа содержит внутри себя три вещественных поля с именами x,

y, z. После того как структура определена, можно описывать переменные такого типа, при этом в качестве имени типа следует использовать выражение `struct R3Vector`. Например, в следующей строке описываются два вещественных вектора в трехмерном пространстве с именами u, v:

```
struct R3Vector u, v;
```

С объектами типа структура можно работать как с единым целым, например, копировать эти объекты целиком:

```
struct R3Vector u, v;
. . .
u = v;  // Копируем вектор как единое целое
```

В этом примере вектор v копируется в вектор u; копирование структур сводится к переписыванию области памяти. Сравнивать структуры нельзя:

```
struct R3Vector u, v;
. . .
if (u == v) { // Ошибка! Сравнивать структуры нельзя
    . . .
}
```

Имеется также возможность работать с полями структуры. Для этого используется операция *точка* ".": пусть s — объект типа структура, f — имя поля структуры. Тогда выражение

```
s.f
```

является полем f структуры s, с ним можно работать как с обычной переменной. Например, в следующем фрагменте в вектор w записывается векторное произведение векторов u и v трехмерного пространства: $w = u \times v$.

```
struct R3Vector u, v, w;
. . .
// Вычисляем векторное произведение w = u * v
w.x = u.y * v.z - u.z * v.y;
w.y = (-u.x) * v.z + u.z * v.x;
w.z = u.x * v.y - u.y * v.x;
```

В приведенных примерах все поля структуры *R3Vector* имеют один и тот же тип *double*, однако это совершенно не обязательно. Полями структуры могут быть другие структуры, никаких ограничений нет. Пример: плоскость в трехмерном пространстве задается точкой и вектором нормали, ей соответствует структура *R3Plane*. Точке трехмерного пространства соответствует структура *R3Point*, которая определяется аналогично вектору. Полное описание всех трех структур:

```
struct R3Vector { // Вектор трехмерного пространства
    double x;
    double y;
    double z;
};
struct R3Point { // Точка трехмерного пространства
    double x;
    double y;
    double z;
};
struct R3Plane { // Плоскость в трехмерном пр-ве
    struct R3Point origin;  // точка в плоскости
    struct R3Vector normal; // нормаль к плоскости
};
```

Пусть *plane* — это объект типа плоскость. Для того, чтобы получить координату x точки плоскости, надо два раза применить операцию «точка» доступа к полю структуры:

```
plane.origin.x
```

3.8.1. Структуры и указатели

Указатели на структуры используются довольно часто. Указатель на структуру S описывается обычным образом, в качестве имени типа фигурирует *struct* $S*$. Например, в следующем фрагменте переменная p описана как указатель на структуру S:

```
struct S { . . . }; // Определение структуры S
struct S *p; // Описание указателя на структуру S
```

Описание структуры может содержать указатель на структуру того же типа в качестве одного из полей. Язык Си допускает использование указателей на структуры, определение которых еще не завершено. Например, рассмотрим структуру *TreeNode* (вершина дерева), которая используется при определении бинарного дерева (см. раздел 4.5.4). Она содержит указатели на родительский узел и на левого и правого сыновей, которые также имеют тип *struct TreeNode*:

```
struct TreeNode {   // Вершина дерева
    struct TreeNode *parent; // Указатель на отца,
    struct TreeNode *left;   //   на левого сына,
    struct TreeNode *right;  //   на правого сына
    void *value;             // Значение в вершине
};
```

Здесь при описании полей *parent*, *left*, *right* используется тип "указатель на структуру *TreeNode*", определение которой еще не завершено, что допустимо в языке Си. Возможны и более сложные комбинации, например, структура A содержит указатель на структуру B, а структура B — указатель на структуру A. В этом случае можно использовать *предварительное описание структуры*, например, строка

```
struct A;
```

просто сообщает компилятору, что имя A является именем структуры, полное определение которой будет дано ниже. Тогда упомянутое описание двух структур A и B, ссылающихся друг на друга, может выглядеть следующим образом:

```
struct A; // Предварительное описание структуры A
struct B; // Предварительное описание структуры B

struct A { // Определение структуры A
    . . .
    struct B *p; // Указатель на структуру B
    . . .
};

struct B { // Определение структуры B
```

```
    . . .
    struct A *q; // Указатель на структуру A
    . . .
};
```

Для доступа к полям структуры через указатель на структуру служит операция «стрелочка», которая обозначается двумя символами –> (минус и знак больше), их нужно рассматривать как одну неразрывную лексему (т.е. единый знак, единое слово). Пусть S — имя структуры, f — некоторое поле структуры S, p — указатель на структуру S. Тогда выражение

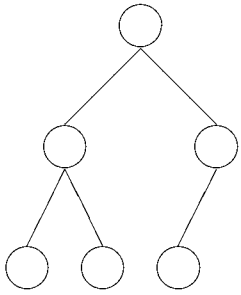
```
p->f
```

обозначает поле f структуры S (само поле, а не указатель не него!). Это выражение можно записать, используя операцию «звездочка» (доступ к объекту через указатель),

```
p->f    ~    (*p).f
```

но, конечно, первый способ гораздо нагляднее. (Во втором случае круглые скобки вокруг выражения `*p` обязательны, поскольку приоритет операции «точка» выше, чем операции «звездочка».)

3.8.2. Пример: рекурсивный обход дерева



В качестве примера использования указателей на структуры приведем фрагмент программы, вычисляющий количество вершин бинарного дерева. Бинарным деревом называется связный граф без циклов, у которого одна вершина отмечена как корневая, а все вершины упорядочены иерархически по длине пути от корня к вершине. У каждой вершины должно быть не больше двух сыновей, причем задан их порядок (левый и правый сыновья).

Вершина дерева описывается структурой *TreeNode*, которая рассматривалась в предыдущем разделе. Если у вершины один из сыновей отсутствует, то соответствующий указатель содержит нулевой адрес.

Для подсчета числа вершин дерева используем функцию *numNodes* с прототипом

```
int numNodes(const struct TreeNode *root);
```

Ей передается константный указатель на корневую вершину дерева или поддерева. Функция возвращает суммарное число вершин дерева или поддерева. Эта функция легко реализуется с помощью рекурсии: достаточно подсчитать число вершин для каждого из двух поддеревьев, соответствующих левому и правому сыновьям корневой вершины, сложить их и прибавить к сумме единицу. Если левый или правый сын отсутствует, то соответствующее слагаемое равно нулю. Вот фрагмент программы, реализующий функцию *numNodes*.

```
// Описание структуры, представляющей вершину дерева
struct TreeNode {
    struct TreeNode *parent; // Указатель на отца,
    struct TreeNode *left;   //   на левого сына,
    struct TreeNode *right;  //   на правого сына
    void *value;             // Значение в вершине
};

// Рекурсивная реализация функции,
// вычисляющей число вершин дерева.
// Вход: указатель на корень поддерева
// Возвращаемое значение: число вершин поддерева
int numNodes(const struct TreeNode *root) {
    int num = 0;
    if (root == 0) { // Для нулевого указателя на корень
        return 0;    // возвращаем ноль
    }

    if (root->left != 0) {           // Есть левый сын =>
        num += numNodes(root->left); // вызываем функцию
    }                                // для левого сына

    if (root->right != 0) {          // Есть правый сын =>
        num += numNodes(root->right); // вызываем ф-цию
    }                                  // для правого сына
```

```
    return num + 1; // Возвращаем суммарное число вершин
}
```

Здесь неоднократно применялась операциея «стрелочка» `->` для доступа к полю структуры через указатель на нее.

3.8.3. Структуры и оператор определения типа typedef

Синтаксис языка Си позволяет в одном предложении определить структуру и описать несколько переменных структурного типа. Например, строка

```
struct R2_point { double x; double y; } t, *p;
```

одновременно определяет структуру `R2_point` (точка на двумерной плоскости) и описывает две переменные t и p. Первая имеет тип `struct R2_point` (точка плоскости), вторая — `struct R2_point *` (указатель на точку плоскости). Таким образом, после закрывающей фигурной скобки может идти необязательный список определяемых переменных, причем можно использовать все конструкции Си для построения сложных типов (указатели, массивы, функции). Список всегда завершается точкой с запятой, поэтому даже при пустом списке точка с запятой после фигурной скобки обязательна.

Возможно *анонимное определение структуры*, когда имя структуры после ключевого слова *struct* опускается; в этом случае список описываемых переменных должен быть непустым (иначе такое описание совершенно бессмысленно). Пример:

```
struct { double x; double y; } t, *p;
```

Здесь имя структуры отсутствует. Определены две переменные t и p, первая имеет структурный тип с полями x и y типа *double*, вторая — указатель на данный структурный тип. Такие описания в чистом виде программисты обычно не используют, гораздо чаще анонимное определение структуры комбинируют с оператором определения имени типа *typedef* (см. с. 117). Например, можно определить два типа *R2Point* (точка вещественной двумерной плоскости) и *R2PointPtr* (указатель на точку вещественной двумерной плоскости) в одном предложении, комбинируя оператор *typedef* с анонимным определением структуры:

```
typedef struct {
    double x;
    double y;
} R2Point, *R2PointPtr;
```

Такая технология довольно популярна среди программистов и применяется в большинстве системных h-файлов. Преимущество ее состоит в том, что в дальшейшем при описании переменных структурного типа не нужно использовать ключевое слово *struct*, например,

```
R2Point a, b, c;     // Описываем три точки a, b, c
R2PointPtr p;        // Описываем указатель на точку
R2Point *q;          // Эквивалентно R2PointPtr q;
```

Сравните с описаниями, использующими приведенное выше определение структуры *R2_point*:

```
struct R2_Point a, b, c;
struct R2_Point *p;
struct R2_Point *q;
```

Первый способ лаконичнее и нагляднее.

Вовсе не обязательно комбинировать оператор *typedef* непременно с анонимным определением структуры; можно в одном предложении как определить имя структуры, так и ввести новый тип. Например, предложение

```
typedef struct R2_point {
    double x;
    double y;
} R2Point, *R2PointPtr;
```

определяет структуру R2_point, а также два новых типа R2Point (структура R2_point) и R2PointPtr (указатель на структуру R2_point). К сожалению, имя структуры не должно совпадать с именем типа, именно поэтому здесь в качестве имени структуры приходится использовать несколько вычурное имя R2_point. Впрочем, обычно в дальнейшем оно не нужно.

> Все вышесказанное касательно языка Си справедливо и в C++. Кроме того, в C++ считается, что определение структуры S одновременно вводит и новый тип с именем S. Поэтому в случае C++ нет

необходимости в использовании оператора typedef при задании структурных типов. Связано это с тем, что структура с точки зрения C++ является классом, а классы и определяемые ими типы — это основа языка C++. Сравните описания Си и C++:

```
struct S { ... };
struct S a, b, c;
struct S *p, *q;
```

```
struct S { ... };
S a, b, c;
S *p, *q;
```

Конечно, описания C++ проще и нагляднее.

3.9. Технология программирования на Си

В этом разделе будут рассмотрены некоторые приемы программирования на Си (например, реализация матриц и многомерных массивов), а также работа с текстами и файлами при помощи функций стандартной Си-библиотеки.

3.9.1. Представление матриц и многомерных массивов

Специального типа данных «матрица» или «многомерный массив» в Си нет, однако, можно использовать массив элементов типа массив. Например, переменная *a* представляет матрицу размера 3×3 с вещественными элементами:

```
double a[3][3];
```

Элементы матрицы располагаются в памяти последовательно по строкам: сначала идут элементы строки с индексом 0, затем строки с индексом 1, в конце строки с индексом 2 (в программировании отсчет индексов всегда начинается с нуля, а не с единицы!). При этом выражение

```
a[i]
```

где i — целая переменная, представляет собой *указатель* на начальный элемент i-й строки и имеет тип `double*`.

Для обращения к элементу матрицы надо записать его индексы в квадратных скобках, например, выражение

```
a[i][j]
```

представляет собой элемент матрицы a в строке с индексом i и столбце с индексом j. Элемент матрицы можно использовать в любом выражении как обычную переменную (например, можно читать его значение или присваивать новое).

Такая реализация матрицы удобна и максимально эффективна с точки зрения времени доступа к элементам. У нее только один существенный недостаток: так можно реализовать только матрицу, размер которой известен заранее. Язык Си не позволяет описывать массивы переменного размера, размер массива должен быть известен до начала работы программы еще на стадии компиляции.

Пусть нужна матрица, размер которой определяется во время работы программы. Тогда пространство под нее надо захватывать в динамической памяти с помощью функции *malloc* языка Си или оператора *new* языка C++ (см. раздел 3.7.3). При этом в динамической памяти захватывается линейный массив и возвращается указатель на него. Рассмотрим вещественную матрицу размером m строк на n столбцов. Захват памяти выполняется с помощью функции *malloc* языка Си

```
double *a;
int m, n;
. . .
a = (double *) malloc(m * n * sizeof(double));
```

или с помощью оператора *new* языка C++:

```
double *a;
int m, n;
. . .
a = new double[m * n];
```

При этом считается, что элементы матрицы будут располагаться в массиве следующим образом: сначала идут элементы строки с индексом 0, затем элементы строки с индексом 1 и т.д., последними идут элементы строки с индексом $m - 1$. Каждая строка состоит из n элементов, следовательно, индекс элемента строки i и столбца j в линейном массиве равен

$$i \cdot n + j$$

(действительно, поскольку индексы начинаются с нуля, то i равно количеству строк, которые нужно пропустить, $i \cdot n$ — суммарное количество элементов в пропускаемых строках; число j равно смещению внутри последней строки). Таким образом, элементу матрицы в строке i и столбце j соответствует выражение

```
a[i * n + j]
```

Этот способ представления матрицы удобен и эффективен. Его основное преимущество состоит в том, что элементы матрицы хранятся в *непрерывном* отрезке памяти. Во-первых, это позволяет оптимизирующему компилятору преобразовывать текст программы, добиваясь максимального быстродействия; во-вторых, при выполнении программы максимально используется механизм кеш-памяти, сводящий к минимуму обращения к памяти и значительно ускоряющий работу программы.

В некоторых книгах по Си рекомендуется реализовывать матрицу как массив указателей на ее строки, при этом память под каждую строку захватывается отдельно в динамической памяти:

```
double **a; // Адрес массива указателей
int m, n;   // Размеры матрицы: m строк, n столбцов
int i;
. . .
// Захватывается память под массив указателей
a = (double **) malloc(m * sizeof(double *));

for (i = 0; i < m; ++i) {
    // Захватывается память под строку с индесом i
    a[i] = (double *) malloc(n * sizeof(double));
}
```

После этого к элементу a_{ij} можно обращаться с помощью выражения

```
a[i][j]
```

Несмотря на всю сложность этого решения, никакого выигрыша нет, наоборот, программа проигрывает в скорости! Причина состоит в том, что матрица не хранится в непрерывном участке памяти, это мешает как оптимизации программы, так и эффективному использованию кеш-памяти. Так что лучше не применять такой метод представления матрицы.

Многомерные массивы реализуются аналогично матрицам. Например, вещественный трехмерный массив размера $4 \times 4 \times 2$ описывается как

```
double a[4][4][2];
```

обращение к его элементу с индексами x, y, z осуществляется с помощью выражения

```
a[x][y][z]
```

Многомерные массивы переменного размера с числом индексов большим двух встречаются в программах довольно редко, но никаких проблем с их реализацией нет: они реализуются аналогично матрицам. Например, пусть надо реализовать трехмерный вещественный массив размера $m \times n \times k$. Захватывается линейный массив вещественных чисел размером $m \cdot n \cdot k$:

```
double *a;
. . .
a = (double *) malloc(m * n * k * sizeof(double));
```

Доступ к элементу с индексами x, y, z осуществляется с помощью выражения

```
a[(x * n + y) * k + z]
```

3.9.2. Пример: приведение матрицы к ступенчатому виду методом Гаусса

В качестве примера работы с матрицами рассмотрим алгоритм Гаусса приведения матрицы к ступенчатому виду. Метод Гаусса — один из основных результатов линейной алгебры и аналитической геометрии, к нему сводятся множество других теорем и методов линейной алгебры (теория и вычисление определителей, решение систем линейных уравнений, вычисление ранга матрицы и обратной матрицы, теория базисов конечномерных векторных пространств и т.д.).

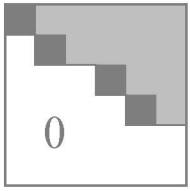
Напомним, что матрица A с элементами a_{ij} называется ступенчатой, если она обладает следующими двумя свойствами:

1) если в матрице есть нулевая строка, то все строки ниже нее также нулевые;

2) пусть $a_{ij} \neq 0$ — первый ненулевой элемент в строке с индексом i, т.е. элементы $a_{il} = 0$ при $l < j$. Тогда все элементы в j-м столбце ниже элемента a_{ij} равны нулю, и все элементы левее и ниже a_{ij} также равны нулю:

$$a_{kl} = 0 \quad \text{при} \quad k > i \quad \text{и} \quad l \leq j.$$

Ступенчатая матрица выглядит примерно так:



здесь темными квадратиками отмечены первые ненулевые элементы строк матрицы. Белым цветом изображаются нулевые элементы, серым цветом — произвольные элементы.

Алгоритм Гаусса использует элементарные преобразования матрицы двух типов.

Преобразование первого рода: две строки матрицы меняются местами, и при этом знаки всех элементов одной из строк изменяются на противоположные.

Преобразование второго рода: к одной строке матрицы прибавляется другая строка, умноженная на произвольное число.

Элементарные преобразования сохраняют определитель и ранг матрицы, а также множество решений линейной системы. Алгоритм Гаусса приводит произвольную матрицу элементарными преобразованиями к ступенчатому виду. Для ступенчатой квадратной матрицы определитель равен произведению диагональных элементов, а ранг — числу ненулевых строк (рангом по определению называется размерность линейной оболочки строк матрицы).

Метод Гаусса в «математическом» варианте состоит в следующем:

1) ищем сначала *ненулевой* элемент в первом столбце. Если все элементы первого столбца нулевые, то переходим ко второму столбцу, и так далее. Если нашли ненулевой элемент в k-й строке, то при помощи элементарного преобразования первого

рода меняем местами первую и k-ю строки, добиваясь того, чтобы первый элемент первой строки был отличен от нуля;

2) используя элементарные преобразования второго рода, обнуляем все элементы первого столбца, начиная со второго элемента. Для этого от строки с номером k вычитаем первую строку, умноженную на коэффициент a_{k1}/a_{11}

3) переходим ко второму столбцу (или j-му, если все элементы первого столбца были нулевыми), и в дальнейшем рассматриваем только часть матрицы, начиная со второй строки и ниже. Снова повторяем пункты 1) и 2) до тех пор, пока не приведем матрицу к ступенчатому виду.

«Программистский» вариант метода Гаусса имеет три отличия от «математического»:

1) индексы строк и столбцов матрицы начинаются с нуля, а не с единицы;

2) недостаточно найти просто *ненулевой* элемент в столбце. В программировании все действия с вещественными числами производятся приближенно, поэтому можно считать, что точного равенства вещественных чисел вообще не бывает. Некоторые компиляторы даже выдают предупреждения на каждую операцию проверки равенства вещественных чисел. Поэтому вместо проверки на равенство нулю числа a_{ij} следует сравнивать его абсолютную величину $|a_{ij}|$ с очень маленьким числом ε (например, $\varepsilon = 0.0000001$). Если $|a_{ij}| \leq \varepsilon$, то следует считать элемент a_{ij} нулевым;

3) при обнулении элементов j-го столбца, начиная со строки $i+1$, мы к k-й строке, где $k > i$, прибавляем i-ю строку, умноженную на коэффициент $r = -a_{kj}/a_{ij}$:

$$r = -a_{kj}/a_{ij}$$
$$\overrightarrow{a_k} = \overrightarrow{a_k} + r \cdot \overrightarrow{a_i}$$

Такая схема работает нормально только тогда, когда коэффициент r по абсолютной величине не превосходит единицы. В

противном случае, ошибки округления умножаются на большой коэффициент и, таким образом, экспоненциально растут. Математики называют это явление *неустойчивостью* вычислительной схемы. Если вычислительная схема неустойчива, то полученные с ее помощью результаты не имеют никакого отношения к исходной задаче. В нашем случае схема устойчива, когда коэффициент $r = -a_{kj}/a_{ij}$ не превосходит по модулю единицы. Для этого должно выполняться неравенство

$$|a_{ij}| \geq |a_{kj}| \quad \text{при} \quad k > i.$$

Отсюда следует, что при поиске разрешающего элемента в j-м столбце необходимо найти не первый попавшийся ненулевой элемент, а *максимальный по абсолютной величине*. Если он по модулю не превосходит ε, то считаем, что все элементы столбца нулевые; иначе меняем местами строки, ставя его на вершину столбца, и затем обнуляем столбец элементарными преобразованиями второго рода.

Ниже дан полный текст программы на Си, приводящей вещественную матрицу к ступенчатому виду. Функция, реализующая метод Гаусса, одновременно подсчитывает и ранг матрицы. Программа вводит размеры матрицы и ее элементы с клавиатуры и вызывает функцию приведения к ступенчатому виду. Затем программа печатает ступенчатый вид матрицы и ее ранг. В случае квадратной матрицы также вычисляется и печатается определитель матрицы, равный произведению диагональных элементов ступенчатой матрицы.

При реализации метода Гаусса используется схема построения цикла с помощью инварианта, см. раздел 1.5.2. В цикле меняются две переменные — индекс строки i, $0 \leq i < m-1$, и индекс столбца j, $0 \leq j < n-1$. Инвариантом цикла является утверждение о том, что часть матрицы (математики говорят *минор*) в столбцах $0, 1, \ldots j-1$ приведена к ступенчатому виду и что первый ненулевой элемент в строке $i-1$ стоит в столбце с индексом меньшим j. В теле цикла рассматривается только минор матрицы в строках $i, \ldots, m-1$ и столбцах $j, \ldots, n-1$. Сначала ищется максимальный по модулю элемент в j-м столбце. Если он по абсолютной величине не превосходит ε, то j увеличивается на единицу (считается, что столбец нулевой). Иначе перестановкой строк разрешающий элемент

ставится на вершину j-го столбца минора, и затем столбец обнуляется элементарными преобразованиями второго рода. После этого оба индекса i и j увеличиваются на единицу. Алгоритм завершается, когда либо $i = m$, либо $j = n$. По окончании алгоритма значение переменной i равно числу ненулевых строк ступенчатой матрицы, т.е. рангу исходной матрицы.

Для вычисления абсолютной величины вещественного числа x типа `double` мы пользуемся стандарной математической функцией `fabs(x)`, описанной в стандартном заголовочном файле "math.h".

```
#include <stdio.h>  // Описания функций ввода-вывода
#include <math.h>   // Описания математических функций
#include <stdlib.h> // Описания функций malloc и free

// Прототип функции приведения матрицы
// к ступенчатому виду.
// Функция возвращает ранг матрицы
int gaussMethod(
    int m,          // Число строк матрицы
    int n,          // Число столбцов матрицы
    double *a,      // Адрес массива элементов матрицы
    double eps      // Точность вычислений
);

int main() {
    int m, n, i, j, rank;
    double *a;
    double eps, det;

    printf("Введите размеры матрицы m, n: ");
    scanf("%d%d", &m, &n);

    // Захватываем память под элементы матрицы
    a = (double *) malloc(m * n * sizeof(double));

    printf("Введите элементы матрицы:\n");
    for (i = 0; i < m; ++i) {
        for (j = 0; j < n; ++j) {
```

```
            // Вводим элемент с индексами i, j
            scanf("%lf", &(a[i*n + j]));
        }
    }

    printf("Введите точность вычислений eps: ");
    scanf("%lf", &eps);

    // Вызываем метод Гаусса
    rank = gaussMethod(m, n, a, eps);

    // Печатаем ступенчатую матрицу
    printf("Ступенчатый вид матрицы:\n");
    for (i = 0; i < m; ++i) {
        // Печатаем i-ю строку матрицы
        for (j = 0; j < n; ++j) {
            printf(         // Формат %10.3lf означает 10
                "%10.3lf ", // позиций на печать числа,
                a[i*n + j]  // 3 знака после точки
            );
        }
        printf("\n");   // Перевести строку
    }

    // Печатаем ранг матрицы
    printf("Ранг матрицы = %d\n", rank);

    if (m == n) {
        // Для квадратной матрицы вычисляем и печатаем
        //     ее определитель
        det = 1.0;
        for (i = 0; i < m; ++i) {
            det *= a[i*n + i];
        }
        printf("Определитель матрицы = %.3lf\n", det);
    }
```

```
    free(a);    // Освобождаем память
    return 0;   // Успешное завершение программы
}

// Приведение вещественной матрицы
// к ступенчатому виду методом Гаусса с выбором
// максимального разрешающего элемента в столбце.
// Функция возвращает ранг матрицы
int gaussMethod(
    int m,          // Число строк матрицы
    int n,          // Число столбцов матрицы
    double *a,      // Адрес массива элементов матрицы
    double eps      // Точность вычислений
) {
    int i, j, k, l;
    double r;

    i = 0; j = 0;
    while (i < m && j < n) {
        // Инвариант: минор матрицы в столбцах 0..j-1
        //   уже приведен к ступенчатому виду, и строка
        //   с индексом i-1 содержит ненулевой эл-т
        //   в столбце с номером, меньшим чем j

        // Ищем максимальный элемент в j-м столбце,
        // начиная с i-й строки
        r = 0.0;
        for (k = i; k < m; ++k) {
            if (fabs(a[k*n + j]) > r) {
                l = k;      // Запомним номер строки
                r = fabs(a[k*n + j]); // и макс. эл-т
            }
        }
        if (r <= eps) {
            // Все элементы j-го столбца по абсолютной
            // величине не превосходят eps.
            // Обнулим столбец, начиная с i-й строки
```

```
            for (k = i; k < m; ++k) {
                a[k*n + j] = 0.0;
            }
            ++j;       // Увеличим индекс столбца
            continue; // Переходим к следующей итерации
        }

        if (l != i) {
            // Меняем местами i-ю и l-ю строки
            for (k = j; k < n; ++k) {
                r = a[i*n + k];
                a[i*n + k] = a[l*n + k];
                a[l*n + k] = (-r); // Меняем знак строки
            }
        }

        // Утверждение: fabs(a[i*n + k]) > eps

        // Обнуляем j-й столбец, начиная со строки i+1,
        // применяя элем. преобразования второго рода
        for (k = i+1; k < m; ++k) {
            r = (-a[k*n + j] / a[i*n + j]);

            // К k-й строке прибавляем i-ю, умноженную на r
            a[k*n + j] = 0.0;
            for (l = j+1; l < n; ++l) {
                a[k*n + l] += r * a[i*n + l];
            }
        }

        ++i; ++j;   // Переходим к следующему минору
    }

    return i; // Возвращаем число ненулевых строк
}
```

Приведем два примера работы этой программы. В первом случае вводится вырожденная матрица размера 4×4:

```
Введите размеры матрицы m, n: 4 4
Введите элементы матрицы:
1 2 3 4
4 3 2 1
5 6 7 8
8 7 6 5
Введите точность вычислений eps: 0.00001
Ступенчатый вид матрицы:
     8.000      7.000      6.000      5.000
     0.000      1.625      3.250      4.875
     0.000      0.000      0.000      0.000
     0.000      0.000      0.000      0.000
Ранг матрицы = 2
Определитель матрицы = 0.000
```

Во втором случае вводится матрица размера 3×4 максимального ранга:

```
Введите размеры матрицы m, n: 3 4
Введите элементы матрицы:
1 0 2 1
2 1 0 -1
1 0 1 0
Введите точность вычислений eps: 0.00001
Ступенчатый вид матрицы:
     2.000      1.000      0.000     -1.000
     0.000      0.500     -2.000     -1.500
     0.000      0.000     -1.000     -1.000
Ранг матрицы = 3
```

3.9.3. Работа с файлами

Стандартная библиотека Си содержит набор функций для работы с файлами. Эти функции описаны в стандарте ANSI. Отметим, что файловый ввод-вывод не является частью языка Си, и ANSI-функции — не единственное средство ввода-вывода. Так, в операционной системе Unix более популярен другой набор функций ввода-вывода, который можно использовать не только для работы

с файлами, но и для обмена по сети. В C++ часто используются библиотеки классов для ввода-вывода. Тем не менее, функции ANSI-библиотеки поддерживаются всеми Си-компиляторами, и потому программы, применяющие их, легко переносятся с одной платформы на другую. Прототипы функций ввода-вывода и используемые для этого типы данных описаны в стандартном заголовочном файле "stdio.h".

Открытие файла: функция fopen

Для доступа к файлу применяется тип данных *FILE*. Это структурный тип, имя которого задано с помощью оператора *typedef* в стандартном заголовочном файле "stdio.h". Программисту не нужно знать, как устроена структура типа файл: ее устройство может быть системно зависимым, поэтому в целях переносимости программ обращаться явно к полям струтуры *FILE* запрещено. Тип данных "указатель на структуру *FILE*" используется в программах как черный ящик: функция открытия файла возвращает этот указатель в случае успеха, и в дальнейшем все файловые функции применяют его для доступа к файлу.

Прототип функции открытия файла выглядит следующим образом:

```
FILE *fopen(const char *path, const char *mode);
```

Здесь *path* — путь к файлу (например, имя файла или абсолютный путь к файлу), *mode* — режим открытия файла. Строка *mode* может содержать несколько букв. Буква "r" (от слова read) означает, что файл открывается для чтения (файл должен существовать). Буква "w" (от слова write) означает запись в файл, при этом старое содержимое файла теряется, а в случае отсутствия файла он создается. Буква "a" (от слова append) означает запись в конец существующего файла или создание нового файла, если файл не существует.

В некоторых операционных системах имеются различия в работе с текстовыми и бинарными файлами (к таким системам относятся MS DOS и MS Windows; в системе Unix различий между текстовыми и бинарными файлами нет). В таких системах при открытии бинарного файла к строке *mode* следует добавлять букву "b" (от слова binary), а при открытии текстового файла — букву "t" (от слова

text). Кроме того, при открытии можно разрешить выполнять как операции чтения, так и записи; для этого используется символ + (плюс). Порядок букв в строке *mode* следующий: сначала идет одна из букв "r", "w", "a", затем в произвольном порядке могут идти символы "b", "t", "+". Буквы "b" и "t" можно использовать, даже если в операционной системе нет различий между бинарными и текстовыми файлами, в этом случае они просто игнорируются.

Значения символов в строке *mode* сведены в следующую таблицу:

r	Открыть существующий файл на чтение.
w	Открыть файл на запись. Старое содержимое существующего файла теряется, в случае отсутствия файла он создается.
a	Открыть файл на запись. Если файл существует, то запись производится в его конец.
t	Открыть текстовый файл
b	Открыть бинарный файл
+	Разрешить и чтение, и запись

Несколько примеров открытия файлов:

```
FILE *f, *g, *h;
. . .
// 1. Открыть текстовый файл "abcd.txt" для чтения
f = fopen("abcd.txt", "rt");

// 2. Открыть бинарный файл "c:\Windows\Temp\tmp.dat"
// для чтения и записи
g = fopen("c:/Windows/Temp/tmp.dat", "wb+");

// 3. Открыть текстовый файл "c:\Windows\Temp\abcd.log"
// для дописывания в конец файла
h = fopen("c:\\Windows\\Temp\\abcd.log", "at");
```

Обратите внимание, что во втором случае мы используем обычную косую черту / для разделения директорий, хотя в системах MS DOS и MS Windows для этого принято использовать обратную косую черту \. Дело в том, что в операционной системе Unix и в языке Си,

который является для нее «родным», символ \ используется в качестве «экранирующего» символа, т.е. для защиты следующего за ним символа от интерпретации как специального. Поэтому во всех строковых константах Си обратную косую черту надо повторять дважды, как это и сделано в третьем примере. Впрочем, стандартная библиотека Си позволяет в именах файлов использовать нормальную косую черту вместо обратной; эта возможность была использована во втором примере.

В случае удачи функция *fopen* открытия файла возвращает ненулевой указатель на структуру типа *FILE*, описывающую параметры открытого файла. Этот указатель надо затем использовать во всех файловых операциях. В случае неудачи (например, при попытке открыть на чтение несуществующий файл) возвращается ненулевой указатель. При этом глобальная системная переменная *errno*, описанная в стандартном заголовочном файле "errno.h", содержит численный код ошибки. В случае неудачи при открытии файла этот код можно распечатать, чтобы получить дополнительную информацию:

```
#include <stdio.h>
#include <errno.h>
. . .

    FILE *f = fopen("filnam.txt", "rt");
    if (f == NULL) {
        printf(
            "Ошибка открытия файла с кодом %d\n",
            errno
        );
        . . .
    }
```

Константа NULL

В приведенном выше примере при открытии файла функция *fopen* в случае ошибки возвращает нулевой указатель на структуру *FILE*. Чтобы проверить, произошла ли ошибка, следует сравнить возвращенное значение с нулевым указателем. Для наглядности стандартный заголовочный файл "stdio.h" определяет символическую кон-

станту *NULL* как нулевой указатель на тип *void*:

```
#define NULL ((void *) 0)
```

Сделано это вроде бы с благой целью: чтобы отличить число ноль от нулевого указателя. При этом язык Си, в котором контроль ошибок осуществляется недостаточно строго, позволяет сравнивать указатель общего типа `void *` с любым другим указателем. Между тем,в Си вместо константы NULL всегда можно использовать просто 0, и вряд ли от этого программа становится менее понятной. Более строгий язык C++ запрещает сравнение разных указателей, поэтому в случае C++ стандартный заголовочный файл определяет константу NULL как обычный ноль:

```
#define NULL 0
```

Автор языка C++ Б. Страуструп советует использовать обычный ноль 0 вместо символического обозначения NULL. Тем не менее, по традиции большинство программистов любят константу NULL.

Константа NULL не является частью языка Си или C++, и без подключения одного из стандартных заголовочных файлов, в котором она определяется, использовать ее нельзя. (По этой причине авторы языка Java добавили в язык ключевое слово null, записываемое строчными буквами.) Так что в случае Си или C++ безопаснее следовать совету Б. Страуструпа и использовать обычный ноль 0 вместо символической константы NULL.

Диагностика ошибок: функция perror

Использовать переменную *errno* для печати кода ошибки не очень удобно, поскольку необходимо иметь под рукой таблицу возможных кодов ошибок и их значений. В стандартной библиотеке Си существует более удобная функция *perror*, которая печатает системное сообщение о последней ошибке вместо ее кода. Печать производится на английском языке, но есть возможность добавить к системному сообщению любой текст, который указывается в качестве единственного аргумента функции *perror*. Например, предыдущий фрагмент переписывается следующим образом:

```
#include <stdio.h>
. . .

    FILE *f = fopen("filnam.txt", "rt");
    if (f == 0) {
        perror("Не могу открыть файл на чтение");
        . . .
    }
```

Функция *perror* печатает сначала пользовательское сообщение об ошибке, затем после двоеточия системное сообщение. Например, при выполнении приведенного фрагмента в случае ошибки из-за отсутствия файла будет напечатано

```
Не могу открыть файл на чтение: No such file or directory
```

Функции бинарного чтения и записи fread и fwrite

После того как файл открыт, можно читать информацию из файла или записывать информацию в файл. Рассмотрим сначала функции *бинарного* чтения и записи *fread* и *fwrite*. Они называются бинарными потому, что не выполняют никакого преобразования информации при вводе или выводе (с одним небольшим исключением при работе с текстовыми файлами, которое будет рассмотрено ниже): информация хранится в файле как последовательность байтов ровно в том виде, в котором она хранится в памяти компьютера.

Функции чтения *fread* имеет следующий прототип:

```
size_t fread(
    char *buffer,     // Массив для чтения данных
    size_t elemSize, // Размер одного элемента
    size_t numElems, // Число элементов для чтения
    FILE *f          // Указатель на структуру FILE
);
```

Здесь *size_t* определен как беззнаковый целый тип в системных заголовочных файлах. Функция пытается прочесть *numElems* элементов из файла, который задается указателем *f* на структуру FILE, размер каждого элемента равен *elemSize*. Функция возвращает реальное число прочитанных элементов, которое может быть меньше,

чем *numElems*, в случае конца файла или ошибки чтения. Указатель *f* должен быть возвращен функцией *fopen* в результате успешного открытия файла. Пример использования функции *fread*:

```
FILE *f;
double buff[100];
size_t res;

f = fopen("tmp.dat", "rb"); // Открываем файл
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для чтения");
    exit(1);  // завершить работу с кодом 1
}

// Пытаемся прочесть 100 вещественных чисел из файла
res = fread(buff, sizeof(double), 100, f);
// res равно реальному количеству прочитанных чисел
```

В этом примере файл "tmp.dat" открывается на чтение как бинарный, из него читается 100 вещественных чисел размером 8 байт каждое. Функция *fread* возвращает реальное количество прочитанных чисел, которое меньше или равно, чем 100.

Функция *fread* читает информацию в виде потока байтов и в неизменном виде помещает ее в память. Следует различать текстовое представление чисел и их бинарное представление! Таким образом, в приведенном выше примере числа в файле должны быть записаны в *бинарном виде*, а не в виде текста. Для текстового ввода чисел надо использовать функции *ввода по формату*, которые будут рассмотрены ниже.

> Внимание! Открытие файла как текстового с помощью функции *fopen*, например,
>
> ```
> FILE *f = fopen("tmp.dat", "rt");
> ```
>
> вовсе не означает, что числа при вводе с помощью функции *fopen* будут преобразовываться из текстовой формы в бинарную! Из этого следует только то, что в операционных системах, в которых строки текстовых файлов разделяются парами символами "`\r\n`" (они имеют названия CR и LF — «возврат каретки» и «продергивание бумаги»,

Carriage Return и Line Feed), при вводе такие пары символов заменяются на один символ '\n' («продергивание бумаги»). Обратно, при выводе символ '\n' заменяется на пару "\r\n". Такими операционными системами являются MS DOS и MS Windows. В системе Unix строки разделяются одним символом '\n' (отсюда проистекает обозначение '\n', которое расшифровывается как new line). Таким образом, внутреннее представление текста всегда соответствует системе Unix, а внешнее — реально используемой операционной системе. Отметим также, что создатели операционной системы компьютеров Apple Macintosh выбрали, чтобы жизнь не казалась скучной, третий, отличный от двух предыдущих, вариант: текстовые строки разделяются одним символом '\r' «возврат каретки»!

Такое представление текстовых файлов восходит к тем уже далеким временам, когда еще не было компьютерных мониторов и для просмотра текста использовались электрифицированные пишущие машинки или посимвольные принтеры. Текстовый файл фактически представлял собой программу печати на пишущей машинке и, таким образом, содержал команды возврата каретки и продергивания бумаги в конце каждой строки.

Функции бинарной записи в файл *fwrite* аналогична функции чтения *fread*. Она имеет следующий прототип:

```
size_t fwrite(
    char *buffer,    // Массив записываемых данных
    size_t elemSize, // Размер одного элемента
    size_t numElems, // Число записываемых элементов
    FILE *f          // Указатель на структуру FILE
);
```

Функция возвращает число реально записанных элементов, которое может быть меньше, чем *numElems*, если при записи произошла ошибка — например, не хватило свободного пространства на диске. Пример использования функции *fwrite*:

```
FILE *f;
double buff[100];
size_t num;
. . .

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"
```

```
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для записи");
    exit(1);  // завершить работу программы с кодом 1
}

// Записываем 100 вещественных чисел в файл
res = fwrite(buff, sizeof(double), 100, f);
// В случае успеха res == 100
```

Закрытие файла: функция fclose

По окончании работы с файлом его надо обязательно закрыть. Система обычно запрещает полный доступ к файлу до тех пор, пока он не закрыт. (Например, в нормальном режиме система запрещает одновременную запись в файл для двух разных программ.) Кроме того, информация реально записывается полностью в файл лишь в момент его закрытия. До этого она может содержаться в оперативной памяти (в так называемой файловой кеш-памяти), что при выполнении многочисленных операций записи и чтения значительно ускоряет работу программы.

Для закрытия файла используется функция *fclose* с прототипом

```
int fclose(FILE *f);
```

В случае успеха функция *fclose* возвращает ноль, при ошибке — отрицательное значение (точнее, константу «конец файла» EOF, определенную в системных заголовочных файлах как минус единица). При ошибке можно воспользоваться функцией *perror*, чтобы напечатать причину ошибки. Отметим, что ошибка при закрытии файла — явление очень редкое (чего не скажешь в отношении открытия файла), так что анализировать значение, возвращаемое функцией *fclose*, в общем-то, не обязательно. Пример использования функции *fclose*:

```
FILE *f;

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
```

```
        perror("Не могу открыть файл для записи");
        exit(1);  // завершить работу программы с кодом 1
    }

    . . .

    // Закрываем файл
    if (fclose(f) < 0) {
        // Напечатать сообщение об ошибке
        perror("Ошибка при закрытии файла");
    }
```

Пример: подсчет числа символов и строк в текстовом файле

В качестве содержательного примера использования рассмотренных выше функций файлового ввода приведем программу, которая подсчитывает число символов и строк в текстовом файле. Программа сначала вводит имя файла с клавиатуры. Для этого используется функция *scanf* ввода по формату из входного потока, для ввода строки применяется формат "%s". Затем файл открывается на чтение как бинарный (это означает, что при чтении не будет происходить никакого преобразования разделителей строк). Используя в цикле функцию чтения *fread*, мы считываем содержимое файла порциями по 512 байтов, каждый раз увеличивая суммарное число прочитанных символов. После чтения очередной порции сканируется массив прочитанных символов и подсчитывается число символов '\n' продергивания бумаги, которые записаны в концах строк текстовых файлов как в системе Unix, так и в MS DOS или MS Windows. В конце закрывается файл и печатается результат.

```
//
// Файл "wc.cpp"
// Подсчет числа символов и строк в текстовом файле
//
#include <stdio.h>  // Описания функций ввода-вывода
#include <stdlib.h> // Описание функции exit

int main() {
```

```
char fileName[256]; // Путь к файлу
FILE *f;            // Структура, описывающая файл
char buff[512];     // Массив для ввода символов
size_t num;         // Число прочитанных символов
int numChars = 0;   // Суммарное число символов := 0
int numLines = 0;   // Суммарное число строк := 0
int i;              // Переменная цикла

printf("Введите имя файла: ");
scanf("%s", fileName);

f = fopen(fileName, "rb"); // Открываем файл на чтение
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для чтения");
    exit(1);  // закончить работу программы с кодом 1
              // ошибочного завершения
}

while ((num = fread(buff, 1, 512, f)) > 0) { // Читаем
    // блок из 512 символов. num -- число реально
    // прочитанных символов. Цикл продолжается, пока
    // num > 0

    numChars += num; // Увеличиваем число символов

    // Подсчитываем число символов перевода строки
    for (i = 0; i < num; ++i) {
        if (buff[i] == '\n') {
            ++numLines; // Увеличиваем число строк
        }
    }
}

fclose(f);

// Печатаем результат
```

```
    printf("Число символов в файле = %d\n", numChars);
    printf("Число строк в файле = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы: она применяется к собственному тексту, записанному в файле “wc.cpp”.

```
Введите имя файла: wc.cpp
Число символов в файле = 1635
Число строк в файле = 50
```

Форматный ввод-вывод: функции fscanf и fprintf

В отличие от функции бинарного ввода *fread*, которая вводит байты из файла без всякого преобразования непосредственно в память компьютера, функция *форматного ввода fscanf* предназначена для ввода информации с преобразованием ее из текстового представления в бинарное. Пусть информация записана в текстовом файле в привычном для человека виде (т.е. так, что ее можно прочитать или ввести в файл, используя текстовый редактор). Функция *fscanf* читает информацию из текстового файла и преобразует ее во внутреннее представление данных в памяти компьютера. Информация о количестве читаемых элементов, их типах и особенностях представления задается с помощью *формата*. В случае функции ввода формат — это строка, содержащая описания одного или нескольких вводимых элементов. Форматы, используемые функцией *fscanf*, аналогичны применяемым функцией *scanf*, они уже неоднократно рассматривались (см. раздел 3.5.4). Каждый элемент формата начинается с символа процента ‘%’. Наиболее часто используемые при вводе форматы приведены в таблице:

%d	целое десятичное число типа int (d — от decimal)
%lf	вещ. число типа double (lf — от long float)
%c	один символ типа char
%s	ввод строки. Из входного потока выделяется слово, ограниченное пробелами или символами перевода строки ’\n’. Слово помещается в массив символов. Конец слова отмечается нулевым байтом.

Прототип функции *fscanf* выглядит следующим образом:

```
int fscanf(FILE *f, const char *format, ...);
```

Многоточие здесь означает, что функция имеет переменное число аргументов, большее двух, и что количество и типы аргументов, начиная с третьего, произвольны. На самом деле, фактические агрументы, начиная с третьего, должны быть указателями на вводимые переменные. Несколько примеров использования функции *fscanf*:

```
int n, m; double a; char c; char str[256];
FILE *f;
. . .
fscanf(f, "%d", &n);  // Ввод целого числа
fscanf(f, "%lf", &a); // Ввод вещественного числа
fscanf(f, "%c", &c);  // Ввод одного символа
fscanf(f, "%s", str); // Ввод строки (выделяется очередное
                      // слово из входного потока)
fscanf(f, "%d%d", &n, &m); // Ввод двух целых чисел
```

Функция *fscanf* возвращает число успешно введенных элементов. Таким образом, возвращаемое значение всегда меньше или равно количеству процентов внутри форматной строки (которое равно числу фактических аргументов минус 2).

Функция *fprintf* используется для форматного вывода в файл. Данные при выводе преобразуются в их текстовое представление в соответствии с форматной строкой. Ее отличие от форматной строки, используемой в функции ввода *fscanf*, заключается в том, что она может содержать не только форматы для преобразования данных, но и обычные символы, которые записываются без преобразования в файл. Форматы, как и в случае функции *fscanf*, начинаются с символа процента '%'. Они аналогичны форматам, используемым функцией *fscanf*. Небольшое отличие заключается в том, что форматы функции *fprintf* позволяют также управлять представлением данных, например, указывать количество позиций, отводимых под запись числа, или количество цифр после десятичной точки при выводе вещественного числа. Некоторые типичные примеры форматов

для вывода приведены в следующей таблице:

%d	вывод целого десятичного числа
%10d	вывод целого десятичного числа, для записи числа отводится 10 позиций, запись при необходимости дополняется пробелами слева
%lf	вывод вещ. числа типа double в форме с фиксированной десятичной точкой
%.3lf	вывод вещественного числа типа double с печатью трех знаков после десятичной точки
%12.3lf	вывод вещественного числа типа double с тремя знаками после десятичной точки, под число отводится 12 позиций
%c	вывод одного символа
%s	вывод строки, т.е. массива символов. Конец строки задается нулевым байтом

Прототип функции *fprintf* выглядит следующим образом:

```
int fprinf(FILE *f, const char *format, ...);
```

Многоточие, как и в случае функции *fscanf*, означает, что функция имеет переменное число аргументов. Количество и типы аргументов, начиная с третьего, должны соответствовать форматной строке. В отличие от функции *fscanf*, фактические аргументы, начиная с третьего, представляют собой выводимые значения, а не указатели на переменные. Для примера рассмотрим небольшую программу, выводящую данные в файл “tmp.dat”:

```
#include <stdio.h>  // Описания функций ввода вывода
#include <math.h>   // Описания математических функций
#include <string.h> // Описания функций работы со строками

int main() {
    int n = 4, m = 6; double x = 2.;
    char str[256] = "Print test";
    FILE *f = fopen("tmp.dat", "wt"); // Открыть файл
    if (f == 0) {                     // для записи
        perror("Не могу открыть файл для записи");
```

```
        return 1; // Завершить программу с кодом ошибки
    }
    fprintf(f, "n=%d, m=%d\n", m, n);
    fprintf(f, "x=%.4lf, sqrt(x)=%.4lf\n", x, sqrt(x));
    fprintf(
        f, "Строка \"%s\" содержит %d символов.\n",
        str, strlen(str)
    );
    fclose(f);  // Закрыть файл
    return 0;   // Успешное завершение программы
}
```

В результате выполнения этой программы в файл “tmp.dat” будет записан следующий текст:

```
n=6, m=4
x=2.0000, sqrt(x)=1.4142
Строка "Print test" содержит 10 символов.
```

В последнем примере форматная строка содержит внутри себя двойные апострофы. Это специальные символы, выполняющие роль ограничителей строки, поэтому внутри строки их надо экранировать (т.е. защищать от интерпретации как специальных символов) с помощью обратной косой черты \, которая, напомним, в системе Unix и в языке Си выполняет роль защитного символа. Отметим также, что мы воспользовались стандартной функцией *sqrt*, вычисляющей квадратный корень числа, и стандартной функцией *strlen*, вычисляющей длину строки.

Понятие потока ввода или вывода

В операционной системе Unix и в других системах, использующих идеи системы Unix (например, MS DOS и MS Windows), применяется понятие потока ввода или вывода. Поток представляет собой последовательность байтов. Различают потоки ввода и вывода. Программа может читать данные из потока ввода и выводить данные в поток вывода. Программы можно запускать в конвейере, когда поток вывода первой программы является потоком ввода второй программы и т.д. Для запуска двух программ в конвейере используется символ

вертикальной черты | между именами программ в командной строке. Например, командная строка

```
ab | cd | ef
```

означает, что поток вывода программы *ab* направляется на вход программе *cd*, а поток вывода программы *cd* — на вход программе *ef*. По умолчанию, потоком ввода для программы является клавиатура, поток вывода назначен на терминал (или, как говорят программисты, на консоль). Потоки можно переправлять в файл или из файла, используя символы «больше» > и «меньше» <, которые можно представлять как воро́нки. Например, командная строка

```
abcd > tmp.res
```

перенаправляет выходной поток программы *abcd* в файл "tmp.res", т.е. данные будут выводиться в файл вместо печати на экране терминала. Соответственно, командная строка

```
abcd < tmp.dat
```

заставляет программу *abcd* читать исходные данные из файла "tmp.dat" вместо ввода с клавиатуры. Командная строка

```
abcd < tmp.dat > tmp.res
```

перенаправляет как входной, так и выходной потоки: входной назначается на файл "tmp.dat", выходной — на файл "tmp.res".

В Си работа с потоком не отличается от работы с файлом. Доступ к потоку осуществляется с помощью переменной типа `FILE *`. В момент начала работы Си-программы открыты три потока:

stdin — стандартный входной поток. По умолчанию он назначен на клавиатуру;

stdout — стандартный выходной поток. По умолчанию он назначен на экран терминала;

stderr — выходной поток для печати информации об ошибках. Он также назначен по умолчанию на экран терминала.

Переменные *stdin*, *stdout*, *stderr* являются глобальными, они описаны в стандартном заголовочном файле “stdio.h”. Операции файлового ввода-вывода могут использовать эти потоки, например, строка

```
fscanf(stdin, "%d", &n);
```

вводит значение целочисленной переменной n из входного потока. Строка

```
fprintf(stdout, "n = %d\n", n);
```

выводит значение переменой n в выходной поток. Строка

```
fprintf(stderr, "Ошибка при открытии файла\n");
```

выводит указанный текст в поток *stderr*, используемый обычно для печати сообщений об ошибках. Функция *perror* также выводит сообщения об ошибках в поток *stderr*.

По умолчанию, стандартный выходной поток и выходной поток для печати ошибок назначены на экран терминала. Однако, операция перенаправления вывода в файл > действует только на стандартный выходной поток. Например, в результате выполнения командной строки

```
abcd > tmp.res
```

обычный вывод программы *abcd* будет записываться в файл “tmp.res”, а сообщения об ошибках по-прежнему будут печататься на экране терминала. Для того чтобы перенаправить в файл “tmp.log” стандартный поток печати ошибок, следует использовать командную строку

```
abcd 2> tmp.log
```

(между двойкой и символом > не должно быть пробелов!). Двойка здесь означает номер перенаправляемого потока. Стандартный входной поток имеет номер 0, стандартный выходной поток — номер 1, стандартный поток печати ошибок — номер 2. Данная команда перенаправляет только поток *stderr*, поток *stdout* по-прежнему будет выводиться на терминал. Можно перенаправить потоки в разные файлы:

```
abcd 2> tmp.log > tmp.res
```

Таким образом, существование двух разных потоков вывода позволяет при необходимости «отделить мух от котлет», т.е. направить нормальный вывод и вывод информации об ошибках в разные файлы.

Функции scanf и printf ввода и вывода в стандартные потоки

Поскольку ввод из стандартного входного потока, по умолчанию назначенного на клавиатуру, и вывод в стандартный выходной поток, по умолчанию назначенный на экран терминала, используются особенно часто, библиотека функций ввода-вывода Си предоставляет для работы с этими потоками функции *scanf* и *printf*. Они отличаются от функций *fscanf* и *fprintf* только тем, что у них отсутствует первый аргумент, означающий поток ввода или вывода. Строка

```
scanf(format, ...); // Ввод из станд. входного потока
```

эквивалентна строке

```
fscanf(stdin, format, ...); // Ввод из потока stdin
```

Аналогично, строка

```
printf(format, ...); // Вывод в станд. выходной поток
```

эквивалентна строке

```
fprintf(stdout, format, ...); // Вывод в поток stdout
```

Функции текстового преобразования sscanf и sprintf

Стандартная библиотека ввода-вывода Си предоставляет также две замечательные функции *sscanf* и *sprintf* ввода и вывода не в файл или поток, а в строку символов (т.е. массив байтов), расположенную в памяти компьютера. Мнемоника названий функций следующая: в названии функции *fscanf* первая буква *f* означает файл (file), т.е. ввод производится из файла; соответственно, в названии функции *sscanf* первая буква *s* означает строку (string), т.е. ввод производится из текстовой строки. (Последняя буква *f* в названиях этих функций означает «форматный»). Первым аргументом функций *sscanf* и *sprintf* является строка (т.е. массив символов, ограниченный нулевым байтом), из которой производится ввод или в которую производится вывод. Эта строка как бы стоит на месте файла в функциях *fscanf* и *fprintf*.

Фунции *sscanf* и *sprintf* удобны для преобразования данных из текстового представления во внутреннее и обратно. Например, в результате выполнения фрагмента

```
char txt[256] = "-135.76"; double x;
sscanf(txt, "%lf", &x);
```

текстовая запись вещественного числа, содержащаяся в строке *txt*, преобразуется во внутреннее представление вещественного числа, результат записывается в переменную x. Обратно, при выполнения фрагмента

```
char txt[256]; int x = 12345;
sprintf(txt, "%d", x);
```

значение целочисленной переменной x будет преобразовано в текстовую форму и записано в строку *txt*, в результате строка будет содержать текст "12345", ограниченный нулевым байтом.

Для преобразования данных из текстового представления во внутреннее в стандартной библиотеке Си имеются также функции *atoi* и *atof* с прототипами

```
int atoi(const char *txt);    // текст => int
double atof(const char *txt); // текст => double
```

Функция *atoi* преобразует текстовое представление целого числа типа *int* во внутреннее. Соответственно, функция *atof* преобразует текстовое представление вещественного числа типа *double*. Мнемоника имен следующая:

atoi означает "ch**a**racter **to i**nteger";
atof означает "ch**a**racter **to f**loat".

(В последнем случае float следует понимать как плавающее, т.е. вещественное, число, имеющее тип *double*, а вовсе не *float*! Тип *float* является атавизмом и практически не используется.)

Прототипы функций *atoi* и *atof* описаны в стандартном заголовочном файле "stdlib.h", а не "stdio.h", поэтому при их использовании надо подключать этот файл:

```
#include <stdlib.h>
```

(вообще-то, это можно делать всегда, поскольку "stdlib.h" содержит описания многих полезных функций, например, функции завершения программы *exit*, генератора случайных чисел *rand* и др.).

Отметим, что аналогов функции *sprintf* для обратного преобразования из внутреннего в текстовое представление в стандартной библиотеке Си нет. Компилятор Си фирмы Borland предоставляет

функции *itoa* и *ftoa*, однако, эти функции не входят в стандарт и другими компиляторами не поддерживаются, поэтому пользоваться ими не следует.

Другие полезные функции ввода-вывода

Стандартная библиотека ввода-вывода Си содержит ряд других полезных функций ввода-вывода. Отметим некоторые из них.

Посимвольный ввод-вывод	
int fgetc(FILE *f); int fputc(int c, FILE *f);	ввести символ из потока f вывести символ в поток f
Построковый ввод-вывод	
char *fgets(char *line, int size, FILE *f); char *fputs(char *line, FILE *f);	ввести строку из потока f вывести строку в поток f
Позиционирование в файле	

int fseek(FILE *f,
long offset, int whence);
long ftell(FILE *f);
 	
int feof(FILE *f);
	установить текущую
	позицию в файле f
	получить текущую
	позицию в файле f
	проверить, достигнут ли
	конец файла f

Функция *fgetc* возвращает код введенного символа или константу `EOF` (определенную как минус единицу) в случае конца файла или ошибки чтения. Функция *fputc* записывает один символ в файл. При ошибке *fputc* возвращает константу `EOF` (т.е. отрицательное значение), в случае удачи — код выведенного символа *c* (неотрицательное значение).

В качестве примера использования функции *fgetc* перепишем рассмотренную ранее программу *wc*, подсчитывающую число символов и строк в текстовом файле:

```
//
// Файл "wc1.cpp"
// Подсчет числа символов и строк в текстовом файле
// с использованием функции чтения символа fgetc
//
```

```
#include <stdio.h>  // Описания функций ввода-вывода

int main() {
    char fileName[256]; // Путь к файлу
    FILE *f;            // Структура, описывающая файл
    int c;              // Код введенного символа
    int numChars = 0;   // Суммарное число символов := 0
    int numLines = 0;   // Суммарное число строк := 0

    printf("Введите имя файла: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); // Открываем файл
    if (f == 0) { // При ошибке открытия файла
        // Напечатать сообщение об ошибке
        perror("Не могу открыть файл для чтения");
        return 1; // закончить работу программы с кодом 1
    }

    while ((c = fgetc(f)) != EOF) { // Читаем символ
        // Цикл продолжается, пока c != -1 (конец файла)

        ++numChars; // Увеличиваем число символов

        // Подсчитываем число символов перевода строки
        if (c == '\n') {
            ++numLines; // Увеличиваем число строк
        }
    }

    fclose(f);

    // Печатаем результат
    printf("Число символов в файле = %d\n", numChars);
    printf("Число строк в файле = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
```

```
}
```

Пример выполнения программы *wc1* в применении к собственному тексту, записанному в файле "wc1.cpp":

```
Введите имя файла: wc1.cpp
Число символов в файле = 1334
Число строк в файле = 44
```

Функция *fgets* с прототипом

```
char *fgets(char *line, int size, FILE *f);
```

выделяет из файла или входного потока *f* очередную строку и записывает ее в массив символов *line*. Второй аргумент *size* указывает размер массива для записи строки. Максимальная длина строки на единицу меньше, чем *size*, поскольку всегда в конец считанной строки добавляется нулевой байт. Функция сканирует входной поток до тех пор, пока не встретит симол перевода строки '\n' или пока число введенных символов не станет равным *size* – 1. Символ перевода строки '\n' также записывается в массив непосредственно перед терминирующим нулевым байтом. Функция возвращает указатель *line* в случае успеха или нулевой указатель при ошибке или конце файла.

> Раньше в стандартную библиотеку Си входила также функция *gets* с прототипом
>
> ```
> char *gets(char *line);
> ```
>
> которая считывала очередную строку из стандартного входного потока и помещала ее в массив, адрес которого являлся ее единственным аргументом. Отличие от функции *fgets* в том, что не указывается размер массива *line*. В результате, подав на вход функции *gets* очень длинную строку, можно добиться переполнения массива и стереть или подменить участок памяти, используемый программой. Если программа имеет привилегии суперпользователя, то применение в ней функции *gets* открывает путь к взлому системы, который использовался хакерами. Поэтому в настоящее время функция *gets* считается опасной и применение ее не рекомендовано. Вместо *gets* следует использовать *fgets* с третьим аргументом *stdin*.

При выполнении файловых операций исполняющая система поддерживает указатель текущей позиции в файле. При чтении или записи n байтов указатель текущей позиции увеличивается на n; таким

образом, чтение или запись происходят последовательно. Библиотека ввода-вывода Си предоставляет, однако, возможность нарушать эту последовательность путем позиционирования в произвольную точку файла. Для этого используется стандартная функция *fseek* с прототипом

```
int fseek(FILE *f, long offset, int whence);
```

Первый аргумент *f* функции определяет файл, для которого производится операция позиционирования. Второй аргумент *offset* задает смещение в байтах, оно может быть как положительным, так и отрицательным. Третий аргумент *whence* указывает, откуда отсчитывать смещение. Он может принимать одно из трех значений, заданных как целые константы в стандартном заголовочном файле "stdio.h":

SEEK_CUR	смещение отсчитывается от текущей позиции;
SEEK_SET	смещение отсчитывается от начала файла;
SEEK_END	смещение отсчитывается от конца файла.

Например, фрагмент

```
fseek(f, 0, SEEK_SET);
```

устанавливает текущую позицию в начало файла. Фрагмент

```
fseek(f, -4, SEEK_END);
```

устанавливает текущую позицию в четырех байтах перед концом файла. Наконец, фрагмент

```
fseek(f, 12, SEEK_CUR);
```

продвигает текущую позицию на 12 байтов вперед.

Отметим, что смещение может быть положительным даже при использовании константы SEEK_END (т.е. при позиционировании относительно конца файла): в этом случае при следующей записи размер файла соответственно увеличивается.

Функция возвращает нулевое значение в случае успеха и отрицательное значение EOF (равное -1) при неудаче — например, если указанное смещение некорректно при заданной операции или если файл или поток не позволяет выполнять прямое позиционирование.

Узнать текущую позицию относительно начала файла можно с помощью функции *ftell* с прототипом

```
long ftell(FILE *f);
```

Функция *ftell* возвращает текущую позицию (неотрицательное значение) в случае успеха или отрицательное значение −1 при неудаче (например, если файл не разрешает прямое позиционирование).

Наконец, узнать, находится ли текущая позиция в конце файла, можно с помощью функции *feof* с прототипом

```
int feof(FILE *f);
```

Она возвращает ненулевое значение (т.е. «истину»), если конец файла достигнут, и нулевое значение (т.е. «ложь») в противном случае. Например, в следующем фрагменте в цикле проверяется, достигнут ли конец файла, и, если нет, считывается очередной байт:

```
FILE *f;
. . .
while (!feof(f)) {     // цикл пока не конец файла
    int c = fgetc(f); // | прочесть очередной байт
    . . .             // | . . .
}                     // конец цикла
```

3.9.4. Работа с текстами

Стандартная библиотека Си предоставляет набор функций для работы с текстами. К сожалению, большая часть из них ориентирована на представление символов в виде одного байта (во время разработки языка Си кодировка Unicode, в которой на символ отводится два байта, еще не существовала). Функции можно разделить на две группы:

1) функции, определяющие тип символа, — является ли он буквой, цифрой, пробелом, знаком препинания и т.п. Это функции описаны в стандартном заголовочном файле “ctype.h”. Увы, функции, касающиеся букв, работают только для латинского алфавита;

2) функции для работы с текстовыми строками. Строкой в Си считается последовательность байтов, ограниченная в конце нулевым байтом. Функции работы со строками описаны в стандартном заголовочном файле “string.h”.

Определение типов символов

Библиотека Си предоставляет следующие функции для определения типа символов, описанные в стандартном заголовочном файле "ctype.h":

int isdigit(int c);	символ c — цифра;
int isalpha(int c);	c — латинская буква;
int isspace(int c);	c — пробел, перевод строки и т.п.;
int ispunkt(int c);	c — знак препинания;
int isupper(int c);	c — прописная латинская буква;
int islower(int c);	c — строчная латинская буква;
int toupper(int c);	если c — лат. буква, то преобразовать c к прописной букве;
int tolower(int c);	если c — лат. буква, то преобразовать c к строчной букве.

Функции, начинающиеся с префикса *is*, возвращают ненулевое значение (т.е. «истину»), если символ с кодом c принадлежит указанному классу, и нулевое значение («ложь») в противном случае. Функции *toupper* и *tolower* преобразуют латинские буквы к верхнему или нижнему регистру, на остальных символах они действуют тождественно.

В качестве примера использования функции *isspace* модифицируем программу "wc.cpp", подсчитывающую число строк и символов в текстовом файле. Добавим в нее подсчет слов. Будем считать словами любые связные группы символов, разделенные пробелами, табуляциями или разделителями строк.

```
//
// Файл "wc2.cpp"
// Подсчет числа символов, слов и строк в текстовом файле
//
#include <stdio.h>  // Описания функций ввода-вывода
#include <ctype.h>  // Описания типов символов

int main() {
    char fileName[256]; // Путь к файлу
    FILE *f;            // Структура, описывающая файл
    int c;              // Код введенного символа
    int numChars = 0;   // Суммарное число символов := 0
```

```
int numLines = 0;   // Суммарное число строк := 0
int numWords = 0;   // Суммарное число слов := 0
bool readingWord = false; // Читаем слово := false

printf("Введите имя файла: ");
scanf("%s", fileName);

f = fopen(fileName, "rb"); // Открываем файл
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для чтения");
    return 1; // закончить работу программы с кодом 1
}

while ((c = fgetc(f)) != EOF) { // Читаем символ
    // Цикл продолжается, пока c != -1 (конец файла)

    ++numChars; // Увеличиваем число символов

    // Подсчитываем число символов перевода строки
    if (c == '\n') {
        ++numLines; // Увеличиваем число строк
    }

    // Подсчитываем число слов
    if (!isspace(c)) {      // если c не пробел
        if (!readingWord) { //  если не читаем слово
            ++numWords;     //   увеличить число слов
            readingWord = true; // читаем слово:=true
        }                   //  конец если
    } else {                // иначе
        readingWord = false;//  читаем слово:=false
    }                       // конец если
}

fclose(f);
```

```
    // Печатаем результат
    printf("Число символов в файле = %d\n", numChars);
    printf("Число слов = %d\n", numWords);
    printf("Число строк = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы *wc2* в применении к собственному тексту, записанному в файле “wc2.cpp”:

```
Введите имя файла: wc2.cpp
Число символов в файле = 1998
Число слов = 260
Число строк = 57
```

Работа с текстовыми строками

Стандартная библиотека Си предоставляет средства вычисления длины строки, копирования, сравнения, соединения (конкатенации) строк, поиска вхождений одной строки в другую. Функции описаны в стандартном заголовочном файле “string.h”. Прототипы наиболее часто используемых функций приведены ниже.

Определение длины строки

`size_t strlen(const char *s);` длина строки.

Копирование строк

`char *strcpy(char *dst, const char *src);` копировать строку *src* в строку *dst*;

`char *strncpy(char *dst, const char *src, size_t maxlen);` копировать строку *src* в *dst*, не более *maxlen* символов;

`char *strcat(char *dst, const char *src);` копировать строку *src* в конец *dst* (конкатенация строк).

Работа с произвольными массивами байтов

```
void *memmove(void *dst, const void *src, size_t len);
```

копировать область памяти с адресом *src* размером *len* байтов в область памяти с адресом *dst*;

`void *memset(void *dst, int value, size_t len);` записать значение *value* в каждый из *len* байтов, начиная с адреса *dst*.

Сравнение строк

`int strcmp(const char *s1, const char *s2);` лексикографическое сравнение строк $s1$ и $s2$. Результат нулевой, если строки равны, отрицательный, если первая строка меньше второй, и положительный, если первая строка больше второй;

`int strncmp(const char *s1, const char *s2, size_t maxlen);` сравнение строк $s1$ и $s2$, сравнивается не более *maxlen* символов;

`int memcmp(const void *m1, const void *m2, size_t len);` сравнение областей памяти с адресами $m1$ и $m2$ размером *len* каждая.

Поиск

`char *strchr(const char *s, int c);` найти первое вхождение символа c в строку s. Функция возвращает указатель на найденный символ или ноль в случае неудачи;

`char *strstr(const char *s1, const char *s2);` найти первое вхождение строки $s2$ в строку $s1$. Функция возвращает указатель на найденную подстроку в $s1$, равную строке $s2$, или ноль в случае неудачи.

Пример: программа «Записная книжка»

В качестве примера работы с текстовой информацией рассмотрим программу «Записная книжка». Записная книжка хранит имена людей и их телефоны. Под именем понимается полное имя, включающее фамилию, имя, отчество в произвольном формате: имя представляется любой текстовой строкой. Телефон также представляется

текстовой строкой (представление с помощью целого числа не подходит, потому что номер может быть очень длинным и содержать специальные символы, например, знак плюс).

Программа позволяет добавлять записи в записную книжку, удалять записи и искать номер телефона по имени. При поиске не обязательно вводить имя полностью, достаточно нескольких первых символов. Можно также распечатать содержимое всей записной книжки. В перерывах между запусками программы информация сохраняется в файле "NoteBook.dat".

Записная книжка соответствует структуре данных «нагруженное множество», которая будет рассмотрена в разделе 4.6.1 и для которой имеется ряд реализаций, обеспечивающих быстрый поиск и модификацию. Мы, однако, ограничимся сейчас простейшей реализацией: не упорядочиваем записи по алфавиту и применяем последовательный поиск. Записи хранятся в массиве, его максимальный размер равен 1000 элементов. Новая запись добавляется в конец массива. При удалении записи из книжки последняя запись переписывается на место удаленной.

Для записей используется структурный тип *NameRecord*, определенный следующим образом:

```
typedef struct {
    char *name;
    char *phone;
} NameRecord;
```

Здесь поле *name* структуры является указателем на имя абонента, которое представляет собой текстовую строку. Номер телефона также задается строкой, указатель на которую записывается в поле *phone*. Пространство под строки захватывается в динамической памяти с помощью функции *malloc*. При удалении записи память освобождается с помощью функции *free*.

Записи хранятся в массиве *records*:

```
const int MAXNAMES = 1000;
static NameRecord records[MAXNAMES];
static int numRecords = 0;
```

Константа *MAXNAMES* задает максимально возможный размер массива *records*. Текущее количество записей (т.е. реальный размер массива) хранится в переменной *numRecords*.

В начале работы программа вводит содержимое записной книжки из файла "NoteBook.dat", имя которого задается как константный указатель на константную строку:

```
const char* const NoteBookFile = "NoteBook.dat";
```

В конце работы содержимое записной книжки сохраняется в файле. Для ввода используется функция *loadNoteBook*, для сохранения — функция *saveNoteBook* с прототипами

```
static bool loadNoteBook();
static bool saveNoteBook();
```

Каждой записи соответствует пара строк файла. Пусть, например, имя абонента "Иван Петров", телефон — "123-45-67". Этой записи соответствует пара строк

```
name=Иван Петров
phone=123-45-67
```

Записи в файле разделяются пустыми строками.

Сохранение файла выполняется только в случае, когда содержимое записной книжки изменялось в процессе работы. За это отвечает переменная

```
static bool modified;
```

которая принимает значение *true*, если хотя бы раз была выполнена одна из команд, меняющих содержимое записной книжки.

Работа с записной книжкой происходит в интерактивном режиме. Пользователь вводит команду, программа ее выполняет. При выполнении команды программа просит ввести дополнительную информацию, если это необходимо, и печатает результат выполнения. Команды записываются латинскими буквами, чтобы исключить возможные проблемы с русскими кодировками. После команды может идти имя абонента. Реализованы следующие команды:

add — добавить запись. Программа просит ввести имя абонента, если оно не указано непосредственно после команды, затем его телефон, после этого запись добавляется в записную книжку, или телефон изменяется, если данное имя уже содержится в книжке;

remove — удалить запись. Программа просит ввести имя абонента, если оно не указано непосредственно после команды, и удаляет запись из книжки, если она там присутствует;

find — найти запись. Программа при необходимости просит ввести имя абонента и печатает либо его телефон, либо сообщение, что данного имени в книжке нет;

modify — изменить номер телефона. Программа просит ввести имя абонента, если оно не указано непосредственно после команды. Затем осуществляется поиск записи с данным именем. В случае успеха программа просит ввести новый номер телефона, после этого старый номер заменяется на новый; в противном случае, печатается сообщение, что имя не содержится в книжке;

show — напечатать записи, для которых имена начинаются с данного префикса. Если имя или начало имени не указано непосредственно после команды, то печатается все содержимое записной книжки. Если указано начало имени, то печатаются все записи, для которых начало имени совпадает с заданным;

help — напечатать подсказку;

quit — закончить работу.

Каждой команде соответствует отдельная функция, выполняющая команду. Например, команде *add* соответствует функция *onAdd*, команде *find* — функция *onFind* и т.д. Начало искомого имени и его длина, если имя указано в командной строке, передаются через статические переменные

```
static char namePrefix[256];
static int namePrefixLen;
```

Если имя не указано в командной строке, то для его ввода вызывается функция *readName*, которая просит пользователя ввести имя с клавиатуры, считывает имя и заполняет переменные *namePrefix* и *namePrefixLen*.

Для поиска записи с заданным началом имени используется функция *search* с прототипом

```
static int search(
    const char *namePrefix, // Начало искомого имени
    int startPosition       // Позиция начала поиска
);
```

Ей передается начало искомого имени и индекс элемента массива, с которого начинается поиск. Второй аргумент нужен для того, чтобы последовательно находить имена с одинаковым префиксом (например, имена, начинающиеся с заданной буквы). Функция возвращает индекс найденного элемента в случае успеха или отрицательное значение -1 при неудаче. Применяется последовательный поиск, при котором просматриваются все элементы, начиная со стартовой позиции. Для сравнения имен используется стандартная функция *strncmp(s1, s2, n)*, которая сравнивает n первых символов строк *s1* и *s2*. При поиске в качестве *s1* используется заданное начало имени, в качестве *s2* — очередное имя из записной книжке, n равно длине начала имени.

Для ввода строки с клавиатуры мы используем вспомогательную функцию *readLine* с прототипом

```
static bool readLine(
    char *buffer, int maxlen, int *len
);
```

Функция вводит строку из стандартного входного потока, используя библиотечную функцию *fgets*. Затем из конца строки удаляются символы-разделители строк '\r' и '\n'. Введенная строка помещается в массив *buffer* с максимальной длиной *maxlen*. Реальная длина введенной строки записывается в переменную, адрес которой передается через указатель *len*.

Полный текст программы:

```
// Файл "NoteBook.cpp"
// Программа "Записная книжка"
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef struct {
    char *name;  // Указатель на имя
    char *phone; // Указатель на телефон
} NameRecord;

// Максимальный размер записной книжки
const int MAXNAMES = 1000;

// Массив записей
static NameRecord records[MAXNAMES];

// Текущее количество записей в книжке
static int numRecords = 0;

// Имя файла для сохранения содержимого книжки
const char* const NoteBookFile = "NoteBook.dat";

static bool modified;
static char namePrefix[256]; // Начало имени,
static int namePrefixLen;    //   его длина
static char phone[256];      // Телефон,
static int phoneLen;         //   его длина

// Прототипы функций
static bool loadNoteBook(); // Загрузить книжку из файла
static bool saveNoteBook(); // Сохранить книжку в файле
static bool readLine(       // Ввести строку с клавиатуры
    char *line, int maxlen, int *len
);
```

```
static bool readName();     // Ввести имя с клавиатуры

static int search(          // Поиск имени в массиве
    const char *namePrefix, //   начало искомого имени
    int startPosition       //   позиция начала поиска
);

static void releaseMemory(); // Освободить память

// Прототипы функций, реализующих команды
static void onAdd();
static void onRemove();
static void onFind();
static void onModify();
static void onShow();
static void onHelp();

int main() {
    char line[256];      // Введенная строка
    int lineLen;         // Длина строки
    int comBeg,          // Индексы начала
        comEnd;          //      и за-конца команды
    int comLen;          // Длина команды
    const char *command; // Указатель на начало команды
    int nameBeg;         // Индекс начала имени
    int i;               // Индекс в массиве line

    printf("Программа \"Записная книжка\"\n");
    onHelp();           // Напечатать подсказку

    loadNoteBook(); // Загрузить содержимое книжки
                    //           из файла
    while (true) {
        printf(">");    // Приглашение на ввод команды

        // Ввести командную строку
        if (!readLine(line, 256, &lineLen)) {
```

```
    break;  // При ошибке завершить программу
}

// Разбор командной строки
// 1. Пропустить пробелы в начале строки
i = 0;
while (i < lineLen && isspace(line[i])) {
    ++i;
}

// 2. Выделить команду
comBeg = i;
while (i < lineLen && isalpha(line[i])) {
    ++i;
}
comEnd = i;

command = line + comBeg;  // Указ.начала команды
comLen = comEnd - comBeg; // ее длина
if (comLen <= 0)          // Команда пустая =>
    continue;             //   ввести следующую

// 3. Выделить префикс имени
i = comEnd;
// Пропустить пробелы перед именем
while (i < lineLen && isspace(line[i])) {
    ++i;
}
nameBeg = i;
if (nameBeg >= lineLen) {
    // Имя не указано в команде
    namePrefix[0] = 0; // Запомним, что имя
    namePrefixLen = 0; //            пустое
} else {
    // Имя задано в команде, запомним его.
    // Указ. на начало имени равен line+nameBeg,
    // длина имени равна lineLen-nameBeg.
```

```
            strcpy(namePrefix, line + nameBeg);
            namePrefixLen = lineLen - nameBeg;
        }

        // Разбор строки закончен.
        // Вызовем функцию, реализующую команду
        if (strncmp(command, "add", comLen) == 0) {
            onAdd();
        } else if (
            strncmp(command, "remove", comLen) == 0
        ) {
            onRemove();
        } else if (
            strncmp(command, "find", comLen) == 0
        ) {
            onFind();
        } else if (
            strncmp(command, "modify", comLen) == 0
        ) {
            onModify();
        } else if (
            strncmp(command, "show", comLen) == 0
        ) {
            onShow();
        } else if (
            strncmp(command, "help", comLen) == 0
        ) {
            onHelp();
        } else if (
            strncmp(command, "quit", comLen) == 0
        ) {
            break;      // Завершить работу
        } else {        // Неправильная команда =>
            onHelp();   //    напечатать подсказку
        }
    } // конец цикла while
```

```
    if (modified) {     // Если книжка модифицирована,
        saveNoteBook(); // то сохранить ее содержимое
    }

    releaseMemory(); // Освободить память
    return 0;   // Завершить программу с кодом успеха
}

static bool readLine( // Считать строку с клавиатуры
    char *line, int maxlen, int *len
) {
    int size;

    *line = 0; *len = 0; // Инициализация пустой строкой
    if (fgets(line, maxlen, stdin) == 0)
        return false;    // Ошибка ввода

    size = strlen(line); // Длина введенной строки

    // Удалить разделители строк из конца строки
    if (size > 0 && line[size - 1] == '\n') {
        line[size - 1] = 0; --size;
    }
    if (size > 0 && line[size - 1] == '\r') {
        line[size - 1] = 0; --size;
    }

    *len = size;    // Выдать длину строки
    return true;
}

static int search(          // Поиск имени в массиве
    const char *namePrefix, //  искомый префикс имени
    int startPosition       //  позиция начала поиска
) {
    int i = startPosition;
    int len = strlen(namePrefix);
```

```
    if (len == 0)
        return startPosition;

    while (i < numRecords) {
        if (
            strncmp( // Сравнить имя и префикс
                records[i].name, namePrefix, len
            ) == 0
        ) {
            return i;   // Нашли имя с данным префиксом
        }
        ++i;            // К следующей записи
    }
    return (-1);    // Имя не найдено
}

static bool readName() {    // Ввести имя с клавиатуры
    int size;
    printf("Введите имя:\n");
    return readLine(namePrefix, 256, &namePrefixLen);
}

static void onAdd() { // Добавить или изменить запись
    int i;

    if (namePrefixLen == 0) { // Если имя не задано в
        if (!readName()) {    // команде, то ввести его
            return;           // Ошибка ввода
        }
    }

    if (namePrefixLen == 0) {
        return;
    }

    // Ищем имя в книжке
```

```
    i = search(namePrefix, 0);
    if (i < 0) {
        // Имя не содержится в книжке, добавим его
        if (numRecords >= MAXNAMES) {
            printf("Переполнение книжки.\n");
            return;
        }
        i = numRecords; ++numRecords;

        // Захватим память под имя
        records[i].name = (char *)
            malloc(namePrefixLen + 1);
        // Запишем имя в книжку
        strcpy(records[i].name, namePrefix);
    }

    printf("Введите телефон:\n");
    readLine(phone, 256, &phoneLen);

    // Захватим память под телефон
    records[i].phone = (char *) malloc(phoneLen + 1);
    // Запишем телефон
    strcpy(records[i].phone, phone);

    modified = true; // Запомним, что содержимое менялось
}

static void onRemove() {    // Удалить запись
    int i;

    if (namePrefixLen == 0) { // Если имя не задано в
        if (!readName()) {    // команде, то ввести его
            return;           // Ошибка ввода
        }
    }

    if (namePrefixLen == 0) {
```

```
        return;
    }

    // Ищем имя в книжке
    i = search(namePrefix, 0);
    if (i < 0) { // Если имя не содержится в книжке,
        return;  // то ничего не делать
    }

    // Освободим память
    free(records[i].name);
    free(records[i].phone);

    // Перепишем последнюю запись на место удаляемой
    if (i >= 2 && i != numRecords - 1) {
        records[i] = records[numRecords - 1];
    }
    --numRecords;    // Уменьшим число записей

    modified = true; // Запомним, что содержимое менялось
}

static void onFind() {      // Найти запись
    int i;
    if (namePrefixLen == 0) {
        if (!readName()) {
            return; // Ошибка ввода
        }
    }
    i = search(namePrefix, 0);
    if (i < 0) {
        printf("Имя не найдено.\n");
    } else {
        printf("Имя: %s\n", records[i].name);
        printf("Телефон: %s\n", records[i].phone);
    }
}
```

```
static void onModify() { // Изменить номер телефона
    int i;
    if (namePrefixLen == 0) {
        if (!readName()) {
            return; // Ошибка ввода
        }
    }
    if (namePrefixLen == 0) {
        return;
    }

    // Ищем имя в книжке
    i = search(namePrefix, 0);
    if (i < 0) {
        printf("Имя не найдено.\n");
    } else {
        onAdd(); // Добавление модифицирует телефон,
    }            //     когда имя уже в книжке
}

// Показать все записи с данным префиксом
static void onShow() {
    int pos = 0;
    while (pos < numRecords) {
        if (namePrefixLen > 0) {
            pos = search(namePrefix, pos);
            if (pos < 0) {  // Имя не найдено =>
                break;      // выйти из цикла
            }
        }
        printf("Имя: %s\n", records[pos].name);
        printf("Телефон: %s\n\n", records[pos].phone);
        ++pos;
    }
}
```

```
static void onHelp() {
    printf(
        "Список команд:\n"
        "  add    добавить пару (имя, телефон)\n"
        "  remove удалить имя\n"
        "  find   найти имя и напечатать телефон\n"
        "  modify изменить телефон\n"
        "  show   напечатать записи с данным префиксом\n"
        "         (если префикс пустой, то все записи)\n"
        "  help   напечатать этот текст\n"
        "  quit   закончить работу\n"
    );
    printf("Введите команду и (не обязательно) имя.\n");
}

static bool loadNoteBook() { // Загрузить книжку из файла
    char line[256]; // Буфер для ввода строки
    int i;
    FILE *f = fopen(NoteBookFile, "rt");
    if (f == 0) {
        return false;
    }
    numRecords = 0;
    while (fgets(line, 256, f) != 0) {
        int len = strlen(line);
        char *name;
        char *phone = 0;

        // Удалим разделители строк
        if (len > 0 && line[len - 1] == '\n') {
            line[len - 1] = 0; --len;
        }
        if (len > 0 && line[len - 1] == '\r') {
            line[len - 1] = 0; --len;
        }

        if (len < 6 || strncmp(line, "name=", 5) != 0) {
```

```
            continue;   // К следующей строке
        }

        // Запомним имя
        name = (char *) malloc((len - 5) + 1);
        strcpy(name, line + 5);

        // Считаем строку с телефоном
        if (fgets(line, 256, f) != 0) {
            len = strlen(line);
            // Удалим разделители строк
            if (len > 0 && line[len - 1] == '\n') {
                line[len - 1] = 0; --len;
            }
            if (len > 0 && line[len - 1] == '\r') {
                line[len - 1] = 0; --len;
            }
            if (
                len >= 7 &&
                strncmp(line, "phone=", 6) == 0
            ) {
                // Запомним телефон
                phone = (char *) malloc((len - 6) + 1);
                strcpy(phone, line + 6);
            }
        }

        // Заполним новую запись
        records[numRecords].name = name;
        if (phone == 0) {
            phone = (char *) malloc(1);
            phone[0] = 0;   // Пустая строка
        }
        records[numRecords].phone = phone;
        ++numRecords;   // Увеличим число записей
    }
    return true;
```

```
}

static bool saveNoteBook() { // Сохранить книжку в файле
    int i;
    FILE *f = fopen(NoteBookFile, "wt");
    if (f == 0) {
        return false;
    }
    for (i = 0; i < numRecords; ++i) {
        fprintf(f, "name=%s\n", records[i].name);
        fprintf(f, "phone=%s\n\n", records[i].phone);
    }
    return true;
}

static void releaseMemory() {
    int i;
    for (i = 0; i < numRecords; ++i) {
        free(records[i].name);
        free(records[i].phone);
    }
}
```

Аргументы командной строки

До сих пор во всех примерах программ использовался ввод исходных данных либо с клавиатуры (т.е. из входного потока), либо из файла. Язык Си предоставляет также возможность указывать аргументы программы в командной строке.

Аргументы командной строки являются параметрами функции *main*, с которой начинается выполнение Си-программы. До сих пор применялся вариант функции *main* без параметров, однако, при необходимости доступа к аргументам командной строки можно использовать следующий заголовок функции *main*:

```
int main(int argc, char *argv[]) { . . . }
```

Здесь целая переменная *argc* равна числу аргументов, т.е. отдельных слов командной строки, а массив *argv* содержит указатели на стро-

ки, каждая из которых равна очередному слову командной строки. Нулевой элемент `argv[0]` равен имени программы. Таким образом, число аргументов *argc* всегда не меньше единицы.

Например, при запуске программы *testprog* с помощью командной строки

```
testprog -x abcd.txt efgh.txt
```

значение переменной *argc* будет равно 4, а массив *argv* будет содержать 4 строки "`testprog`", "`-x`", "`abcd.txt`" и "`efgh.txt`".

> В операционной системе Unix нулевой элемент массива *argv* содержит полный путь к файлу с выполняемой программой. В системах MS DOS и MS Windows строка *argv*[0] может быть равна как полному пути к файлу, так и первому слову командной строки (зависит от используемого командного процессора).

Пример программы, печатающей аргументы своей командной строки:

```
// Файл "comargs.cpp"
// Напечатать аргументы командной строки
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    printf("Число аргументов ком. строки = %d\n", argc);
    printf("Аргументы командной строки:\n");
    for (i = 0; i < argc; ++i) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

3.9.5. Разработка больших проектов

До сих пор все рассмотренные примеры программ на Си имели небольшой объем (за исключением, возможно, программы «Записная книжка»). Такие маленькие программы помещаются в один файл. Однако, реальные проекты имеют, как правило, значительно больший объем, измеряемый десятками, а чаще сотнями тысяч строк.

Реализовать такую программу в виде одного непрерывного текста, помещающегося в одном файле, невозможно. Большой проект разбивается на более или менее независимые модули, которые можно написать и отладить по отдельности. Обычно в один такой модуль выделяется группа функций, работающих над общими глобальными данными и в той или иной мере связанных логически между собой. В простейшем случае каждому модулю соответствуют два файла с исходными текстами: заголовочный, или h-файл, описывающий интерфейс модуля, и файл реализации — *c*- или *cpp*-файл. Заголовочный файл содержит прототипы функций модуля, описания констант и глобальных переменных, структур, определения используемых типов и т.п. Файл реализации содержит определения глобальных переменных (т.е. их описания без слова *extern*), определения статических переменных, которые не экспортируются за пределы данного файла (т.е. описания со словом *static*), реализацию глобальных функций, а также описания прототипов и реализацию вспомогательных функций, которые не экспортируются за пределы файла.

Технология тестирования и отладки отдельного модуля предполагает использование *заглушек* вместо функций из других модулей, вызываемых функциями данного модуля, в том случае, когда другие модули еще не реализованы.

Существуют различные подходы к разбиению проекта на модули. Наиболее популярна *технология сверху вниз*, когда основной модуль реализуется на первом шаге на основе нескольких вспомогательных, интерфейс которых формулируется по мере реализации основного модуля. На следующем шаге реализуются вспомогательные модули, для этого придумываются новые вспомогательные модули более низкого уровня и т.д., пока не дойдем до базовых исполнителей.

Удобнее всего разрабатывать большие проекты в объектно-ориентированных языках (C++, Java, C#, Visual Basic и др.). В них имеются понятия *класса* и *пространства имен*, использование которых значительно облегчает создание больших проектов. Класс — это, говоря упрощенно, набор функций, которые работают над общими данными. Функции называются *методами класса*, а общие данные — *членами класса*. Объектно-ориентированный язык позволяет создавать *объекты класса*, т.е. однотипные наборы данных, соответствующие описанию класса. Пространством имен в C++ или

пакетом в Java называется набор классов и функций, логически связанных между собой, реализуемых и используемых совместно. Отдельные классы или пространства имен соответствуют модулям, на которые разбивается проект.

Отметим, что, даже не используя объектно-ориентированного языка, можно придерживаться объектно-ориентированного стиля программирования, т.е. выделять группы функций и общие глобальные или статические данные, над которыми эти функции работают. Такие группы, подобно классам, реализуются и отлаживаются как отдельные структурные единицы.

Пример небольшого проекта «Стековый калькулятор» будет рассмотрен в следующей главе, посвященной структурам данных.

3.9.6. Задачи по теме «Технология программирования на Си»

Работа с матрицами

1. Найти индексы максимального элемента среди минимальных элементов строк матрицы.

2. Перемножить две прямоугольные матрицы.

3. Решить систему линейных уравнений, применяя метод Гаусса к расширенной матрице линейной системы.

4. Вычислить обратную матрицу для заданной невырожденной квадратной матрицы. Алгоритм аналогичен методу Гаусса и использует приведение исходной матрицы элементарными преобразованиями первого и второго рода сначала к *треугольной* матрице, затем к *диагональной*; после этого диагональная матрица приводится к *единичной* умножениями строк матрицы на ненулевые коэффициенты. При этом все преобразования параллельно применяются ко второй, изначально единичной, матрице. Как только исходная матрица будет приведена к единичной, вторая матрица будет равна обратной матрице к исходной.

Работа с файлами

1. В файле записана последовательность вещественных чисел. Найти

   - сумму чисел;
   - среднее арифметическое;
   - максимальный и минимальный элементы;
   - число максимальных элементов;
   - число перемен знака в последовательности;
   - последовательность задает коэффициенты многочлена *по убыванию* степеней. Найти значение многочлена в заданной точке по схеме Горнера;
   - та же задача для производной многочлена;
   - те же задачи в случае последовательности коэффициентов многочлена *по возрастанию* степеней.

   Во всех перечисленных задачах использовать схему вычисления индуктивной функции или построение индуктивного расширения, см. раздел 1.5.1.

2. Реализовать программу *split*, разрезающую файл на куски заданного размера. Программа должна ввести с клавиатуры имя файла и размер куска в байтах. В результате работы должны быть созданы файлы с именами "*filename*.001", "*filename*.002", "*filename*.003", ..., где "*filename*" — имя исходного файла, а файл "*filename.n*" содержит фрагмент исходного файла с номером n, записанным с помощью трех цифр.

3. Реализовать программу *merge*, объединяющую несколько файлов в один. Она должна быть обратной к программе *split* из предыдущей задачи. Программа должна ввести с клавиатуры имя файла "*filename*" и затем объединить файлы с именами "*filename*.001", "*filename*.002", "*filename*.003", ... в один файл с именем "*filename*".

Работа с текстами и файлами

1. Отформатировать абзацы в текстовом файле, сделав длину строк текста по возможности не превышающей 60 символов. Абзацы ограничиваются пустыми строками или красной строкой (т.е. отступом в первой строке абзаца). Строки можно разрезать по пробелам между словами, слово — это любая связная последовательность непробелов.

2. Удалить комментарии в стиле C++, начинающиеся с пары символов //, из исходного текста программы на Си или C++. Программа должна распознавать пары символов // внутри строковых констант `"..."`, которые не являются комментариями.

3. Написать программу, перекодирующую текст с русскими буквами, из кодировки Windows CP-1251 в кодировку КОИ-8 и обратно. Программа должна вводить имя файла и его кодировку и перекодировать тот же самый файл, открывая его одновременно на чтение и запись

   ```
   FILE *f = fopen(fileName, "rb+");
   ```

   следует использовать функции бинарного чтения и записи *fread*, *fwrite*, а также позиционирование в файле *fseek*. Для перекодировки в программе задаются две строковые константы: в одной перечисляются все буквы русского алфавита в кодировке КОИ-8, в другой в том же порядке — в кодировке Windows.

4. Модифицировать проект “Записная книжка” (см. с. 223): добавить хранение для каждого имени пары телефонов — домашнего и мобильного.

5. Реализовать программу поиска слова в тексте: ввести с клавиатуры имя текстового файла и произвольное слово, напечатать все строки файла, содержащие указанное слово.

6. Заменить в текстовом файле все вхождения заданного слова на другое слово. Имя файла и пара слов должны вводиться с клавиатуры.

7. Проверить сбалансированность круглых, квадратных и фигурных скобок в текстовом файле. Программа должна напечатать либо сообщение о том, что баланс скобок правильный, либо сообщение о том, что баланс скобок нарушен. В последнем случае программа должна указать тип скобки и либо напечатать строку с неправильной закрывающей скобкой и ее порядковый номер, либо сообщение о том, что количество открывающих скобок больше, чем закрывающих (это выясняется только после прочтения всего файла).

Глава 4

Структуры данных

«Алгоритмы + структуры данных = программы». Это — название книги Никлауса Вирта, знаменитого швейцарского специалиста по программированию, автора языков Паскаль, Модула-2, Оберон. С именем Вирта связано развитие структурного подхода к программированию. Н.Вирт известен также как блестящий педагог и автор классических учебников.

Обе составляющие программы, выделенные Н.Виртом, в равной степени важны. Не только несовершенный алгоритм, но и неудачная организация работы с данными может привести к замедлению работы программы в десятки, а иногда и в миллионы раз. С другой стороны, владение теорией прораммирования и умение системтически применять ее на практике позволяет быстро разрабатывать эффективные и в то же время эстетически красивые программы.

4.1. Общее понятие структуры данных

Структура данных — это исполнитель, который организует работу с данными, включая их хранение, добавление и удаление, модификацию, поиск и т.д. Структура данных поддерживает определенный порядок доступа к ним. Структуру данных можно рассматривать как своего рода склад или библиотеку. При описании структуры данных нужно перечислить набор действий, которые возможны для нее, и четко описать результат каждого действия. Будем называть такие действия *предписаниями*. С программной точки зрения, системе

предписаний структуры данных соответствует набор функций, которые работают над общими переменными.

Структуры данных удобнее всего реализовывать в объектно-ориентированных языках. В них структуре данных соответствует класс, сами данные хранятся в переменных-членах класса (или доступ к данным осуществляется через переменные-члены), системе предписаний соответствует набор методов класса. Как правило, в объектно-ориентированных языках структуры данных реализуются в виде библиотеки стандартных классов: это так называемые контейнерные классы языка C++, входящие в стандартную библиотеку классов STL, или классы, реализующие различные структуры данных из библиотеки Java Developer Kit языка Java.

Тем не менее, структуры данных столь же успешно можно реализовывать и в традиционных языках программирования, таких как Фортран или Си. При этом следует придерживаться объектно-ориентированного стиля программирования: четко выделить набор функций, которые осуществляют работу со структурой данных, и ограничить доступ к данным только этим набором функций. Сами данные реализуются как статические (не глобальные) переменные. При программировании на языке Си структуре данных соответствуют два файла с исходными текстами: 1) заголовочный, или h-файл, который описывает интерфейс структуры данных, т.е. набор прототипов функций, соответствующий системе предписаний структуры данных; 2) файл реализации, или Си-файл, в котором определяются статические переменные, осуществляющие хранение и доступ к данным, а также реализуются функции, соответствующие системе предписаний структуры данных.

Структура данных обычно реализуется на основе более простой *базовой структуры*, ранее уже реализованной, или на основе массива и набора простых переменных. Следует четко различать описание структуры данных с логической точки зрения и описание ее реализации. Различных реализаций может быть много, с логической же точки зрения (т.е. с точки зрения внешнего пользователя) все они эквивалентны и различаются, возможно, лишь скоростью выполнения предписаний.

4.2. Массив как базовая структура

Оперативная память с точки зрения программиста — это массив элементов. Любой элемент массива можно прочитать или записать сразу, за одно элементарное действие. Массив можно рассматривать как простейшую структуру данных. Структуры данных, в которых возможен непосредственный доступ к произвольным их элементам, называют структурами данных с *прямым*, или с *произвольным доступом* (по-английски random access). Наряду с массивом, структурой данных с прямым доступом является *множество*, которое будет рассмотрено ниже. В других структурах данных непосредственный доступ возможен лишь к одному или нескольким элементам, для доступа к остальным элементам надо выполнить дополнительные действия. Такие структуры данных называются структурами *последовательного доступа*. Примером структуры последовательного доступа является магнитофон, на которым записаны песни. В любой момент можно прослушать лишь очередную песню. Чтобы добраться до других музыкальных фрагментов, надо перемотать ленту вперед или назад. Кстати, такие магнитофоны, или накопители на магнитной ленте, очень долго использовались на ЭВМ, хотя сейчас уступили свое место более надежным и компактным системам (съемным магнитным и оптическим дискам, флэш-памяти и т.п.). Устройство компьютерного магнитофона было аналогично устройству обычного бытового магнитофона.

С логической точки зрения, массивом является также важнейшая составляющая компьютера — магнитный диск. Элементарной единицей чтения и записи для магнитного диска служит блок. Размер блока зависит от конструкции конкретного диска, обычно он кратен 512. За одну элементарную операцию можно прочесть или записать один блок с заданным адресом.

Итак, наиболее важные запоминающие устройства компьютера — оперативная память и магнитный диск — представляют собой массивы. Массив как бы дан программисту «свыше», так же как математику целые числа. Работа с элементами массива осуществляется исключительно быстро, все элементы массива доступны без всяких предварительных действий.

Тем не менее массивов недостаточно для написания эффективных

программ. Например, *поиск элемента* в массиве, если его элементы не упорядочены, невозможно реализовать эффективно: нельзя изобрести ничего лучшего, кроме последовательного перебора элементов. В случае упорядоченного хранения элементов можно использовать эффективный *бинарный поиск*, но затруднения возникают при добавлении или удалении элементов в середине массива и приводят к *массовым операциям*, т.е. операциям, время выполнения которых зависит от числа элементов структуры. От этих недостатков удается избавиться, реализуя множество элементов на базе *сбалансированных деревьев* или *хеш-функции*.

Есть и другие причины, по которым необходимо использовать более сложные, чем массивы, структуры данных. Логика многих задач требует организации определенного порядка доступа к данным. Например, в случае *очереди* элементы можно добавлять только в конец, а забирать только из начала очереди; в *стеке* доступны лишь элементы в вершине стека, в *списке* — элементы до и за указателем.

Наконец, массив имеет ограниченный размер. Увеличение размера массива в случае необходимости приводит к переписыванию его содержимого в захваченную область памяти большего размера, т.е. опять же к массовой операции. От этого недостатка свободны *ссылочные реализации* структур данных: реализации на основе линейных списков или на основе деревьев.

4.3. Реализация одних структур на базе других

Реализация структуры данных на основе базовой структуры — это описание ее работы в терминах базовой структуры. При этом считается, что базовая структура либо дана изначально, либо уже кем-то реализована. Реализация должна включать в себя описание *идеи реализации* (каким образом элементы реализуемой структуры хранятся в базовой структуре, какие дополнительные переменные используются) и набор подпрограмм, каждая из которых моделирует некоторое предписание реализуемой структуры при помощи предписаний базовой структуры.

При рассмотрении любой структуры данных необходимо сначала описать ее с логической точки зрения, а затем рассмотреть различные способы ее реализации. В качестве базы реализации в большин-

стве случаев выступает либо массив, либо динамическая память (т.е. память, в которой можно захватывать участки требуемого размера и освобождать ранее захваченные участки, когда они уже больше не нужны; см. раздел 3.7.3).

4.4. Простейшие структуры данных. Стек. Очередь

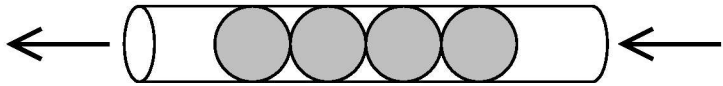
Наиболее важными из простейших структур данных являются стек и очередь. Эти структуры встречаются в программировании буквально на каждом шагу, в самых разнообразных ситуациях. Особенно интересен стек, который имеет самые неожиданные применения. В свое время при разработке серии ЭВМ IBM 360 в начале 70-х годов XX века фирма IBM совершила драматическую ошибку, не предусмотрев аппаратную реализацию стека. Эта серия содержала много других неудачных решений, но, к сожалению, была скопирована в Советском Союзе под названием ЕС ЭВМ (Единая Серия), а все собственные разработки были приостановлены. Это отбросило советскую промышленность на много лет назад в области разработки копьютеров.

4.4.1. Очередь

Очередь как структура данных понятна даже людям, не знакомым с программированием. Очередь содержит элементы, как бы выстроенные друг за другом в цепочку. У очереди есть начало и конец. Добавлять новые элементы можно только в конец очереди, забирать элементы можно только из начала. В отличие от обычной очереди, которую всегда можно при желании покинуть, из середины программистской очереди удалять элементы нельзя.

Очередь можно представить в виде трубки. В один конец трубки можно добавлять шарики — элементы очереди, из другого конца они извлекаются. Элементы в середине очереди, т.е. шарики внутри трубки, недоступны. Конец трубки, в который добавляются шарики, соответствует концу очереди, конец, из которого они извлекаются — началу очереди. Таким образом, концы трубки не симметричны, ша-

рики внутри трубки движутся только в одном направлении.



В принципе, можно было бы разрешить добавлять элементы в оба конца очереди и забирать их также из обоих концов. Такая структура данных в программировании тоже существует, ее название — "дек", от англ. Double Ended Queue, т.е. очередь с двумя концами. Дек применяется значительно реже, чем очередь.

Использование очереди в программировании почти соответствует ее роли в обычной жизни. Очередь практически всегда связана с обслуживанием запросов, в тех случаях, когда они не могут быть выполнены мгновенно. Очередь поддерживает также порядок обслуживания запросов. Рассмотрим, к примеру, что происходит, когда человек нажимает клавишу на клавиатуре компьютера. Тем самым человек просит компьютер выполнить некоторое действие. Например, если он просто печатает текст, то действие должно состоять в добавлении к тесту одного символа и может сопровождаться перерисовкой области экрана, прокруткой окна, переформатированием абзаца и т.п.

Любая, даже самая простая, операционная система всегда в той или иной степени многозадачна. Это значит, что в момент нажатия клавиши операционная система может быть занята какой-либо другой работой. Тем не менее, операционная система ни в какой ситуации не имеет права проигнорировать нажатие на клавишу. Поэтому происходит прерывание работы компьютера, он запоминает свое состояние и переключается на обработку нажатия на клавишу. Такая обработка должна быть очень короткой, чтобы не нарушить выполнение других задач. Команда, отдаваемая нажатием на клавишу, просто добавляется в конец очереди запросов, ждущих своего выполнения. После этого прерывание заканчивается, компьютер восстанавливает свое состояние и продолжает работу, которая была прервана нажатием на клавишу. Запрос, поставленный в очередь, будет выполнен не сразу, а только когда наступит его черед.

В системе Windows работа оконных приложений основана на сообщениях, которые посылаются этим приложениям. Например, бывают сообщения о нажатии на клавишу мыши, о закрытии окна, о

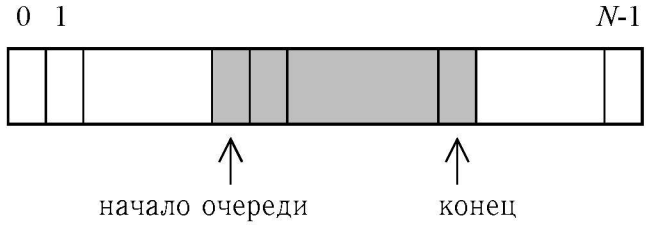
необходимости перерисовки области окна, о выборе пункта меню и т.п. Каждая программа имеет очередь запросов. Когда программа получает свой квант времени на выполнение, она выбирает очередной запрос из начала очереди и выполняет его. Таким образом, работа оконного приложения состоит, упрощенно говоря, в последовательном выполнении запросов из ее очереди. Очередь поддерживается операционной системой.

Подход к программированию, состоящий не в прямом вызове процедур, а в посылке сообщений, которые ставятся в очередь запросов, имеет много преимуществ и является одной из черт объектно-ориентированного программирования. Так, например, если оконной программе необходимо завершить работу по какой-либо причине, лучше не вызывать сразу команду завершения, которая опасна, потому что нарушает логику работы и может привести к потере данных. Вместо этого программа посылает самой себе сообщение о необходимости завершения работы, которое будет поставлено в очередь запросов и выполнено после запросов, поступивших ранее.

Реализация очереди на базе массива

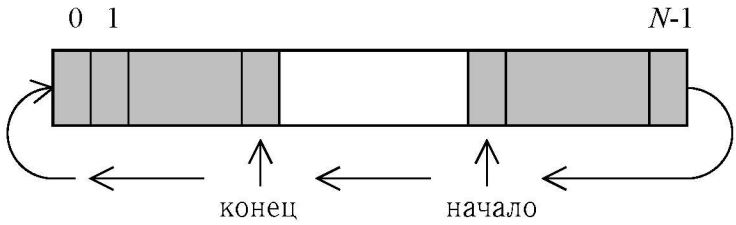
Как уже было сказано, программисту массив дан свыше, все остальные структуры данных нужно реализовывать на его основе. Конечно, такая реализация может быть многоэтапной, и не всегда массив выступает в качестве непосредственной базы реализации. В случае очереди наиболее популярны две реализации: непрерывная на базе массива, которую называют также реализацией на базе кольцевого буфера, и ссылочная реализация, или реализация на базе списка. Ссылочные реализации будут рассмотрены ниже.

При непрерывной реализации очереди в качестве базы выступает массив фиксированной длины N, таким образом, очередь ограничена и не может содержать более N элементов. Индексы элементов массива изменяются в пределах от 0 до $N-1$. Кроме массива, реализация очереди хранит три простые переменные: индекс начала очереди, индекс конца очереди, число элементов очереди. Элементы очереди содержатся в отрезке массива от индекса начала до индекса конца.



При добавлении нового элемента в конец очереди индекс конца сперва увеличивается на единицу, затем новый элемент записывается в ячейку массива с этим индексом. Аналогично, при извлечении элемента из начала очереди содержимое ячейки массива с индексом начала очереди запоминается в качестве результата операции, затем индекс начала очереди увеличивается на единицу. Как индекс начала очереди, так и индекс конца при работе двигаются слева направо. Что поисходит, когда индекс конца очереди достигает конца массива, т.е. $N - 1$?

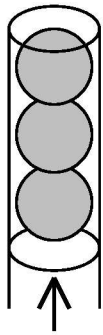
Ключевая идея реализации очереди состоит в том, что массов мысленно как бы зацикливается в кольцо. Считается, что за последним элементом массива следует его первый элемент (напомним, что последний элемент имеет индекс $N - 1$, а первый — индекс 0). При сдвиге индекса конца очереди вправо в случае, когда он указывает на последний элемент массива, он переходит на первый элемент. Таким образом, непрерывный отрезок массива, занимаемый элементами очереди, может переходить через конец массива на его начало.



4.4.2. Стек

Стек — самая популярная и, пожалуй, самая важная структура данных в программировании. Стек представляет собой запоминаю-

щее устройство, из которого элементы извлекаются в порядке, обратном их добавлению. Это как бы неправильная очередь, в которой первым обслуживают того, кто встал в нее последним. В программистской литературе общепринятыми являются аббревиатуры, обозначающие дисциплину работы очереди и стека. Дисциплина работы очереди обозначается FIFO, что означает «первым пришел — первым уйдешь» (First In First Out). Дисциплина работы стека обозначается LIFO, «последним пришел — первым уйдешь» (Last In First Out).



Стек можно представить в виде трубки с подпружиненным дном, расположенной вертикально. Верхний конец трубки открыт, в него можно добавлять, или, как говорят, заталкивать элементы. Общепринятые английские термины в этом плане очень красочны, операция добавления элемента в стек обозначается push, в переводе “затолкнуть, запихнуть”. Новый добавляемый элемент проталкивает элементы, помещенные в стек ранее, на одну позицию вниз. При извлечении элементов из стека они как бы выталкиваются вверх, по-английски pop (“выстреливают”).

Примером стека может служить стог сена, стопка бумаг на столе, стопка тарелок и т.п. Отсюда произошло название стека, что по-английски означает «стопка». Тарелки снимаются со стопки в порядке, обратном их добавлению. Доступна только верхняя тарелка, т.е. тарелка *на вершине стека*. Хорошим примером будет также служить железнодорожный тупик, в который можно составлять вагоны.

Использование стека в программировании

Стек применяется довольно часто, причем в самых разных ситуациях. Объединяет их следующая цель: нужно сохранить некоторую работу, которая еще не выполнена до конца, при необходимости переключения на другую задачу. Стек используется для временного сохранения состояния не выполненного до конца задания. После сохранения состояния компьютер переключается на другую задачу. По окончании ее выполнения состояние отложенного задания восстанавливается из стека, и копьютер продолжает прерванную работу.

Почему именно стек используется для сохранения состояния прерванного задания? Предположим, что компьютер выполняет задачу

A. В процессе ее выполнения возникает необходимость выполнить задачу B. Состояние задачи A запоминается, и компьютер переходит к выполнению задачи B. Но ведь и при выполнении задачи B компьютер может переключиться на другую задачу C, и нужно будет сохранить состояние задачи B, прежде чем перейти к C. Позже, по окончании C будет сперва восстановлено состояние задачи B, затем, по окончании B, — состояние задачи A. Таким образом, восстановление происходит в порядке, обратном сохранению, что соответствует дисциплине работы стека.

Стек позволяет организовать рекурсию, т.е. обращение подпрограммы к самой себе либо непосредственно, либо через цепочку других вызовов. Пусть, например, подпрограмма A выполняет алгоритм, зависящий от входного параметра X и, возможно, от состояния глобальных данных. Для самых простых значений X алгоритм реализуется непосредственно. В случае более сложных значений X алгоритм реализуется как сведение к применению того же алгоритма для более простых значений X. При этом подпрограмма A обращается сама к себе, передавая в качестве параметра более простое значение X. При таком обращении предыдущее значение параметра X, а также все локальные переменные подпрограммы A сохраняются в стеке. Далее создается новый набор локальных переменных и переменная, содержащая новое (более простое) значение параметра X. Вызванная подпрограмма A работает с новым набором переменных, не разрушая предыдущего набора. По окончании вызова старый набор локальных переменных и старое состояние входного параметра X восстанавливаются из стека, и подпрограмма продолжает работу с того места, где она была прервана.

На самом деле даже не приходится специальным образом сохранять значения локальных переменных подпрограммы в стеке. Дело в том, что локальные переменные подпрограммы (т.е. ее внутренние, рабочие переменные, которые создаются в начале ее выполнения и уничтожаются в конце) размещаются в стеке, реализованном аппаратно на базе обычной оперативной памяти. В самом начале работы подпрограмма захватывает место в стеке под свои локальные переменные, этот участок памяти в аппаратном стеке называют обычно *блок локальных переменных* или по-английски *frame* (“кадр”). В момент окончания работы подпрограмма освобождает память, удаляя из

стека блок своих локальных переменных.

Кроме локальных переменных, в аппаратном стеке сохраняются адреса возврата при вызовах подпрограмм. Пусть в некоторой точке программы A вызывается подпрограмма B. Перед вызовом подпрограммы B адрес инструкции, следующей за инструкцией вызова B, сохраняется в стеке. Это так называемый *адрес возврата* в программу A. По окончании работы подпрограмма B извлекает из стека адрес возврата в программу A и возвращает управление по этому адресу. Таким образом, компьютер продолжает выполнение программы A, начиная с инструкции, следующей за инструкцией вызова. В большинстве процессоров имеются специальные команды, поддерживающие вызов подпрограммы с предварительным помещением адреса возврата в стек и возврат из подпрограммы по адресу, извлекаемому из стека. Обычно команда вызова называется call, команда возврата — return.

В стек помещаются также параметры подпрограммы или функции перед ее вызовом. Порядок их помещения в стек зависит от соглашений, принятых в языках высокого уровня. Так, в языке Си или C++ на вершине стека лежит первый аргумент функции, под ним второй и так далее. В Паскале все наоборот, на вершине стека лежит последний аргумент функции. (Поэтому, кстати, в Си возможны функции с переменным числом аргументов, такие, как printf, а в Паскале нет.)

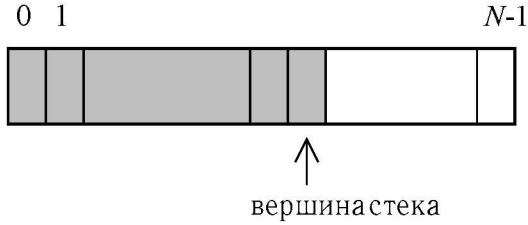
В Фортране-4, одном из самых старых и самых удачных языков программирования, аргументы передаются через специальную область памяти, которая может располагаться не в стеке, поскольку до конца 70-х годов XX века еще существовали компьютеры вроде IBM 360 или ЕС ЭВМ без аппаратной реализации стека. Адреса возврата также сохранялись не в стеке, а в фиксированных для каждой подпрограммы ячейках памяти. Программисты называют такую память статической в том смысле, что статические переменные занимают всегда одно и то же место в памяти в любой момент работы программы. При использовании только статической памяти рекурсия невозможна, поскольку при новом вызове предыдущие значения локальных переменных разрушаются. В эталонном Фортране-4 использовались только статические переменные, а рекурсия была запрещена. До сих пор язык Фортран широко используется в научных

и инженерных расчетах, однако, современный стандарт Фортрана-90 уже вводит стековую память, устраняя недостатки ранних версий языка.

Реализация стека на базе массива

Реализация стека на базе массива является классикой программирования. Иногда даже само понятие стека не вполне корректно отождествляется с этой реализацией.

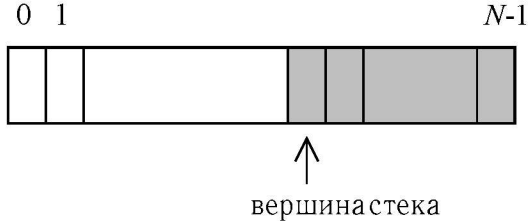
Базой реализации является массив размера N, таким образом, реализуется стек ограниченного размера, максимальная глубина которого не может превышать N. Индексы ячеек массива изменяются от 0 до $N-1$. Элементы стека хранятся в массиве следующим образом: элемент на дне стека располагается в начале массива, т.е. в ячейке с индексом 0. Элемент, расположенный над самым нижним элементом стека, хранится в ячейке с индексом 1, и так далее. Вершина стека хранится где-то в середине массива. Индекс элемента на вершине стека хранится в специальной переменной, которую обычно называют *указателем стека* (по-английски Stack Pointer или просто SP).



Когда стек пуст, указатель стека содержит значение «минус единица». При добавлении элемента указатель стека сначала увеличивается на единицу, затем в ячейку массива с индексом, содержащимся в указателе стека, записывается добавляемый элемент. При извлечении элемента из стека сперва содержимое ячейки массива с индексом, содержащимся в указателе стека, запоминается во временной переменной в качестве результата операции, затем указатель стека уменьшается на единицу.

В приведенной реализации стек растет в сторону увеличения индексов ячеек массива. Часто используется другой вариант реализа-

ции стека на базе вектора, когда дно стека помещается в последнюю ячейку массива, т.е. в ячейку с индексом $N-1$. Элементы стека занимают непрерывный отрезок массива, начиная с ячейки, индекс которой хранится в указателе стека, и заканчивая последней ячейкой массива. В этом варианте стек растет в сторону уменьшения индексов. Если стек пуст, то указатель стека содержит значение N (которое на единицу больше, чем индекс последней ячейки массива).



Реализация стека на языке Си

Реализуем стек вещественных чисел. Ниже мы используем эту реализацию как составную часть проекта «Стековый калькулятор» (см. раздел 4.4.2).

Реализация включает два файла: “streal.h”, в котором описывается интерфейс исполнителя “Стек”, и “streal.cpp”, реализующий функции работы со стеком. Слово real обозначает вещественное число.

Используется первый вариант реализации стека на базе массива, описанный в предыдущем разделе: стек растет в сторону увеличения индексов массива. Пространство под массив элементов стека захватывается в динамической памяти в момент инициализации стека. Функции инициализации *st_init* передается размер массива, т.е. максимально возможное число элементов в стеке. Для завершения работы стека нужно вызвать функцию *st_terminate*, которая освобождает захваченную в *st_init* память. Ниже приведено содержимое файла “streal.h”, описывающего интерфейс стека.

```
// Файл "streal.h"
// Стек вещественных чисел, интерфейс
//
#ifndef ST_REAL_H
#define ST_REAL_H
```

```
// Прототипы функций, реализующих предписания стека:

void st_init(int maxSize); // Начать работу (вх: цел
                           //     макс. размер стека)
void st_terminate();    // Закончить работу
void st_push(double x); // Добавить эл-т (вх: вещ x)
double st_pop();        // Взять элемент: вещ
double st_top();        // Вершина стека: вещ
int st_size();          // Текущий размер стека: цел
bool st_empty();        // Стек пуст? : лог
int st_maxSize();       // Макс. размер стека: цел
bool st_freeSpace();    // Есть свободное место? : лог
void st_clear();        // Удалить все элементы
double st_elementAt(int i); // Элемент стека на
                            //   глубине (вх: i): вещ
#endif
// Конец файла "streal.h"
```

Отметим, что директивы условной трансляции

```
#ifndef ST_REAL_H
#define ST_REAL_H
. . .
#endif
```

используются для предотвращения повторного включения h-файла: при первом включении файла определяется переменная препроцессора `ST_REAL_H`, а директива "`#ifndef ST_REAL_H`" подключает текст, только если эта переменная не определена. Такой трюк используется практически во всех h-файлах. Нужен он потому, что одни h-файлы могут подключать другие, и без этого механизма избежать повторного включения одного и того же файла трудно.

Файл "streal.cpp" описывает общие статические переменные, над которыми работают функции, соответствующие предписаниям стека, и реализует эти функции.

```
// Файл "streal.cpp"
// Стек вещественных чисел, реализация
```

```
//
#include <stdlib.h>
#include <assert.h>
#include "streal.h" // Подключить описания функций стека

// Общие переменные для функций, реализующих
// предписания стека:
static double *elements = 0; // Указатель на массив эл-тов
                             //     стека в дин. памяти
static int max_size = 0;     // Размер массива
static int sp = (-1);        // Индекс вершины стека

// Предписания стека:

void st_init(int maxSize) { // Начать работу (вх:
                            //     макс. размер стека)
    assert(elements == 0);
    max_size = maxSize;
    elements = (double *) malloc(
        max_size * sizeof(double)
    );
    sp = (-1);
}

void st_terminate() { // Закончить работу
    if (elements != 0) {
        free(elements);
    }
}

void st_push(double x) { // Добавить эл-т (вх: вещ x)
    assert(              // утв:
        elements != 0 && //   стек начал работу и
        sp < max_size-1  //   есть своб. место
    );
    ++sp;
    elements[sp] = x;
```

```
}

double st_pop() { // Взять элемент: вещ
    assert(sp >= 0); // утв: стек не пуст
    --sp;            // элемент удаляется из стека
    return elements[sp + 1];
}

double st_top() { // Вершина стека: вещ
    assert(sp >= 0); // утв: стек не пуст
    return elements[sp];
}

int st_size() { // Текущий размер стека: цел
    return (sp + 1);
}

bool st_empty() { // Стек пуст? : лог
    return (sp < 0);
}

int st_maxSize() { // Макс. размер стека: цел
    return max_size;
}

bool st_freeSpace() { // Есть своб. место? : лог
    return (sp < max_size - 1);
}

void st_clear() { // Удалить все элементы
    sp = (-1);
}

double st_elementAt(int i) { // Элемент стека на
                             //   глубине (вх: i): вещ
    assert(                     // утв:
        elements != 0 &&        // стек начал работу и
```

```
        0 <= i && i < st_size() // 0 <= i < размер стека
    );
    return elements[sp - i];
}
// Конец файла "streal.cpp"
```

Использование функции assert для проверки утверждений и ситуация «отказ»

В реализации стека на Си неоднократно использовалась функция *assert*, в переводе с английского "утверждение". Фактическим аргументом функции является логическое выражение. Если оно истинно, то ничего не происходит; если ложно, то программа завершается аварийно, выдавая диагностику ошибки.

Функция *assert* является реализаций конструкции "*утверждение*", использование которой преследует две цели:

1) программист в процессе написания программы может явно сформулировать утверждение, которое, по его мнению, должно выполняться в данной точке программы. В этом случае конструкция "утверждение" выполняет роль комментария, облегчая создание и понимание программы;

2) компьютер при выполнении программы проверяет все явно сформулированные утверждения. Истинность утверждения соответствует предположениям программиста, сделанным в процессе написания программы, поэтому выполнение программы продолжается. Ложность утверждения свидетельствует об ошибке программиста. При этом выдается сообщение об ошибке и выполнение программы немедленно прекращается. Таким образом, конструкция "утверждение" позволяет компьютеру проверять корректность программы непосредственно в процессе ее выполнения.

Ситуация, когда программа аварийно завершается из-за того, что утверждение не выполняется, называется *отказом*.

Многие предписания структуры данных выполнимы не во всех ситуациях. Например, элемент можно взять из стека только в случае,

когда стек не пуст. Поэтому перед началом выполнения предписания "Взять элемент" проверяется условие "стек не пуст":

```
double st_pop() { // Взять элемент: вещ
    assert(sp >= 0); // утв: стек не пуст
    . . .
```

Если это утверждение ложно, то возникает ситуация "отказ": комьютер завершает работу, выдавая диагностику ошибки. Невыполнение утверждения всегда свидетельствует об ошибке программиста: программа должна быть написана таким образом, чтобы некорректные исходные данные не приводили к отказу.

Использование конструкции "утверждение" — мощное средство отладки программы. Важность ситуации "отказ" была осознана в процессе эволюции программирования далеко не сразу. На заре развития программирования на первом плане были требования эффективности программы — ее компактности и быстродействия. И лишь по мере проникновения компьютеров во все области жизни на первый план выступила надежность программы. В современном мире от надежности программы во многих случаях зависит человеческая жизнь, поэтому любые способы повышения надежности очень важны.

Почему в случае невыполнения утверждения возникает ситуация "отказ"? Альтернативой могло бы быть игнорирование некорректной ситуации и то или иное продолжение программы: например, при попытке извлечения элемента из пустого стека выдавался бы ноль. На самом деле это худшее решение из всех, которые только можно придумать! Любую ошибку надо диагностировать и исправлять как можно раньше, а имитация полезной деятельности в некорректной ситуации крайне опасна. Это может привести, например, к тому, что программа, управляющая посадкой самолетов, будет успешно сажать в среднем 999 самолетов из 1000, а каждый тысячный будет разбиваться.

Практическая рекомендация: используйте конструкцию "утверждение" как можно чаще. Если при разработке программы вы считаете, что в данной точке должно выполняться некоторое утверждение, сформулируйте его явно, чтобы компьютер при выполнении программы мог бы его проверить. Таким способом находится и исправляется львиная доля ошибок.

При частом использовании конструкции "утверждение" возникает проблема с уменьшением скорости выполнения программы. Она решена в языках Си и C++ следующим образом: на самом деле конструкция *assert* в Си — это не функция, а макроопределение, которое обрабатывается препроцессором. Текст Си-программы может транслироваться в двух режимах: отладочном и нормальном. В нормальном режиме в соответствии со стандартом ANSI определена переменная *NDEBUG* препроцессора. Для определения макрокоманды *assert* используется условная трансляция: если переменная *NDEBUG* определена, то *assert* определяется как пустое выражение; если нет, то *assert* определяется как проверка условия и вызов функции *_assert* (к имени добавлен символ подчеркивания) в случае, когда условие ложно. Функция *_assert* печатает диагностику ошибки и завершает выполнение программы, т.е. реализует ситуацию "отказ".

Таким образом, условие реально проверяется лишь при трансляции программы в отладочном режиме. Благодаря этому быстродействие программы не снижается, однако, в нормальном режиме условие не проверяется. Примерно как при обучении плаванию: в бассейне человек надевает спасательный круг, но снимает его, когда начинает плавать в океане.

Стековый калькулятор и обратная польская запись формулы

В 1920 г. польский математик Ян Лукашевич предложил способ записи арифметических формул, не использующий скобок. В привычной нам записи знак операции записывается между аргументами, например, сумма чисел 2 и 3 записывается как $2+3$. Ян Лукашевич предложил две другие формы записи: префиксная форма, в которой знак операции записывается перед аргументами, и постфиксная форма, в которой знак операции записывается после аргументов. В префиксной форме сумма чисел 2 и 3 записывается как + 2 3, в постфиксной — как 2 3 +. В честь Яна Лукашевича эти формы записи называют прямой и обратной польской записью.

В польской записи скобки не нужны. Например, выражение

```
(2+3)*(15-7)
```

записывается в прямой польской записи как

```
* + 2 3 - 15 7,
```

в обратной польской записи — как

```
2 3 + 15 7 - *.
```

Если прямая польская запись не получила большого распространения, то обратная оказалась чрезвычайно полезной. Неформально преимущество обратной записи перед прямой польской записью или обычной записью можно объяснить тем, что гораздо удобнее выполнять некоторое действие, когда объекты, над которыми оно должно быть совершено, уже даны.

Обратная польская запись формулы позволяет вычислять выражение любой сложности, используя стек как запоминающее устройство для хранения промежуточных результатов. Такой стековый калькулятор был впервые выпущен фирмой Hewlett Packard. Обычные модели калькуляторов не позволяют вычислять сложные формулы без использования бумаги и ручки для записи промежуточных результатов. В некоторых моделях есть скобки с одним или двумя уровнями вложенности, но более сложные выражения вычислять невозможно. Также в обычных калькуляторах трудно понять, как результат и аргументы перемещаются в процессе ввода и вычисления между регистрами калькулятора. Калькулятор обычно имеет регистры X, Y и регистр памяти, промежуточные результаты каким-то образом перемещаются по регистрам, каким именно — запомнить невозможно.

В отличие от других калькуляторов, устройство стекового калькулятора вполне понятно и легко запоминается. Калькулятор имеет память в виде стека. При вводе числа оно просто добавляется в стек. При нажатии на клавишу операции, например, на клавишу +, аргументы операции сначала извлекаются из стека, затем с ними выполняется операция, наконец, результат операции помещается обратно в стек. Таким образом, при выполнении операции с двумя аргументами, например, сложения, в стеке должно быть не менее двух чисел. Аргументы удаляются из стека и на их место кладется результат, то есть при выполнении сложения глубина стека уменьшается на единицу. Вершина стека всегда содержит результат последней операции и высвечивается на дисплее калькулятора.

Для вычисления выражения надо сначала преобразовать его в обратную польскую запись (при некотором навыке это легко сделать в уме). В приведенном выше примере выражение `(2+3)*(15-7)` преобразуется к

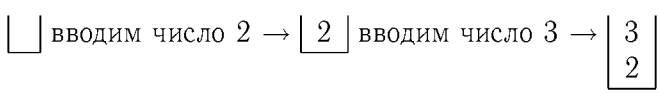
```
2 3 + 15 7 - *
```

Затем обратная польская запись просматривается последовательно слева направо. Если мы видим число, то просто вводим его в калькулятор, т.е. добавляем его в стек. Если мы видим знак операции, то нажимаем соответствующую клавишу калькулятора, выполняя таким образом операцию с числами на вершине стека.

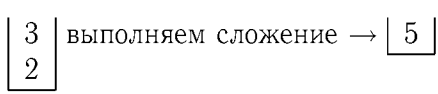
Изобразим последовательные состояния стека калькулятора при вычислении по приведенной формуле. Сканируем слева направо ее обратную польскую запись:

```
2 3 + 15 7 - *
```

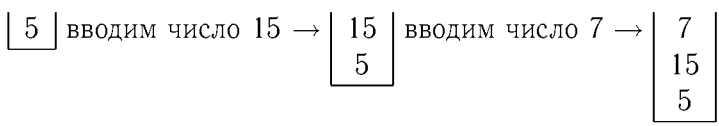
Стек вначале пуст. Последовательно добавляем числа 2 и 3 в стек.



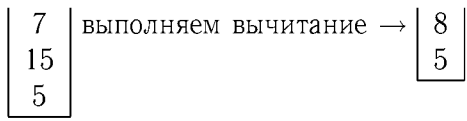
Далее читаем символ + и нажимаем на клавишу + калькулятора. Числа 2 и 3 извлекаются из стека, складываются, и результат помещается обратно в стек.



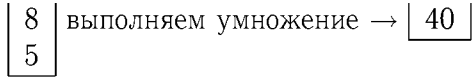
Далее, в стек добавляются числа 15 и 7.



Читаем символ — и нажимаем на клавишу — калькулятора. Со стека при этом снимаются два верхних числа 7 и 15 и выполняется операция вычитания. Причем уменьшаемым является то число, которое было введено раньше, а вычитаемым — число, введенное позже. Иначе говоря, при выполнении некоммутативных операций, таких как вычитание или деление, правым аргументом является число на вершине стека, левым — число, находящееся под вершиной стека.



Наконец, читаем символ * и нажимаем на клавишу * калькулятора. Калькулятор выполняет умножение, со стека снимаются два числа, перемножаются, результат помещается обратно в стек.



Число 40 является результатом вычисления выражения. Оно находится на вершине стека и высвечивается на дисплее стекового калькулятора.

Реализация стекового калькулятора на Си

Рассмотрим небольшой проект, реализующий стековый калькулятор на Си. Такая программа весьма полезна, поскольку позволяет проводить вычисления, не прибегая к записи промежуточных результатов на бумаге.

Программа состоит из трех файлов: “streal.h”, “streal.cpp” и “stcalc.cpp”. Первые два файла реализуют стек вещественных чисел, эта реализация уже рассматривалась ранее (см. с. 258). Файл “stcalc.cpp” реализует стековый калькулятор на базе стека. Для сборки программы следует объединить все три файла в один проект. Команды построения программы зависят от операционной системы и компилятора, например, в системе Unix с компилятором “gcc” программу можно собрать с помощью команды

```
g++ -o stcalc -lm stcalc.cpp streal.cpp
```

в результате которой создается файл “stcalc” с программой, готовой к выполнению.

Ниже приведено содержимое файла “stcalc.cpp”. Функция *main*, описанная в этом файле, организует диалог с пользователем в режиме команда–ответ. Пользователь может ввести число с клавиатуры, это число просто добавляется в стек. При вводе одного из четырех знаков арифметических операций +, −, *, / программа извлекает из стека два числа, выполняет указанное арифметическое действие над ними и помещает результат обратно в стек. Значение результата отображается также на дисплее. Кроме арифметических операций, пользователь может ввести название одной из стандартных функций: sin, cos, exp, log (натуральный логарифм). При этом программа

извлекает из стека аргумент функции, вычисляет значение функции и помещает его обратно в стек. При желании список стандартных функций и возможных операций можно расширить. Наконец, можно выполнять еще несколько команд:

pop	удалить вершину стека;
clear	очистить стек;
=	напечатать вершину стека;
show	напечатать содержимое стека;
clear	очистить стек;
help	напечатать подсказку;
quit	завершить работу программы.

Каждая команда стекового калькулятора реализуется с помощью отдельной функции. Например, вычитание реализуется с помощью функции onSub():

```
static void onSub() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
    st_push(x - y);
    display();
}
```

В начале функции проверяется, что глубина стека не меньше двух. В противном случае, выдается сообщение об ошибке, и функция завершается. Далее из стека извлекаются операнды y и x операции вычитания. Элементы извлекаются из стека в порядке, обратном их помещению в стек, поэтому y извлекается раньше, чем x. Затем вычисляется разность $y - x$, ее значение помещается обратно в стек и печатается на дисплее, для печати вершины стека вызывается функция *display*.

Приведем полный текст программы.

```
// Файл "stcalc.cpp"
// Реализация стекового калькулятора на базе стека
```

```
//
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "streal.h" // Интерфейс исполнителя "стек"

// Прототипы функций, реализующих команды калькулятора:
// Арифметические операции
static void onAdd();
static void onSub();
static void onMul();
static void onDiv();

// Добавить число в стек(вх: текстовая запись числа)
static void onPush(const char* line);

// Вычисление математических функций
static void onSin();     // sin
static void onCos();     // cos
static void onExp();     // Экспонента
static void onLog();     // Натуральный логарифм
static void onSqrt();    // Квадратный корень

// Другие команды
static void onPop();     // Удалить вершину стека
static void onClear();   // Очистить стек
static void display();   // Напечатать вершину стека
static void onShow();    // Напечатать содержимое стека
static void printHelp(); // Напечатать подсказку

int main() {
    char line[256]; // Буфер для ввода строки
    int linelen;    // Длина строки

    st_init(1024);  // Стек.начать работу(1024)
```

```
                //   1024 -- макс. глубина стека
printHelp();    // Напечатать подсказку

while (true) {          // Цикл до бесконечности
    scanf("%s", line);      // ввести строку
    linelen = strlen(line); // длина строки
    if (linelen == 0)
        continue;

    // Разобрать команду и вызвать реализующую
    // ее функцию
    if (strcmp(line, "+") == 0) {
        onAdd();
    } else if (strcmp(line, "-") == 0) {
        onSub();
    } else if (strcmp(line, "*") == 0) {
        onMul();
    } else if (strcmp(line, "/") == 0) {
        onDiv();
    } else if (strcmp(line, "sin") == 0) {
        onSin();
    } else if (strcmp(line, "cos") == 0) {
        onCos();
    } else if (strcmp(line, "exp") == 0) {
        onExp();
    } else if (strcmp(line, "log") == 0) {
        onLog();
    } else if (strcmp(line, "sqrt") == 0) {
        onSqrt();
    } else if (strcmp(line, "=") == 0) {
        display();
    } else if (         // Если это число
        isdigit(line[0]) || (
            linelen > 1 &&
            (line[0] == '-' || line[0] == '+') &&
            isdigit(line[1])
        )
```

```
        ) {
            onPush(line);   // Добавить число в стек
        } else if (strcmp(line, "pop") == 0) {
            onPop();
        } else if (strcmp(line, "clear") == 0) {
            onClear();
        } else if (strcmp(line, "show") == 0) {
            onShow();
        } else if (strcmp(line, "quit") == 0) {
            break;       // Завершить работу
        } else {         // Неправильная команда =>
            printHelp(); //     напечатать подсказку
        }
    }
    return 0;
}

static void onAdd() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
    st_push(x + y);
    display();
}

static void onSub() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
```

```
    st_push(x - y);
    display();
}

static void onMul() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
    st_push(x * y);
    display();
}

static void onDiv() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
    st_push(x / y);
    display();
}

static void onPush(const char* line) {
    double x = atof(line);
    st_push(x);
}

static void onSin() {
    double x;
    if (st_empty()) {
        printf("Stack empty.\n");
```

```
        return;
    }
    x = st_pop();
    st_push(sin(x));
    display();
}

static void onCos() {
    double x;
    if (st_empty()) {
        printf("Stack empty.\n");
        return;
    }
    x = st_pop();
    st_push(cos(x));
    display();
}

static void onExp() {
    double x;
    if (st_empty()) {
        printf("Stack empty.\n");
        return;
    }
    x = st_pop();
    st_push(exp(x));
    display();
}

static void onLog() {
    double x;
    if (st_empty()) {
        printf("Stack empty.\n");
        return;
    }
    x = st_pop();
    st_push(log(x));
```

```
    display();
}

static void onSqrt() {
    double x;
    if (st_empty()) {
        printf("Stack empty.\n");
        return;
    }
    if (st_top() < 0.0) {
        printf("Arg. of square root is negative.\n");
        return;
    }
    x = st_pop();
    st_push(sqrt(x));
    display();
}

static void onPop() {
    st_pop();
}

static void onClear() {
    st_clear();
}

static void display() {
    if (!st_empty()) {
        printf("=%lf\n", st_top());
    } else {
        printf("stack empty\n");
    }
}

static void onShow() {
    int d = st_size();
    printf("Depth of stack = %d.", d);
```

```
    if (d > 0)
        printf(" Stack elements:\n");
    else
        printf("\n");

    for (int i = 0; i < d; i++) {
        printf("  %lf\n", st_elementAt(i));
    }
}

static void printHelp() {
    printf(
        "Stack Calculator commands:\n"
        "    <number>    Push a number in stack\n"
        "    +, -, *, /  Ariphmetic operations\n"
        "    sin, cos,   Calculate a function\n"
        "    exp, log,   \n"
        "    sqrt        \n"
        "    =           Display the stack top\n"
        "    pop         Remove the stack top\n"
        "    show        Show the stack\n"
        "    clear       Clear the stack\n"
        "    quit        Terminate the program\n"
    );
}
// Конец файла "stcalc.cpp"
```

Пример работы программы “stcalc”. Пусть нужно вычислить выражение

$$\sqrt{3 \cdot 3 + 4 \cdot 4}$$

Запишем выражение, используя обратную польскую запись:

```
3, 3, *, 4, 4, *, sqrt
```

(через *sqrt* обозначается операция извлечения квадратного корня). Последовательно отдаем соответствующие команды стековому калькулятору. При работе программы stcalc получается следующий диалог:

```
Stack Calculator commands:
    <number>    Push a number in stack
    +, -, *, /  Ariphmetic operations
    sin, cos,   Calculate a function
    exp, log,
    sqrt
    =           Display the stack top
    pop         Remove the stack top
    show        Show the stack
    clear       Clear the stack
    quit        Terminate the program
3 3 *
=9.000000
4 4 *
=16.000000
+
=25.000000
sqrt
=5.000000
```

Обратная польская запись формул оказалась исключительно удобной при работе с компьютерами. Для вычислений используется стек, что позволяет работать с выражениями любой степени сложности. Реализация стекового вычислителя не представляет никакого труда. Имеется также простой алгоритм преобразования выражения из обычной записи, в которой знак операции указывается между аргументами, в ее обратную польскую запись. Все это привело к тому, что многие компиляторы языков высокого уровня используют обратную польскую запись в качестве внутренней формы представления программы. Рассмотрим, к примеру, язык программирования Java. Как всякий объектно-ориентированный язык, он является интерпретируемым, а не компилируемым языком. Это означает, что компилятор Java преобразует исходную Java-программу не в машинные коды, а в промежуточный язык, предназначенный для выполнения (интерпретации) на специальной Java-машине. В случае Java этот промежуточный язык называют байткодом. Компилятор Java помещает байткод в файл с расширением ".class". Байткод представляет

собой, упрощенно говоря, обратную польскую запись Java-прогаммы, а Java-машина — стековый вычислитель.

Язык PostScript

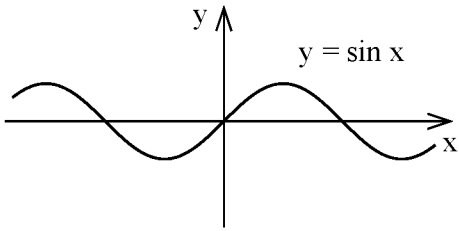
Другой яркий пример использования обратной польской записи — это графический язык PostScript. Он предназначен для печати текстов высокого качества на лазерных принтерах; он является стандартом представления текстов типографского качества, не зависящим от конкретной модели принтера.

То, что PostScript — язык программирования, для многих людей, знакомых с типографским делом, но далеких от программирования, звучит непривычно. Общепринятое мнение, что компьютер работает с числами и в основном что-то вычисляет, не вполне верно. Не менее часто компьютерная программа работает с текстами и с изображениями. Текст, содержащийся в обычном текстовом файле, можно рассматривать с двух точек зрения. Можно трактовать его просто как текст статьи или книги. Рассмотрим, однако, процесс печати текста на обычном (не графическом) принтере. Принтер соединен с компьютером кабелем, и компьютер просто посылает через этот кабель один за другим символы, составляющие текст. В этом случае букву A, входящую в текст, следует рассматривать как команду, предписывающую принтеру напечатать символ A в текущей точке страницы, используя текущий шрифт. После этого коодинату x текущей точки надо увеличить на ширину буквы A. С этой точки зрения весь текст можно трактовать как программу его собственной печати. В случае обычного текстового файла эта программа весьма примитивна, в ней, к примеру, нет команд смены шрифтов, изменения текущей позиции, рисования линий и т.д. Понятно, что текст типографского качества не может быть представлен обычным текстовым файлом.

В случае использования языка PostScript файл, пересылаемый на PostScript-принтер, представляет собой программу печати текста. Язык PostScript имеет огромное количество возможностей, и вряд ли найдется много людей, владеющих им. Чаще всего PostScript–программа создается другой программой обработки текста. Например, PostScript–файл создается TEX-ом для печати на принтере. (TEX — это язык записи текстов, содержащих математические формулы, созданный замечательным математиком и теоретиком програм-

мирования Дональдом Кнутом, см. книгу [2]. Фактически TEX представляет собой язык программирования. Данная книга подготовлена в TEX'е с использованием макропакета LATEX 2_ε.) Текстовые процессоры, такие, как Adobe Acrobat или MS Word, также в случае печати на профессиональном PostScript–принтере преобразуют текст в PostScript–программу. (Более точно, такое преобразование осуществляется драйверами операционной системы.) PostScript–файлы очень удобны для распространения: поскольку это файлы в обычном текстовом формате, они будут напечатаны одинаково в любой стране независимо от национальных кодировок, операционных систем, наличия шрифтов и т.п.

PostScript–программа представляет собой обратную польскую запись в том смысле, что всякая команда записывается после своих аргументов. При выполнении PostScript–программы используется стек. Рассмотрим для примера несложную программу, рисующую график функции $y = sin(x)$. Вот какая картинка рисуется в результате выполнения этой программы:



(Подчеркнем особо, что все рисунки, содержащиеся в данном пособии, реализованы в виде PostScript–программ, написанных вручную без использования каких-либо графических редакторов.)

Ниже приведен полный текст PostScript–программы, рисующей график функции. Отметим сразу, что символ процента % используется в языке PostScript в качестве комментария:

```
% Файл "func.ps"
% Рисование графика функции y = f(x)

% Перейти от пунктов (1/72 дюйма) к миллиметрам
2.83 2.83 scale
```

```
0.2 setlinewidth  % Установить толщину линии

% Нарисовать координатную ось X:
1 15 moveto   % переместиться в точку (1, 15)
60 15 lineto  % провести линию к точке (60, 15)
              % Рисуем стрелку:
57 16 moveto  % переместиться в точку (57, 16)
60 15 lineto  % провести линию к точке (60, 15)
57 14 lineto  % провести линию к точке (57, 14)
stroke        % нарисовать построенные линии

% Нарисовать координатную ось Y:
30 1 moveto   % переместиться в точку (30, 1)
30 30 lineto  % провести линию к точке (30, 30)
              % Рисуем стрелку:
29 27 moveto  % переместиться в точку (29, 27)
30 30 lineto  % провести линию к точке (30, 30)
31 27 lineto  % провести линию к точке (31, 27)
stroke        % нарисовать построенные линии

% Определение функции
%   f(x) = 5 * sin((x - 30) * 0.2 * 180/Pi) + 15
% Дано: число x на вершине стека.
% Надо: заменить вершину стека на f(x).
/Func {
    30 sub 0.2 mul 57.296 mul sin 5 mul 15 add
} def

% Рисуем график функции

0.3 setlinewidth  % установить толщину линии
2 2 Func moveto   % переместиться в точку (2, f(2))
2.5 0.5 58 { % цикл для x от 2.5 до 58 шаг 0.5
    dup       %   удвоить вершину стека
    Func      %   заменить x в вершине стека на f(x)
    lineto    %   провести линию к точке (x, f(x))
} for         % конец цикла
```

```
stroke          % нарисовать построенные линии

/Times-Roman findfont % загрузить шрифт Times-Roman
4 scalefont           % установить размер шрифта 4 мм
setfont               % установить текущий шрифт

40 23 moveto          % переместиться в точку (40, 23)
(y = sin x) show      % напечатать текст "y = sin x"

% Надписи на осях координат
59 11 moveto          % переместиться в точку (59, 11)
(x) show              % напечатать текст "x"

26 28 moveto          % переместиться в точку (26, 28)
(y) show              % напечатать текст "y"

showpage              % напечатать страницу
```

Разберем подробно некоторые элементы этой программы. В первой выполняемой строке устанавливается миллиметровая шкала:

```
2.83 2.83 scale
```

По умолчанию в языке PostScript единицей измерения является один пункт, или 1/72 дюйма. Всякий программист помнит, что один дюйм равен 25.4 миллиметра. Вычислим отношение одного миллиметра к одному пункту:

```
1 mm / 1 pt = 1 mm / (1/72)" =
    = 1 mm / (25.4/72) mm = 2.834645
```

Таким образом, увеличивая масштаб в 2.83 раза, мы переходим от пунктов к миллиметрам. Для изменения масштаба мы помещаем в стек два числа 2.83, соответствующих изменению масштабов по x и y. Затем выполняется команда scale (изменить масштаб). Со стека при этом снимаются два числа, и масштабы по x и по y изменяются.

Разные команды могут иметь различное число аргументов. Например, вторая строка устанавливает толщину линии.

```
0.2 setlinewidth
```

Команда setlinewidth имеет один аргумент, который помещается на стек перед ее вызовом. При выполнении команды он снимается со стека, и толщина линии устанавливается равной числу, снятому со стека.

Третья строка перемещает текущую позицию в точку с координатами $x = 1$, $y = 15$.

```
1 15 moveto   % переместиться в точку (1, 15)
```

(В качестве единиц используются миллиметры, начало координат находится в левом нижнем углу страницы.) В стек сначала добавляются два числа, соответствующие координатам x и y, и затем выполняется команда moveto, которая снимает два числа со стека и перемещает курсор в точку, координаты которой были получены из стека.

Большинство строк программы понятны благодаря комментариям, которые в языке PostScript можно записывать в конце любой строки после символа %. Отметим две конструкции, которые использованы в программе. Фрагмент

```
/Func {
    30 sub 0.2 mul 57.296 mul sin 5 mul 15 add
} def
```

определяет функцию с именем Func. Функцию затем можно вызвать, просто записав ее имя, как это делается, например, в строке

```
Func      % заменить x в вершине стека на f(x)
```

Вызов функции в PostScript'е эквивалентен записи тела функции в точке вызова. Параметры и результат функции передаются через стек.

Тело функции при ее определении записывается внутри фигурных скобок. В данном случае это строка

```
30 sub 0.2 mul 57.296 mul sin 5 mul 15 add
```

Функция вызывается при условии, что ее аргумент x находится на вершине стека. В результате выполнения тела функции вершина стека заменяется на выражение

$$5 \cdot \sin((x - 30) \cdot 0.2 \cdot 57.296) + 15$$

Здесь используются масштабирующие множители, чтобы график выглядел красиво на печати. Так, масштаб по обеим осям равен 5 мм, поэтому мы умножаем значение sin на 5, а аргумент sin на 0.2, т.е. на $1/5$. Центр графика смещается в точку с координатами $(30, 15)$, поэтому мы прибавляем к значению sin число 15, а от аргумента вычитаем 30. Наконец, аргумент функции sin в языке PostScript задается в градусах. Чтобы перейти от радианов к градусам, мы умножаем агрумент в радианах на множитель $57.296 = 180/\pi$. Сравните выражения:

$$5 \cdot \sin((x - 30) \cdot 0.2 \cdot 57.296) + 15$$

```
x 30 sub 0.2 mul 57.296 mul sin 5 mul 15 add
```

Первое представляет собой обычную запись формулы, второе — обратную польскую запись.

Фрагмент

```
2.5 0.5 58 { % цикл для x от 2.5 до 58 шаг 0.5
    dup      %   удвоить вершину стека
    Func     %   заменить x в вершине стека на f(x)
    lineto   %   провести линию к точке (x, f(x))
} for        % конец цикла
```

представляет собой *арифметический цикл* (см. с. 41). На вершину стека последовательно помещается число x в диапазоне от 2.5 до 58 с шагом 0.5. Для каждого значения выполняется тело цикла, заключенное в фигурные скобки. В теле цикла вершина стека сначала удваивается командой *dup*, после ее выполнения в вершине стека будут лежать два одинаковых значения x, x. Затем вызывается функция *Func*, которая заменяет значение x в вершине стека на y, где y равно значению функции. Затем проводится линия от предыдущей точки к точке (x, y). При этом команда *lineto* удаляет оба значения x, y из стека.

Язык PostScript является достаточно мощным языком программирования, в нем есть переменные, функции, циклы, условный оператор и т.п. Используемая в нем обратная польская запись очень удобна для выполнения на компьютере. Поэтому интерпретатор языка PostScript легко встраивается в конструкцию принтера (который,

конечно же, всегда содержит более или менее сложный компьютер внутри себя) и не сильно увеличивает стоимость принтера.

Язык Java, байткод которого также представляет собой обратную польскую запись, был тоже первоначально разработан для программирования недорогих бытовых приборов. Стековый вычислитель устроен просто и может быть применен там, где быстродействие не играет особой роли, зато важна скорость и дешевизна разработки программы и аппаратуры.

4.5. Ссылочные реализации структур данных

Большинство структур данных реализуется на базе массива. Все реализации можно разделить на два класса: непрерывные и ссылочные. В непрерывных реализациях элементы структуры данных располагаются последовательно друг за другом в непрерывном отрезке массива, причем порядок их расположения в массиве соответствует их порядку в реализуемой структуре. Рассмотренные выше реализации очереди и стека относятся к непрерывным.

В ссылочных реализациях элементы структуры данных хранятся в произвольном порядке. При этом вместе с каждым элементом хранятся ссылки на один или несколько соседних элементов. В качестве ссылок могут выступать либо индексы ячеек массива, либо адреса памяти. Можно представить себе шпионскую сеть, в которой каждый участник знает лишь координаты одного или двух своих коллег. Контрразведчикам, чтобы обезвредить сеть, нужно пройти последовательно по всей цепочке, начиная с выявленного шпиона.

Ссылочные реализации обладают двумя ярко выраженными недостатками: 1) для хранения ссылок требуется дополнительная память; 2) для доступа к некоторому элементу структуры необходимо сначала добраться до него, проходя последовательтно по цепочке других элементов. Казалось бы, зачем нужны такие реализации?

Все недостатки ссылочных реализаций компенсируются одним чрезвычайно важным достоинством: в них можно добавлять и удалять элементы в середине структуры данных, не перемещая остальные элементы.

4.5.1. Массовые операции

Массовые операции — это операции, затрагивающие значительную часть всех элементов структуры данных. Пусть нужно добавить или удалить один элемент. Если при этом приходится, например, переписывать значительную часть остальных элементов с одного места на другое, то говорят, что добавление или удаление приводит к массовым операциям. Массовые операции — это бедствие для программиста, то, чего он всегда стремится избежать. Хорошая реализация структуры данных — та, в которой массовых операций либо нет совсем, либо они происходят очень редко. Например, добавление элемента должно выполняться за ограниченное число шагов, независимо от того, содержит ли структура десять или десять тысяч элементов.

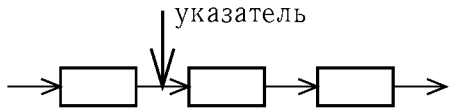
В непрерывных реализациях добавление или удаление элементов в середине структуры неизбежно приводит к массовым операциям. Поэтому структуры, в которых можно удалять или добавлять элементы в середине, обязательно должны быть реализованы ссылочным образом.

> Пример неудачного использования непрерывных реализаций — файловые системы в некоторых старых операционных системах, например, в уже упомянутой ОС ЕС или в системе РАФОС, применявшейся на СМ ЭВМ, на старых советских персональных компьютерах «Электроника» и т.п. В современных файловых системах файлы фрагментированы, т.е. кусочки большого файла, непрерывного с точки зрения пользователя,на самом деле могут быть «разбросаны» по всему диску. Раньше это было не так, файлы должны были обязательно занимать непрерывный участок на диске. При постоянной работе файлы уничтожались и создавались заново на новом месте — и всякое редактирование текстового файла приводило к его обновлению. В результате свободное пространство на диске становилось фрагментированным, т.е. состоящим из множества небольших кусков. Возникала ситуация, когда большой файл невозможно записать на диск: хотя свободного места в сумме много, нет достаточно большого свободного фрагмента. Приходилось постоянно выполять длительную и опасную процедуру сжатия диска, которая часто приводила к потере всех данных на нем.

4.5.2. Список

Классический пример структуры данных последовательного доступа, в которой можно удалять и добавлять элементы в середине структуры, — это линейный список. Различают однонаправленный и двунаправленный списки (иногда говорят односвязный и двусвязный).

Элемены списка как бы выстроены в цепочку друг за другом. У списка есть начало и конец. Имеется также указатель списка, который располагается между элементами. Если мысленно вообразить, что соседние элементы списка связаны между собой веревкой, то указатель — это ленточка, которая вешается на веревку. В любой момент времени в списке доступны лишь два элемента — элементы до указателя и за указателем.



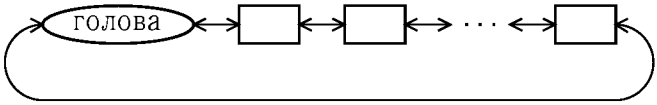
В однонаправленном списке указатель можно передвигать лишь в одном направлении — вперед, в направлении от начала к концу. Кроме того, можно установить указатель в начало списка, перед его первым элементом. В отличие от однонаправленного списка, двунаправленный абсолютно симметричен, указатель в нем можно передвигать вперед и назад, а также устанавливать как перед первым, так и за последним элементами списка.

В двунаправленном списке можно добавлять и удалять элементы до и за указателем. В однонаправленном списке добавлять элементы можно также с обеих сторон от указателя, но удалять элементы можно только за указателем.

Удобно считать, что перед первым элементом списка располагается специальный «пустой» элемент, который называется *головой списка*. Голова списка присутствует всегда, даже в пустом списке. Благодаря этому можно предполагать, что перед указателем всегда есть какой-то элемент, что упрощает процедуры добавления и удаления элементов.

В двунаправленном списке считают, что вслед за последним элементом списка вновь следует голова списка, т.е. список зациклен в

кольцо.



Можно было бы точно так же зациклить и однонаправленной список. Но гораздо чаще считают, что за последним элементом однонаправленного списка ничего не следует. Однонаправленный список, таким образом, представляет собой цепочку, начинающуюся с головы списка, за которой следует первый элемент, затем второй и так далее вплоть до последнего элемента, а заканчивается цепочка ссылкой в никуда.

4.5.3. Ссылочная реализация списка

Мы рассмотрели абстрактное понятие списка. Но в программировании зачастую отождествляют понятие списка с его ссылочной реализацией на базе массива или непосредственно на базе оперативной памяти.

Основная идея реализации двунаправленного списка заключается в том, что вместе с каждым элементом хранятся ссылки на следующий и предыдущий элементы. В случае реализации на базе массива ссылки представляют собой индексы ячеек массива. Чаще, однако, элементы списка не располагают в каком-либо массиве, а просто размещают каждый по отдельности в оперативной памяти, выделенной данной задаче. (Обычно элементы списка размещаются в так называемой *динамической памяти*, или *куче* — это область оперативной памяти, в которой можно при необходимости захватывать куски нужного размера, а после использования освобождать, т.е. возвращать обратно в кучу.) В качестве ссылок в этом случае используют адреса элементов в оперативной памяти.

Голова списка хранит ссылки на первый и последний элементы списка. Поскольку список зациклен в кольцо, то следующим за головой списка будет его первый элемент, а предыдущим — последний элемент. Голова списка хранит только ссылки и не хранит никакого

элемента. Это как бы пустой ящик, в который нельзя ничего положить и который используется только для того, чтобы написать на нем адреса следующего и предыдущего ящиков, т.е. первого и последнего элементов списка. Когда список пуст, голова списка зациклена сама на себя.

Указатель списка реализуется в виде ссылки на следующий и предыдущий элементы, он просто отмечает некоторое место в цепочке элементов.

В случае однонаправленного списка хранится только ссылка на следующий элемент, таким способом экономится память. Голова однонаправленного списка хранит ссылку на первый элемент списка. Последний элемент списка хранит нулевую ссылку, т.е. ссылку «в никуда», т.к. в программах нулевой адрес никогда не используется.

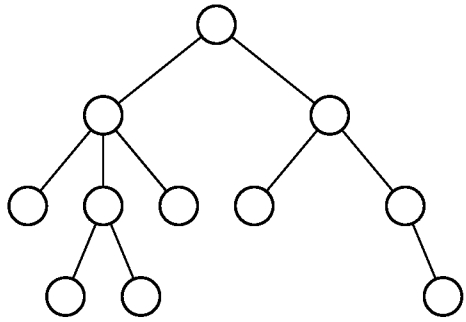
Ценность ссылочной реализации списка состоит в том, что процедуры добавления и удаления элементов не приводят к массовым операциям. Рассмотрим, например, операцию удаления элемента за указателем. Читая ссылку на следующий элемент в удаляемом элементе, мы находим, какой элемент должен будет следовать за указателем после удаления текущего элемента. После этого достаточно «связать» элемент до указателя с новым элементом за указателем. А именно, обозначим через X адрес элемента до указателя, через Y — адрес нового элемента за указателем. В поле «следующий» для элемента с адресом X надо записать значение Y, в поле «предыдущий» для элемента с адресом Y — значение X. Таким образом, при удалении элемента за указателем он исключается из цепочки списка, для этого достаточно лишь поменять ссылки в двух соседних элементах. Аналогично, для добавления элемента достаточно включить его в цепочку, а для этого также нужно всего лишь модифицировать ссылки в двух соседних элементах. Добавляемый элемент может располагаться где угодно, следовательно, нет никаких проблем с захватом и освобождением памяти под элементы.

4.5.4. Деревья и графы

Граф — это фигура, которая состоит из вершин и ребер, соединяющих вершины. Например, схема линий метро — это граф. Ребра могут иметь направления, т.е. изображаться стрелочками; такие графы называются ориентированными. Допустим, надо постро-

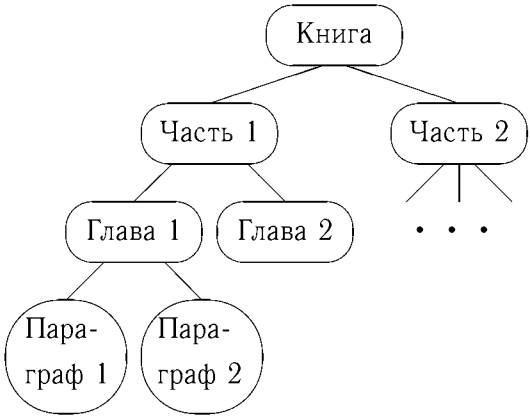
ить схему автомобильного движения по улицам города. Почти во всех городах есть много улиц с односторонним движением. Поэтому такая транспортная схема должна представляться ориентированным графом. Улице с односторонним движением соответствует стрелка, с двусторонним — пара стрелок в противоположных направлениях. Вершины такого графа соответствуют перекресткам и тупикам.

Дерево — это связный граф без циклов. Кроме того, в дереве выделена одна вершина, которая называется корнем дерева. Остальные вершины упорядочиваются по длине пути от корня дерева.



Зафиксируем некоторую вершину X. Вершины, соединенные с X ребрами и расположенные дальше нее от корня дерева, называются «детьми» или «сыновьями» вершины X. Сыновья упорядочены слева направо. Вершины, у которых нет сыновей, называются терминальными. Дерево обычно изображают перевернутым, корнем вверх.

Деревья в программировании используются значительно чаще, чем графы. Так, на построении дереьев основаны многие алгоритмы сортировки и поиска. Компиляторы в процессе перевода программы с языка высокого уровня на машинный язык представляют фрагменты программы в виде деревьев, которые называются синтаксическими. Деревья естественно применять всюду, где имеются какие-либо иерархические структуры, т.е. структуры, которые могут вкладываться друг в друга. Примером может служить оглавление книги.



Пусть книга состоит из частей, части — из глав, главы — из параграфов. Сама книга представляется корнем дерева, из которого выходят ребра к вершинам, соответствующим частям книги. В свою очередь, из каждой вершины-части книги выходят ребра к вершинам-главам, входящим в эту часть, и так далее. Файловую систему компьютера также можно представить в виде дерева. Вершинам соответствуют каталоги (их также называют директориями или папками) и файлы. Из вершины-каталога выходят ребра к вершинам, соответствующим всем каталогам и файлам, которые содержатся в данном каталоге. Файлы представляются терминальными вершинами дерева. Корню дерева соответствует корневой каталог диска. Программы, осуществляющие работу с файлами, такие, как Norton Commander в системе MS DOS или «Проводник» Windows, могут изображать файловую систему графически в виде дерева.

Ссылочные реализации как будто специально придуманы для реализации деревьев. Вершина дерева представляется в виде объекта, содержащего ссылки на родительскую вершину и на всех сыновей, а также некоторую дополнительную информацию, зависящую от конкретной задачи. Объект, представляющий вершину дерева, занимает область фиксированного размера, которая обычно размещается в динамической памяти. Число сыновей обычно ограничено, исходя из смысла решаемой задачи. Так, очень часто рассматриваются бинарные деревья, в которых число сыновей у произвольной вершины не превышает двух. Если один или несколько сыновей у вершины отсутствуют, то соответствующие ссылки содержат нулевые значения. Таким образом, у терминальных вершин все ссылки на сыновей ну-

левые.

При работе с деревьями очень часто используются рекурсивные алгоритмы, т.е. алгоритмы, которые могут вызывать сами себя. При вызове алгоритма ему передается в качестве параметра ссылка на вершину дерева, которая рассматривается как корень поддерева, растущего из этой вершины. Если вершина терминальная, т.е. у нее нет сыновей, то алгоритм просто применяется к данной вершине. Если же у вершины есть сыновья, то он рекурсивно вызывается также для каждого из сыновей. Порядок обхода поддеревьев зависит от сути алгоритма. В главе, посвященной языку Си, уже был рассмотрен простейший рекурсивный алгоритм, подсчитывающий число терминальных вершин бинарного дерева (см. с. 181).

Ниже приведен еще один рекурсивный алгоритм, определяющий высоту дерева. *Высотой дерева* называется максимальная из длин всевозможных путей от корня дерева к терминальным вершинам. Под *длиной пути* понимается число вершин, входящих в него, включая первую и последнюю вершины. Так, дерево, состоящее из одной корневой вершины, имеет высоту 1, дерево, приведенное на рисунке в начале этого раздела — высоту 4.

```
цел алгоритм высота_дерева(вход: вершина V)
| Дано: V - ссылка на корень поддерева
| Надо: Подсчитать высоту поддерева
начало
| цел h, m, s;
| h := 1;
| если у вершины V есть сыновья
| | то    // Ищем поддерево максимальной высоты
| | m := 0;
| | цикл для каждого сына X вершины V выполнить
| | | s := высота_дерева(X);  // Рекурсия!
| | | если s > m
| | | | то m := s;
| | | конец если
| | конец цикла
| | h := h + m;
| конец если
| ответ := h;
```

```
конец алгоритма
```

4.6. Множество

Множество — это структура данных, содержащая конечный набор элементов некоторого типа. Каждый элемент содержится только в одном экземпляре, т.е. разные элементы множества не равны между собой. Элементы множества никак не упорядочены. В множество M можно добавить элемент x, из множества M можно удалить элемент x. Если при добавлении элемента x он уже содержится в множестве M, то ничего не происходит. Аналогично, никакие действия не совершаются при удалении элемента x, когда он не содержится в множестве M. Наконец, для заданного элемента x можно определить, содержится ли он в множестве M. Множество — это потенциально неограниченная структура, оно может содержать любое конечное число элементов.

В некоторых языках программирования накладывают ограничения на тип элементов и на максимальное количество элементов множества. Так, иногда рассматривают множество элементов дискретного типа, число элементов которого не может превышать некоторой константы, задаваемой при создании множества. (Тип называется дискретным, если все возможные значения данного типа можно занумеровать целыми числами.) Для таких множеств употребляют название Bitset ("набор битов") или просто Set. Как правило, для реализации таких множеств используется битовая реализация множества на базе массива целых чисел. Каждое целое число рассматривается в двоичном представлении как набор битов, содержащий 32 элемента. Биты внутри одного числа нумеруются справа налево (от младших разрядов к старшим); нумерация битов продолжается от одного числа к другому, когда мы перебираем элементы массива. К примеру, массив из десяти целых чисел содержит 320 битов, номера которых изменяются от 0 до 319. Множество в данной реализации может содержать любой набор целых чисел в диапазоне от 0 до 319. Число N содержится в множестве тогда и только тогда, когда бит с номером N равен единице (программисты говорят «бит установлен»). Соответственно, если число N не содержится в множестве, то бит с номером N равен нулю (программисты говорят «бит очищен»).

Пусть, например, множество содержит элементы 0, 1, 5, 34. Тогда в первом элементе массива установлены биты с номерами 0, 1, 5, во втором — бит с номером 2 = 34 - 32. Соответственно, двоичное представление первого элемента массива равно 10011 (биты нумеруются справа налево), второго — 100, это числа 19 и 4 в десятичном представлении. Все остальные элементы массива нулевые.

Хотя в языке программирования Паскаль слово Set, в переводе «множество», закреплено за ограниченным множеством элементов дискретного типа, такими множествами далеко не исчерпываются потребности программирования. Например, множество точек на плоскости или множество текстовых строк не являются таковыми. Для множеств общего вида битовая реализация не подходит. Ниже будут рассмотрены несколько других реализаций, используемых для любых множеств.

В программировании довольно часто рассматривают структуру чуть более сложную, чем просто множество: *нагруженное множество*. Пусть каждый элемент множества содержится в нем вместе с дополнительной информацией, которую называют нагрузкой элемента. При добавлении элемента в множество нужно также указывать нагрузку, которую он несет. В разных языках программирования и в различных стандартных библиотеках такие структуры называют «Отображением» (Map) или «Словарем» (Dictionary). Действительно, элементы множества как бы отображаются на нагрузку, которую они несут (заметим, что в математике понятие функции или отображения определяется строго как множество пар; первым элементом каждой пары является конкретное значение аргумента функции, вторым — значение, на которое функция отображает аргумент). В интерпретации «Словаря» элемент множества — это иностранное слово, нагрузка элемента — это перевод слова на русский язык (разумеется, перевод может включать несколько вариантов, но здесь перевод рассматривается как единый текст).

Подчеркнем еще раз, что все элементы содержатся в нагруженном множестве в одном экземпляре, т.е. разные элементы множества не могут быть равны друг другу. В отличие от самих элементов, их нагрузки могут совпадать (так, различные иностранные слова могут иметь одинаковый перевод). Поэтому иногда элементы нагруженного множества называют *ключами*, их нагрузки — *значениями* ключей.

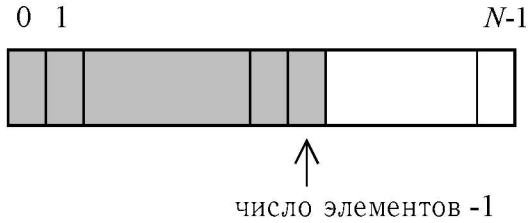
Каждый ключ уникален. Принято говорить, что ключи *отображаются* на их значения.

В качестве примера нагруженного множества наряду со словарем можно рассмотреть множество банковских счетов. Банковский счет — это уникальный идентификатор, состоящий в случае российских банков из 20 десятичных цифр. Нагрузка счета — это вся информация, которая ему соответствует, включающая имя и адрес владельца счета, код валюты, сумму остатка, информацию о последних транзакциях и т.п.

Наиболее часто применяемая операция в нагруженном множестве — это определение нагрузки для заданного элемента x (значения ключа x). Реализация этой операции включает поиск элемента x в множестве, поэтому эффективность любой реализации множества определяется прежде всего быстротой поиска.

4.6.1. Реализации множества: последовательный и бинарный поиск, хеширование

Рассмотрим для простоты реализацию обычного, не нагруженного, множества. (Реализация нагруженного множества аналогична реализации обычного, просто параллельно с элементами нужно дополнительно хранить нагрузки элементов.) В «наивной» реализации множества его элементы хранятся в массиве, начиная с первой ячейки. Специальная переменная содержит текущее число элементов множества, т.е. количество используемых в данный момент ячеек массива.



При добавлении элемента x к множеству сначала необходимо определить, содержится ли он в множестве (если содержится, то множество не меняется). Для этого используется процедура поиска

элемента, которая отвечает на вопрос, принадлежит ли элемент x множеству, и, если принадлежит, выдает индекс ячейки массива, содержащей x. Та же процедура поиска используется и при удалении элемента из множества. При добавлении элемента он дописывается в конец (в ячейку массива с индексом «число элементов») и переменная «число элементов» увеличивается на единицу. Для удаления элемента достаточно последний элемент множества переписать на место удаляемого и уменьшить переменную «число элементов» на единицу.

В «наивной» реализации элементы множества хранятся в массиве в произвольном порядке. Это означает, что при поиске элемента x придется последовательно перебрать все элементы, пока мы либо не найдем x, либо не убедимся, что его там нет. Пусть множество содержит 1,000,000 элементов. Тогда, если x содержится в множестве, придется просмотреть в среднем 500,000 элементов, если нет — все элементы. Поэтому «наивная» реализация годится только для небольших множеств.

4.6.2. Бинарный поиск

Пусть можно сравнивать элементы множества друг с другом, определяя, какой из них больше. (Например, для текстовых строк применяется лексикографическое сравнение: первые буквы сравниваются по алфавиту; если они равны, то сравниваются вторые буквы и т.д.) Тогда можно существенно убыстрить поиск, применяя алгоритм бинарного поиска. Для этого элементы множества хранятся в массиве в возрастающем порядке. Идея бинарного поиска иллюстрируется следующей шуточной задачей: «Как поймать льва в пустыне? Надо разделить пустыню забором пополам, затем ту половину, в которой находится лев, снова разделить пополам и так далее, пока лев не окажется пойманным».

В алгоритме бинарного поиска мы на каждом шагу делим отрезок массива, в котором может находиться искомый элемент x, пополам. Рассматриваем элемент y в середине отрезка. Если x меньше y, то выбираем левую половину отрезка, если больше, то правую. Таким образом, на каждом шаге размер отрезка массива, в котором может находиться элемент x, уменьшается в два раза. Поиск заканчивается, когда размер отрезка массива (т.е. расстояние между его правым и

левым концами) становится равным единице, т.е. через $[\log_2 n]+1$ шагов, где n — размер массива. В нашем примере это произойдет после 20 шагов (т.к. $\log_2 1000000 < 20$). Таким образом, вместо миллиона операций сравнения при последовательном поиска нужно выполнить всего лишь 20 операций при бинарном.

Запишем алгоритм бинарного поиска на псевдокоде. Дан упорядоченный массив a вещественных чисел (вещественные числа используются для определенности; бинарный поиск можно применять, если на элементах множества определен линейный порядок, т.е. для любых двух элементов можно проверить их равенство или определить, какой их них больше). Пусть текущее число элементов равно n. Элементы массива упорядочены по возрастанию:

$$a[0] < a[1] < \cdots < a[n-1].$$

Мы ищем элемент x. Требуется определить, содержится ли x в массиве. Если элемент x содержится в массиве, то надо определить индекс i ячейки массива, содержащей x:

$$\text{найти } i: \quad a[i] = x.$$

Если же x не содержится в массиве, то надо определить индекс i, такой, что при добавлении элемента x в i-ю ячейку массива элементы массива останутся упорядоченными, т.е.

$$\text{найти } i: \quad a[i-1] < x < a[i]$$

(считается, что $a[-1] = -\infty$, $a[n] = +\infty$). Объединив оба случая, получим неравенство

$$a[i-1] < x \le a[i].$$

Используем схему построения цикла с помощью инварианта, см. раздел 1.5.2. В процессе выполнения хранятся два индекса b и e (от слов begin и end), такие, что

$$a[b] < x \le a[e].$$

Индексы b и e ограничивают текущий отрезок массива, в котором осуществляется поиск. Приведенное неравенство является *инвариантом цикла*. Перед началом выполнения цикла рассматриваются

разные исключительные случаи — когда массив пустой ($n = 0$), когда x не превышает минимального элемента массива ($x \le a[0]$) и когда x больше максимального элемента ($x > a[n-1]$). В общем случае выполняется неравенство

$$a[0] < x \le a[n-1].$$

Полагая $b = 0$, $e = n - 1$, мы обеспечиваем выполнение инварианта цикла перед началом выполнения цикла. *Условие завершения* состоит в том, что длина участка массива, внутри которого может находится элемент x, равна единице:

$$b - e = 1.$$

В этом случае

$$a[e-1] < x \le a[e],$$

т.е. искомый индекс i равен e, а элемент x содержится в массиве тогда и только тогда, когда $x = a[e]$. В цикле мы вычисляем середину c отрезка $[b, e]$

$$c = \text{целая часть}((b+e)/2)$$

и выбираем ту из двух половин $[b, c]$ или $[c, e]$, которая содержит x. Для этого достаточно сравнить x со значением $a[c]$ в середине отрезка. Завершение цикла обеспечивается тем, что величина $e - b$ монотонно убывает после каждой итерации цикла.

Приведем текст алгоритма бинарного поиска:

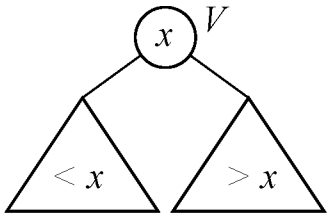
```
лог алгоритм бинарный_поиск(
    вход: цел n, вещ a[n], вещ x, выход: цел i
)
| Дано: n -- число элементов в массиве a,
|       a[0] < a[1] < ... < a[n-1]
| Надо: ответ := (x содержится в массиве),
|       вычислить i, такое, что
|           a[i-1] < x <= a[i]
|       (считая a[-1] = -беск., a[n] = +беск.)
начало
| цел b, e, c;
|
```

```
| // Рассматриваем сначала исключительные случаи
| если (n == 0)
| | то i = 0;
| | ответ := ложь;
| иначе если (x <= a[0])
| | i := 0;
| | ответ := (x == a[0]);
| иначе если (x > a[n-1])
| | i := n
| | ответ := ложь;
| иначе
| | // Общий случай
| | утверждение: a[0] < x <= a[n-1];
| | b := 0; e := n-1;
| |
| | цикл пока e - b > 1
| | | инвариант: a[b] < x  и  x <= a[e];
| | | c := целая часть((a + b) / 2);
| | | если x <= a[c]
| | | | то e := c;
| | | иначе
| | | |    b := c;
| | | конец если
| | конец цикла
| |
| | утверждение: e - b == 1  и
| |              a[b] < x <= a[e];
| | i := e;
| | ответ := (x == a[i]);
| конец если
конец алгоритма
```

4.6.3. Реализации множества на базе деревьев

Реализация множества с помощью бинарного поиска во всех отношениях лучше «наивной» реализации. Вместе с тем, она все же имеет недостатки: 1) при добавлении и удалении элементов в середине массива приходится переписывать элементы в конце массива

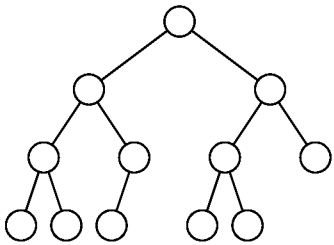
на новое место, чтобы освободить место для добавляемого элемента либо закрыть образовавшуюся лакуну при удалении элемента; 2) поиск выполняется гарантированно быстро, но все-таки не мгновенно. От первого из этих недостатков можно избавиться, применяя вместо непрерывной реализации на базе массива ссылочную реализацию, при которой элементы множества содержатся в вершинах *бинарного дерева*. Элементы в вершинах упорядочены таким образом, что, если зафиксировать некоторую вершину V и рассмотреть два поддерева, соответствующих левому и правому сыновьям вершины, то все элементы в вершинах левого поддерева должны быть меньше, чем элемент в вершине V, а все элементы в вершинах правого поддерева должны быть больше него.



Для такого дерева можно также применять алгоритм бинарного поиска. Максимальное число сравнений при поиске в таком дереве равняется его высоте (т.е. максимальной длине пути от корня к терминальной вешине). Чтобы поиск выполнялся быстро, дерево должно быть *сбалансированным*, т.е. все его ветви должны иметь почти одинаковую длину.

Точное определение сбалансированности следующее: будем считать, что у каждой вершины, включая терминальные, ровно два сына, при необходимости добавляя *внешние*, или нулевые, вершины. Например, у терминальной вершины оба сына нулевые. (Это в точности сответствует представлению дерева в языке Си, где каждая вершина хранит два указателя на сыновей; если сына нет, то соответствующий указатель нулевой.) Обычные вершины дерева будем называть собственными. Рассмотрим путь от корня дерева к внешней (нулевой) вершине. Длиной пути считается количество собственных вершин в нем. Дерево называется *сбалансированным*, если длины всех возможных путей от корня дерева к внешним вершинам различаются не более чем на единицу. Иногда в литературе такие деревья называют *почти сбалансированными*, понимая под сбалансированностью строгое равенство длин всех путей от корня к внешним узлам; мы, однако, будем придерживаться нестрогого определения.

Пример сбалансированного дерева представлен на рисунке.



Высота сбалансированного дерева h оценивается логарифмически в зависимости от числа вершин n:

$$h \leq \log_2 n + 1.$$

Поскольку максимальное число сравнений при поиске элемента в упорядоченном бинарном дереве равняется высоте дерева, поиск в сбалансированном дереве осуществляется исключительно быстро, за время, логарифмически зависящее от числа элементов множества. (Можно доказать, что это является теоретической оценкой снизу: никакой алгоритм не может в общем случае находить элемент быстрее, чем за $\log_2 n$ операций.)

Для эффективной реализации множества на базе дерева процедуры добавления и удаления элементов должны сохранять свойство сбалансированности (или почти сбалансированности). Рассмотрим коротко две наиболее популярные схемы реализации.

AVL-деревья

Так называемые AVL-деревья (названные в честь их двух изобретателей Г.М. Адельсона-Вельского и Е.М. Ландиса) хранят дополнительно в каждой вершине разность между высотами левого и правого поддеревьев, которая в сбалансированном дереве может принимать только три значения: -1, 0, 1. Строго говоря, AVL-деревья не являются сбалансированными в смысле приведенного выше определения. Требуется только, чтобы для любой вершины AVL-дерева разность высот ее левого и правого поддеревьев была по абсолютной величине не больше единицы. При этом длины путей от корня к внешним вершинам могут различаться больше, чем на единицу.

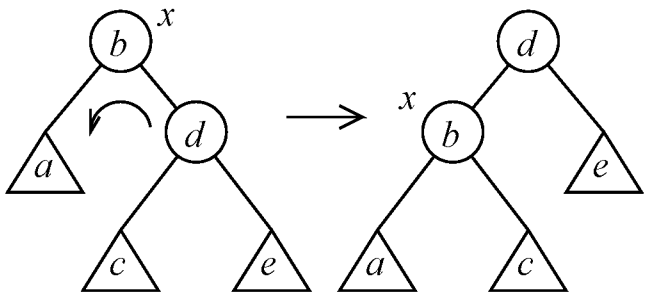
Можно, тем не менее, доказать, что и в случае AVL-деревьев их высота оценивается сверху логарифмически в зависимости от числа вершин:

$$h \leq C \log_2 n,$$

где константа $C = 1.5$. Обычно константы не очень важны в практическом программировании — принципиально лишь, по какому закону увеличивается время работы алгоритма при увеличении n. В данном случае зависимость логарифмическая, т.е. наилучшая из всех возможных (поскольку поиск невозможен быстрее чем за $\log_2 n$ операций).

Новый элемент всегда добавляется в дерево в соответствии с упорядоченностью как левый или правый сын некоторой вершины, у которой данного сына до этого не было (или, как мы считаем, сын являлся внешним). Новая вершина добавляется как терминальная. После этого выполняется процедура восстановления балансировки. В ней используются следующие элементарные преобразования дерева, сохраняющие упорядоченность вершин:

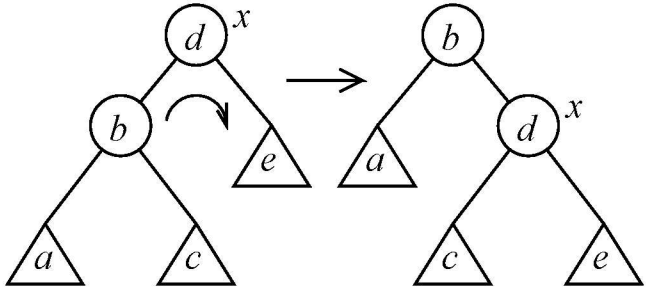
1) вращение вершины x поддерева влево:



Здесь вершина x поддерева, которая является его корнем, опускается вниз и влево. Бывший правый сын d вершины x становится новым корнем поддерева, а x становится левым сыном d. (Вершины x и d, «начальник» и «подчиненный», как бы меняются ролями: бывший начальник становится подчиненным.) Поддерево c, которое было левым сыном вершины d, переходит в подчинение от вершины d к вершине x и становится ее правым сыном. Отметим, что упорядоченность вершин сохраняется: $a < b < c < d < e$. Таким образом, для выполнения преоб-

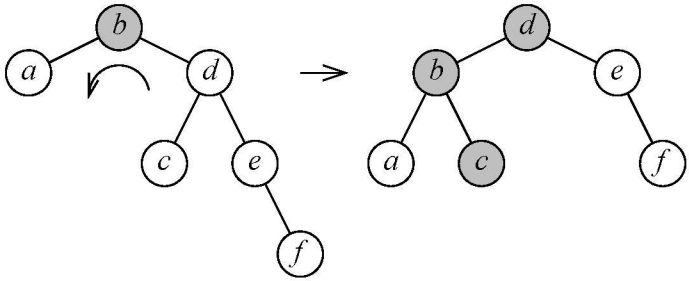
разования надо лишь заменить фиксированное количество указателей в вершинах x, d, c и, возможно, в родительской для x вершине;

2) вращение вершины x поддерева вправо:



Здесь вершина x опускается вниз и вправо, ее бывший левый сын b становится новым корнем поддерева, а x — его правым сыном. Поддерево c переходит в подчинение от b к x.

Операции вращения носят локальный характер и позволяют при необходимости исправить баланс поддерева с корнем x. Например, для восстановления баланса дерева, показанного на следующем рисунке, достаточно выполнить одно вращение вершины b влево:



В случае AVL-деревьев операции вращения повторяются в цикле при восстановлении баланса после добавления или удаления элемента, число вращений не превышает $C \cdot h$, где h — высота дерева, C — константа. Таким образом, как поиск элемента, так и его добавление или удаление выполняется за логарифмическое время: $t \leq C \cdot \log_2 n$.

Красно-черные деревья

Исторически AVL-деревья, изобретенные в 1962 г., были одной из первых схем реализации почти сбалансированных деревьев. В настоящее время, однако, более популярна другая схема: *красно-черные деревья*, или *RB-деревья*, от англ. Red-Black Trees. Красно-черные деревья были введены Р. Байером в 1972 г. В стандартной библиотеке классов языка C++ исполнители «множество» и «нагруженное множество» — классы set и map — реализованы именно как красно-черные деревья.

Вместо баланс-фактора, применяемого в AVL-деревьях, RB-деревья используют цвета вершин. Каждая вершина окрашена либо в красный, либо в черный цвет. (В реализации за цвет отвечает логическая переменная.) При этом выполняется несколько дополнительных условий:

1) каждая внешняя (или нулевая) вершина считается черной;

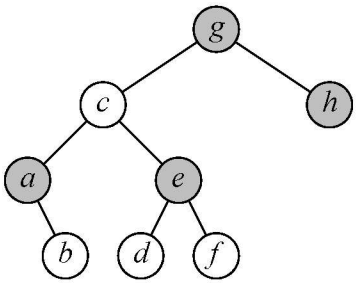
2) корневая вершина дерева черная;

3) у красной вершины дети черные;

4) всякий путь от корня дерева к произвольной внешней вершине имеет одно и то же количество черных вершин.

Последний пункт определения означает сбалансированность дерева по черным вершинам.

Ниже приведен пример красно-черного дерева. Черные вершины изображены темно-серым цветом, красные — белым.



Из пункта 3) определения следует, что в произвольном пути от корня к терминальной вершине не может быть двух красных вершин

подряд. Это означает, что, поскольку число черных вершин в любом пути одинаково, длины разных путей к терминальным вершинам отличаются не более чем вдвое. Это свойство близко по своей сути к сбалансированности. Несложно показать, что для красно-черного дерева справедлива следующая оценка сверху на высоту дерева в зависимости от числа вершин:

$$h \leq 2\log_2(n+1).$$

Из этого следует, что поиск в красно-черном дереве также выполняется за логарифмическое время.

Новая вершина добавляется в красно-черное дерево как терминальная после процедуры поиска (этим RB-дерево ничем не отличается от других упорядоченных деревьев). Новая вершина окрашивается в *красный цвет*. При этом пункт 3) в определении красно-черного дерева может нарушиться. Поэтому после добавления, а также удаления вершины выполняется процедура восстановления структуры красно-черного дерева, играющая ту же роль, что и восстановление балансировки AVL-дерева. Преимущество красно-черных деревьев состоит в том, что процедура восстановления более простая. Во многих случаях она ограничивается перекрашиванием вершин. В ней также могут выполняться операции вращения вершины влево и вправо (см. с. 301), но число вращений может быть не больше двух при добавлении элемента и не больше четырех при удалении. Всего число операций при восстановлении структуры RB-дерева оценивается сверху через высоту дерева:

$$\text{число операций} \leq K \cdot h,$$

где h — высота дерева, K — константа. Поскольку для высоты RB-дерева справедлива приведенная выше логарифмическая оценка от числа вершин n, получаем оценку

$$\text{число операций} \leq C\log_2 n,$$

где C — константа. Таким образом, добавление и удаление элементов выполняется в случае красно-черных деревьев за логарифмическое время в зависимости от числа вершин дерева.

4.6.4. Хеширование

Рассмотрим другую реализацию множества, в которой поиск элемента чаще всего происходит почти мгновенно. Правда, в исключительных случаях поиск может быть долгим, и поэтому такая реализация не подходит для программ, в которых требуется повышенная надежность, например, при управлении реальными процессами.

Идея хеш-реализации состоит в том, что мы сводим работу с одним большим множеством к работе с массивом небольших множеств. Рассмотрим, к примеру, записную книжку. Она содержит список фамилий людей с их телефонами (телефоны — это нагрузка элементов множества). Страницы записной книжки помечены буквами алфавита; страница, помеченная некоторой буквой, содержит только фамилии, начинающиеся с этой буквы. Таким образом, все множество фамилий разбито на 28 подмножеств, соответствующих буквам русского алфавита. При поиске фамилии мы сразу открываем записную книжку на странице, помеченной первой буквой фамилии, и в результате поиск значительно убыстряется.

Разбиение множества на подмножества осуществляется с помощью так называемой *хеш-функции*. Хеш-функция определена на элементах множества и принимает целые неотрицательные значения. Она должна быть подобрана таким образом, чтобы 1) ее можно было легко вычислять; 2) она должна принимать всевозможные различные значения приблизительно с равной вероятностью; 3) желательно, чтобы на близких значениях аргумента она принимала далекие друг от друга значения (свойство, противоположное математическому понятию непрерывности). В приведенном примере значение хеш-функции для данной фамилии равно номеру первой буквы фамилии в русском алфавите.

Параметром реализации является число подмножеств, которое также называют размером хеш-таблицы. Пусть он равен N. Тогда хеш-функция должна принимать значения 0, 1, 2, ... , $N-1$. Если изначально у нас есть некоторая функция $H(x)$, принимающая произвольные целые неотрицательные значения, то в качестве хеш-функции можно использовать функцию $h(x)$, равную остатку от деления $H(x)$ на N.

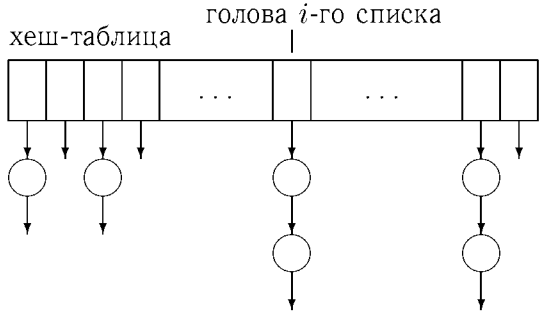
Множество M разбивается на N непересекающихся подмножеств, занумерованных индексами от 0 до $N-1$. Подмножество M_i

с индексом i содержит все элементы x из M, значения хеш-функции для которых равняются i:

$$M_i = \{\, x \in M : h(x) = i \,\}$$

При поиске элемента x сначала вычисляется значение его хеш-функции: $t = h(x)$. Затем мы ищем x в подмножестве M_t. Поскольку это подмножество небольшое, то поиск осуществляется быстро. Для эффективной реализации каждое подмножество должно в большинстве случаев содержать не больше одного элемента. Следовательно, размер хеш-таблицы N должен быть больше, чем среднее число элементов множества. Обычно N выбирают превышающим число элементов множества примерно в три раза. В примере с записной книжкой это означает, что в ней должно быть записано около 10 фамилий. В таком случае большинство страниц либо пустые, либо содержат одну фамилию, хотя не исключены и коллизии, когда на одной странице записаны несколько фамилий.

Мы привели только идею хеш-реализации множества. Различные конкретные схемы по-разному реализуют массив подмножеств и борются с коллизиями. Приведем наиболее универсальную схему, использующую ссылочную реализацию. Каждое подмножество представляется в виде линейного однонаправленного списка, содержащего элементы подмножества. Таким образом, имеемся N списков. Головы всех списков хранятся в одном массиве размера N, который называется хеш-таблицей. Элемент хеш-таблицы с индексом i представляет собой голову i-го списка. Голова содержит ссылку на первый элемент списка, первый элемент — на второй и так далее. Последний элемент в цепочке содержит нулевую ссылку, т.е. ссылку «в никуда». Если список пустой, то элемент хеш-таблицы с индексом i содержит нулевую ссылку.



4.6.5. Циклы «для каждого» и итераторы

Часто бывает необходимо перебрать все элементы структуры данных и для каждого элемента выполнить некоторое действие. Обычно при записи алгоритмов на неформальном языке используется конструкция «цикл для каждого»; например, в случае множества M можно записать следующий фрагмент программы:

```
цикл для каждого элемента x из множества M
| действие(x);
конец цикла
```

В большинстве структур данных такой цикл можно реализовать, пользуясь другими действиями, возможными для структуры этого типа. Например, для списка можно использовать следующий фрагмент программы:

```
установить указатель в начало списка;
цикл пока указатель не в конце списка
| действие(элемент за указателем);
| передвинуть указатель вперед;
конец цикла
```

Множество отличается от всех других структур данных тем, что цикл «для каждого» невозможно смоделировать, пользуясь другими предписаниями множества. Поэтому выполнение такого цикла должно быть обеспечено реализацией множества. Метод построения цикла «для каждого» в сильной степени зависит от используемого языка программирования и от конкретной библиотеки классов или подпрограмм. Как правило, создается некоторый объект, позволяющий последовательно перебирать элементы структуры. Внутреннее устройство такого объекта зависит от структуры данных и скрыто от программиста. Объект такого типа обычно называют «итератором» или «энумератором». С итератором возможны два действия:

1) проверить, есть ли еще элементы структуры данных, которые не были перебраны на предыдущих шагах;

2) получить следующий элемент структуры данных.

Реализация структуры данных должна обеспечить реализацию предписания, которое создает и инициализирует объект типа итератор, предназначенный для перебора элементов структуры.

В случае множества M цикл «для каждого» записывается с помощью итератора примерно так:

```
итератор := начать перебор элементов множества M;
цикл пока итератор.есть еще элементы
| x := итератор.получить следующий элемент множества;
| действие(x);
конец цикла
```

Мы рассмотрели наиболее важные структуры данных, используемые в программировании, и методы их реализации. Конечно, этими примерами не исчерпывается все многообразие структур данных. Однако, большинство реальных примеров либо аналогичны рассмотренным, либо получаются комбинацией нескольких элементарных структур — например, реализация массива множеств может использовать списки или деревья. Искусство программиста состоит в том, чтобы выбрать структуру данных, наиболее подходящую для решаемой задачи, чтобы в ней отсутствовали массовые операции, память расходовалась экономно и поиск выполнялся быстро. И, пожалуй, один из главных критериев — это простота и изящество программы, которая получается в результате.

4.7. Задачи по теме «Структуры данных»

1. Выписать на языке Си реализацию очереди на базе “кольцевого” массива.

2. Реализовать двунаправленный список на базе двух стеков (выписать реализацию на псевдокоде или на Си).

3. Выписать на Си непрерывную реализацию нагруженного множества с помощью бинарного поиска. Элементами множества являются текстовые строки, нагрузками элементов — целые числа. Используя эту реализацию, написать программу, которая находит множество всевозможных слов в заданном тексто-

вом файле, а также количество вхождений каждого слова в текст.

4. Рассмотрим бинарное упорядоченное дерево, в вершинах которого содержатся целые числа. Выписать на Си или на псевдокоде следующие алгоритмы:

   (i) поиск минимальной вершины дерева;
   (ii) поиск максимальной вершины дерева;
   (iii) для заданной вершины найти следующую по порядку;
   (iv) напечатать по порядку все значения в вершинах;
   (v) поиск вершины с заданным значением. Если такой вершины нет, то найти родительскую вершину;
   (vi) добавление к дереву вершины с заданным значением с сохранением упорядоченности;

5. Та же задача, что и в пункте 3, для случая реализации множества на базе хеш-функции и массива списков (эта реализация была рассмотрена в разделе 4.6.4).

6. Реализовать множество на Си или псевдокоде с помощью хеш-функции, используя следующий механизм разрешения коллизий:

   - хеш-таблица представляет собой обычный массив элементов того же типа, что и элементы множества;
   - помимо хеш-таблицы, используется целочисленный массив "мощность слоя" такого же размера. В его ячейке с индексом i хранится общее число элементов множества, значение хеш-функции на которых равно i;
   - используется также массив "остаток" достаточно большой длины, тип его элементов совпадает с типом множества;
   - если в множестве есть только один элемент со значением хеш-функции, равным i, то он хранится в i-й ячейке хеш-таблицы. В противном случае, т.е. при коллизии, первый элемент помещается в i-ю ячейку хеш-таблицы, а остальные элементы со значением хеш-функции, равным i, записываются в массив "остаток" в произвольном порядке.

Литература

[1] Д. Кнут. Искусство программирования для ЭВМ. Т. 1–3. — Пер. с англ. — М.: Мир, 1976–1978.

[2] Д. Кнут. Все про TEX. — Пер. с англ. — Протвино, РДTEX, 1993.

[3] А. Г. Кушниренко, Г. В. Лебедев. Программирование для математиков: Учебное пособие для вузов. — М., Наука, 1988. — 384 с.

[4] А. П. Ершов, А. Г. Кушниренко, Г. В. Лебедев, А. Л. Семенов, А. Х. Шень. Основы информатики и вычислительной техники. — М.: Просвещение, 1988.

[5] А. Г. Кушниренко, Г. В. Лебедев, Р. А. Сворень. Основы информатики и вычислительной техники. — М.: Просвещение, 1990, 1991, 1993, 1996.

[6] Б. Керниган, Д. Ритчи. Язык программирования Си. — Пер. с англ. — М.: Финансы и статистика, 1992.

[7] Б. Страуструп. Язык программирования Си++. — Пер. с англ. — М.: «Радио и связь», 1991. — 352 с.

[8] Б. Страуструп. Язык программирования C++ (третье издание). — Пер. с англ. — СПб., М.: «Невский диалект». Издательство «Бином», 1999.

[9] П. Нотон. Java. Справочное руководство. — Пер. с англ. — М.: Восточная Книжная Компания, 1996. — 448 с.

[10] Б. Э. Смит, М. Т. Джонсон. Архитектура и программирование микропроцессора INTEL 80386. — Пер. с англ. — М.: Конкорд, 1992. — 334 с.

[11] Ф. Доймлинг, Д. Силлеску. Язык программирования PostScript. — Пер. с нем. — М.: Физматлит, 1993. — 136 с.

[12] Н. Вирт. Алгоритмы + структуры данных = программы. — Пер. с англ. — М.: Мир, 1985.

[13] Н. Вирт. Программирование на языке Модула-2. — Пер. с англ. — М.: Мир, 1987. — 224 с.

Предметный указатель

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное бесплатное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Главными целями и задачами проекта являются:

- финансирование разработок учебных курсов по тематике информационно-коммуникационных технологий;
- координация учебно-методической деятельности предприятий компьютерной индустрии по созданию учебных курсов;
- обеспечение профессорско-преподавательских кадров вузов и их библиотек учебниками и методическими материалами;
- содействие органам государственной власти в области развития образовательных программ, связанным с современными информационными технологиями.

В рамках проекта выпускается серии учебников «Основы информационных технологий» и «Основы информатики и математики», в каждой из которых предполагается выпуск более 100 книг по различным учебным дисциплинам. Над их созданием работают авторские коллективы профессоров и преподавателей из ведущих российских вузов, а также представителей бизнеса и академической среды.

Учебные планы курсов разрабатываются с учетом российских государственных образовательных стандартов, рекомендаций профильных международных организаций и современных требований бизнеса. Наиболее важные образовательные курсы создаются при участии и поддержке ведущих иностранных и отечественных компаний, а также общественных организаций и коммерческих ассоциаций в области информационных технологий.

Все учебные курсы выкладываются в открытый и бесплатный доступ во Всемирную сеть в полном виде и могут быть использованы как для изучения студентами, так и в целях формирования учебных программ преподавателями. Любое использование учебных курсов свободно и не требует никаких дополнительных или специальных разрешений со стороны Интернет-университета информационных технологий.

Интернет-сайт www.intuit.ru дает возможность самостоятельного изучения учебных курсов с тестированием и сдачей экзаменов в дистанционном режиме.

Добро пожаловать
в Интернет-Университет Информационных Технологий!

Серия «Основы информатики и математики»

1. **Преподавание информатики и математических основ информатики**, под. ред. А.В. Михалева, 2005, 144 с., ISBN 5-9556-0037-X.
2. **Начала алгебры, часть I**, А.В. Михалев, А.А. Михалев, 2005, 272 с., ISBN 5-9556-0038-8.
3. **Основы программирования**, В.В. Борисенко, 2005, 328 с., ISBN 5-9556-0039-6.
4. **Работа с текстовой информацией. Microsoft Office Word 2003**, О.Б. Калугина, В.С. Люцарев, 2005, 264 с., ISBN 5-9556-0040-0.

Серия «Основы информационных технологий»

1. **Основы Web-технологий**, П.Б. Храмцов и др., 2003, 512 с., ISBN 5-9556-0001-9.
2. **Основы сетей передачи данных**, В.Г. Олифер, Н.А. Олифер, 2005, 176 с., ISBN 5-9556-0035-3.
3. **Основы информационной безопасности**, 2-е издание, В.А. Галатенко, 2004, 264 с., ISBN 5-9556-0015-9.
4. **Основы микропроцессорной техники**, 2-е издание, Ю.В. Новиков, П.К. Скоробогатов, 2004, 440 с., ISBN 5-9556-0016-7.
5. **Язык программирования Си++**, 2-е издание, А.Л. Фридман, 2004, 264 с., ISBN 5-9556-0017-5.
6. **Программирование на Java**, Н.А. Вязовик, 2003, 592 с., ISBN 5-9556-0006-X.
7. **Стандарты информационной безопасности**, В.А. Галатенко, 2004, 328 с., ISBN 5-9556-0007-8.
8. **Основы функционального программирования**, Л.В. Городняя, 2004, 280 с., ISBN 5-9556-0008-6.
9. **Программирование в стандарте POSIX**, В.А. Галатенко, 2004, 560 с., ISBN 5-9556-0011-6.
10. **Введение в теорию программирования**, С.В. Зыков, 2004, 400 с., ISBN 5-9556-0009-4.
11. **Основы менеджмента программных проектов**, И.Н. Скопин, 2004, 336 с., ISBN 5-9556-0013-2.
12. **Основы операционных систем**, В.Е. Карпов, К.А. Коньков, 2004, 632 с., ISBN 5-9556-0012-4.
13. **Основы SQL**, Л.Н. Полякова, 2004, 368 с., ISBN 5-9556-0014-0.

14. **Архитектуры и топологии многопроцессорных вычислительных систем**,
А.В. Богданов, В.В. Корхов, В.В. Мареев, Е.Н. Станкова, 2004, 176 с., ISBN 5-9556-0018-3.
15. **Операционная система UNIX**,
Г.В. Курячий, 2004, 320 с., ISBN 5-9556-0019-1.
16. **Основы сетевой безопасности: криптографические алгоритмы и протоколы взаимодействия**,
О.Р. Лапонина, 2005, 608 с., ISBN 5-9556-0020-5.
17. **Программирование в стандарте POSIX. Часть 2**,
В.А. Галатенко, 2005, 384 с., ISBN 5-9556-0021-3.
18. **Интеграция приложений на основе WebSphere MQ**,
В.А. Макушкин, Д.С. Володичев, 2005, 336 с., ISBN 5-9556-0031-0.
19. **Стили и методы программирования**,
Н.Н. Непейвода, 2005, 320 с., ISBN 5-9556-0023-X.
20. **Основы программирования на PHP**,
Н.В. Савельева, 2005, 264 с., ISBN 5-9556-0026-4.
21. **Основы баз данных**,
С.Д. Кузнецов, 2005, 488 с., ISBN 5-9556-0028-0.
22. **Интеллектуальные робототехнические системы**,
В.Л. Афонин, В.А. Макушкин, 2005, 208 с., ISBN 5-9556-0024-8.
23. **Программирование на языке Pascal**,
Т.А. Андреева, 2005, 240 с., ISBN 5-9556-0025-6.
24. **Основы тестирования программного обеспечения**,
В.П. Котляров, 2005, 360 с., ISBN 5-9556-0027-2.
25. **Программирование на языке Си**
Н.И. Костюкова, Н.А. Калинина, 2005, 224 с., ISBN 5-9556-0026-4.
26. **Основы локальных сетей**,
Ю.В. Новиков, С.В. Кондратенко, 2005, 360 с., ISBN 5-9556-0032-9.
27. **Операционная система Linux**,
Г.В. Курячий, К. Маслинский, 2005, 400 с., ISBN 5-9556-0029-9.
28. **Проектирование информационных систем**,
В.И. Грекул и др., 2005, 296 с., ISBN 5-9556-0033-7.
29. **Основы программирования на языке Пролог**,
П.А. Шрайнер, 2005, 176 с., ISBN 5-9556-0034-5.
30. **Операционная система Solaris**,
Ф.И. Торчинский, 2005, 472 с., ISBN 5-9556-0022-1.

Книги издательства Интернет-Университета Информационных Технологий всегда можно заказать на сайте: **shop.intuit.ru**
По вопросам оптовых закупок звоните **(095) 253-9312**.
Адрес: Россия, Москва 123056, Электрический пер., дом 8, строение 3.

ОСНОВЫ ИНФОРМАТИКИ И МАТЕМАТИКИ

В.В. Борисенко
Основы программирования

Корректор С. Перепелкина
Компьютерная верстка А. Пентус, **В. Б**орисенко
Обложка М. Автономова

Формат 60x90 $^{1}/_{16}$. Усл. печ. л. 20,5. Бумага офсетная.
Подписано в печать 26.04.2005. Тираж 2000 экз. Заказ № .

Санитарно-эпидемиологическое заключение о соответствии санитарным правилам №77.99.02.953.Д.006052.08.03 от 12.08.2003

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
123056, Москва, Электрический пер., 8, стр.3.

Отпечатано с готовых диапозитивов на ФГУП ордена «Знак Почета»
Смоленская областная типография им. В.И.Смирнова.
Адрес: 214000, г. Смоленск, проспект им.Ю.Гагарина, д.2.