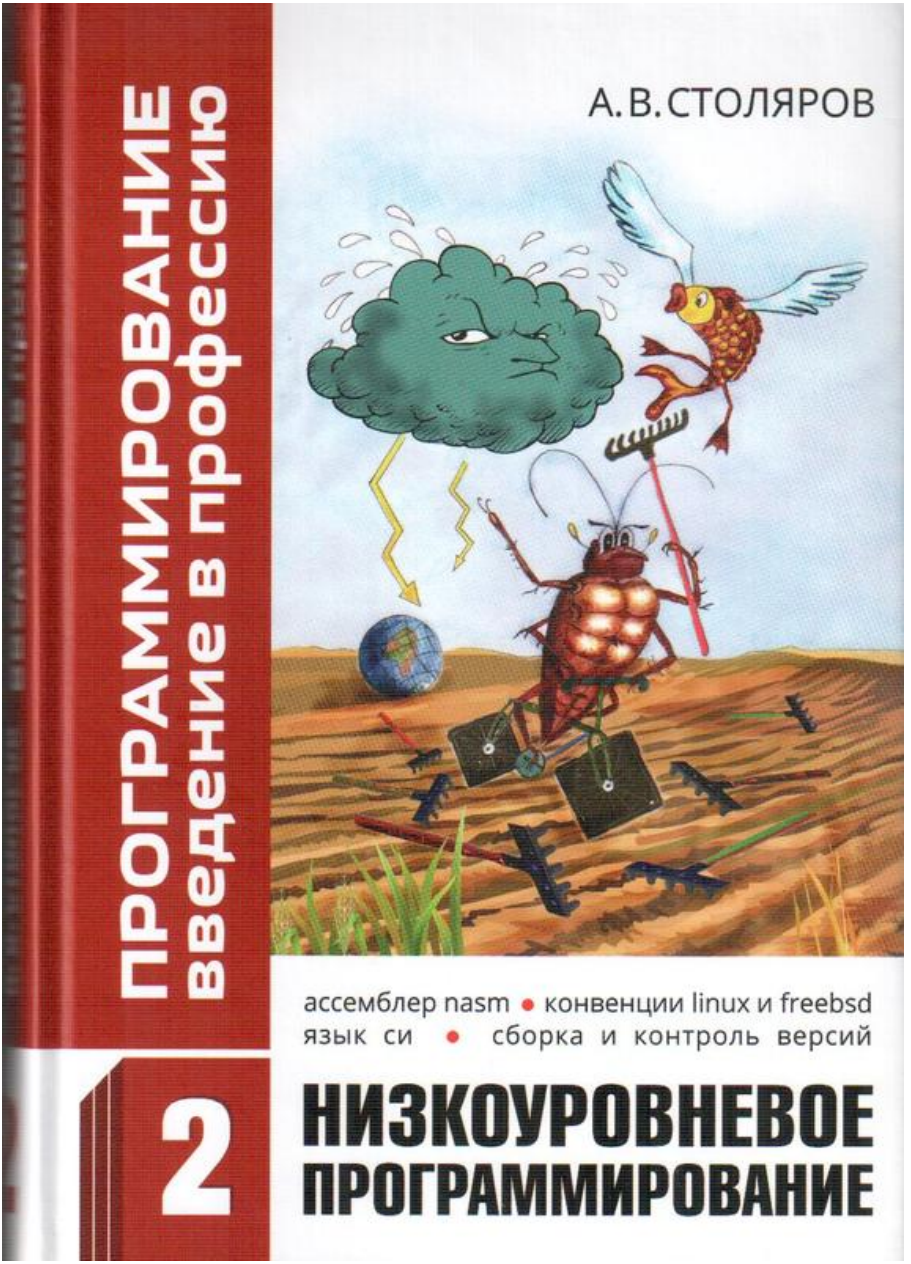

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Программирование: введение в профессию. II: Низкоуровневое программирование», опубликованное в издательстве МАКС Пресс в 2016 году, называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей, а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные файлообменные сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. СТОЛЯРОВ

ПРОГРАММИРОВАНИЕ
ВВЕДЕНИЕ В ПРОФЕССИЮ

II: НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

МАКС ПРЕСС

Москва — 2016

УДК 519.683+004.4
ББК 32.973.26-018.1
С81

Столяров А. В.

С81 Программирование: введение в профессию. II: Низкоуровневое программирование. – М.: МАКС Пресс, 2016. – 496 с.
ISBN 978-5-317-05301-7

Вашему вниманию предлагается второй том учебника «Программирование: введение в профессию», все части которого объединены использованием Unix-систем в качестве единой учебной операционной среды. Учебник ориентирован в основном на самостоятельное изучение программирования.

В том вошли части, посвящённые программированию на уровне машинных команд (на языке ассемблера) и на языке Си. Текст предполагает использование аппаратной платформы i386, ассемблера NASM, компилятора `gcc`, отладчика `gdb`. В конце четвёртой части приведены сведения о системе автоматической сборки Gnu Make, программе динамического анализа программ Valgrind, о системах контроля версий CVS и `git`.

Для школьников, студентов, преподавателей и всех, кто интересуется программированием.

УДК 519.683+004.4
ББК 32.973.26-018.1

ISBN 978-5-317-05301-7

Оглавление

Содержание

Предисловие ко второму тому

Планировалось, что в этот том войдут части, посвящённые языку ассемблера, языку Си, взаимодействию с операционной системой и программированию с разделяемыми данными. Как это часто бывает, объём текста получился гораздо больше, чем предполагалось; довольно быстро стало понятно, что в одну книжку все эти части сугубо физически не помещаются. Том, который вы сейчас читаете, в итоге составлен всего из двух частей, а остальное войдёт в следующий, третий том; к сожалению, сейчас невозможно даже приблизительно назвать сроки его появления. Более того, по состоянию на момент написания этого текста не вполне понятно даже, *сколько* томов в итоге получится — три или четыре.

Текст части, посвящённой языку ассемблера, ранее уже публиковался отдельной книгой; в новую книгу он вошёл после заметной, но не слишком радикальной переработки. Часть, посвящённая языку Си, была написана с нуля.

Публикуя книгу, целиком посвящённую низкоуровневому программированию, я считаю уместным в её предисловии сказать несколько слов для тех, кто сомневается в необходимости изучения низкого уровня как такового. Разница между программистами, умеющими программировать на языке ассемблера и на Си, и теми, кто этого не умеет, *на самом деле* есть разница между теми, кто понимает, что делает, и теми, кто этого не понимает. Утверждение, что в современных условиях «можно и без этого», отчасти верно: среди людей, получающих деньги за написание программ, можно найти и тех, кто не умеет работать с указателями, и тех, кто не знает машинного представления целых чисел, и тех, кто не понимает слова «стек»; верно и то, что все эти люди находят для себя вполне достойно оплачиваемые позиции. Всё это так; но делать из этого вывод о «ненужности» низкоуровневого программирования было бы по меньшей мере странно. Возможность писать программы, не понимая до конца собственных действий, создаётся программным обеспечением, которое само по себе в таком стиле написано быть не может; это программное обеспечение, обычно называемое *системным*, тоже, очевидно, кто-то должен разрабатывать. И уж совсем нелепым кажется утверждение о том, что-де «системщиков много не нужно»: квалифицированных людей объективно *не хватает*, то есть спрос на них превышает предложение, так что нужно их, во всяком случае, больше, чем их есть; ну а то, что их нужно в целом меньше, чем тех, кому высокая квалификация в их работе не нужна, здесь вообще не

имеет никакого отношения к делу, ведь важно соотношение спроса и предложения, а не размер спроса как таковой.

Обучаемый может «не потянуть» программирование на уровне машины и уйти в веб-разработку, компьютерную поддержку бизнес-процессов и прочие подобные области, но ведь это не повод изначально не пытаться никого учить серьёзному программированию. Работу на языке ассемблера и на Си объединяет, помимо прочего, один очень важный момент: ни то, ни другое совершенно невозможно делать, не имея досконального понимания происходящего. Мне остаётся лишь надеяться, что предлагаемый вашему вниманию второй том моего учебника может такое понимание дать.

Оба тома этой книги обязаны своим появлением людям, которые откликнулись на мою просьбу о поддержке проекта и своими пожертвованиями сформировали его бюджет. Ниже приведён полный (на 9 июня 2016 г.) список донэйторов, кроме тех, кто пожелал сохранить инкогнито; каждый участник включён в список под таким именем, которое указал сам:

Gremlin, Grigoriy Kraynov, Шер Арсений Владимирович, Таранов Василий, Сергей Сетченков, Валерия Шакирзянова, Катерина Галкина, Илья Лобанов, Сюзана Тевдорадзе, Иванова Оксана, Куликова Юлия, Соколов Кирилл Владимирович, jeckep, Кулёва Анна Сергеевна, Ермакова Марина Александровна, Переведенцев Максим Олегович, Костарев Иван Сергеевич, Донцов Евгений, Олег Французов, Степан Холопкин, Попов Артём Сергеевич, Александр Быков, Белобородов И. Б., Ким Максим, artyrian, Игорь Эльман, Илюшкин Никита, Кальсин Сергей Александрович, Евгений Земцов, Шрамов Георгий, Владимир Лазарев, eupharina, Николай Королев, Горошевский Алексей Валерьевич, Леменков Д. Д., Forester, say42, Аня «canja» Ф., Сергей, big_fellow, Волканов Дмитрий Юрьевич, Танечка, Татьяна 'Vikora' Алпатова, Беляев Андрей, Лошкины (Александр и Дарья), Кирилл Алексеев, kopish32, Екатерина Глазкова, Олег «burunduk3» Давыдов, Дмитрий Кронберг, yobibyte, Михаил Аграновский, Александр Шепелёв, G.Nerc=Y.uR, Василий Артемьев, Смирнов Денис, Pavel Korzhenko, Руслан Степаненко, Терешко Григорий Юрьевич 15e65d3d, Lothlorien, vasiliandets, Максим Филиппов, Глеб Семёнов, Павел, unDEFER, kilolife, Арбичев, Рябинин Сергей Анатольевич, Nikolay Ksenev, Кучин Вадим, Мария Вихрева, igneus, Александр Чернов, Roman Kurynin, Власов Андрей, Дергачёв Борис Николаевич, Алексей Алексеевич, Георгий Мошкин, Владимир Руцкий, Федулов Роман Сергеевич, Шадрин Денис, Панфёров Антон Александрович, os80, Зубков Иван, Архипенко Константин Владимирович, Асирян Александр, Дмитрий С. Гуськов, Тойгильдин Владислав, Masutacu, D.A.X., Каганов Владислав, Анастасия Назарова, Гена

Иван Евгеньевич, Линара Адылова, Александр, izin, Николай Подонин, Юлия Корухова, Кузьменкова Евгения Анатольевна, Сергей «GDM» Иванов, Андрей Шестимеров, vap, Грацианова Татьяна Юрьевна, Меньшов Юрий Николаевич, nvasil.

Текст построен в предположении, что с материалом первого тома читатель знаком. Программирование на языке ассемблера можно изучать, если вы хотя бы примерно представляете, как работает компьютер (так что первый том для этого изучать не обязательно), но вот приступать к программированию на Си, не понимая указатели, лучше не надо — ничего хорошего из этого не получится. Как показывает мой преподавательский опыт, если на выходе нужно получить умение работать на Си, то эффективнее (несмотря на кажущиеся потери времени) сначала изучить Паскаль, освоить (именно на Паскале!) работу со списками, и только после этого перейти на Си.

Как и раньше, в тексте встречаются фрагменты, набранные уменьшенным шрифтом без засечек. При первом прочтении книги такие фрагменты можно безболезненно пропустить; некоторые из них могут содержать ссылки вперёд и предназначаться для читателей, уже кое-что знающих о программировании. Примеры того, **как не надо делать**, помечены вот таким знаком на полях:

Вводимые новые понятия выделены ***жирным курсивом***. Кроме того, в тексте используется *курсив* для смыслового выделения и **жирный шрифт** для выделения фактов и правил, которые желательно не забывать, иначе могут возникнуть проблемы с последующим материалом. Домашняя страница этой книги в Интернете расположена по адресу

`http://www.stolyarov.info/books/programming_intro`

Здесь вы можете найти архив примеров программ, приведённых в книге, а также электронную версию самой книги. Для примеров, включённых в архив, в тексте книги указаны имена файлов.

Часть 3

Возможности процессора и язык ассемблера

Эта часть нашей книги будет посвящена программированию на языке ассемблера NASM — как обычно, в среде ОС Unix. Между тем подавляющее большинство профессиональных программистов, услышав о таком, лишь усмехнётся и задаст риторический вопрос: «Да кто же пишет под Unix на ассемблере? На дворе ведь XXI век!» Самое интересное, что они будут совершенно правы. В современном мире программирование на языке ассемблера оказалось вытеснено даже из такой традиционно «ассемблерной» области, как программирование микроконтроллеров — маленьких однокристалльных ЭВМ, предназначенных для встраивания во всевозможную технику, от стиральных машин и сотовых телефонов до самолётов и турбин на электростанциях. В большинстве случаев прошивки микроконтроллеров сейчас пишут на языке Си, и лишь небольшие вставки выполняют на языке ассемблера; то же самое верно и для ядер операционных систем, и для других задач, в которых необходима привязка к возможностям конкретного процессора.

Конечно, совсем обойтись без фрагментов на языке ассемблера пока не получается. Отдельные ассемблерные модули, а равно и ассемблерные вставки в текст на других языках присутствуют и в ядрах операционных систем, и в системных библиотеках того же языка Си (и других языков высокого уровня); в особых случаях программисты микроконтроллеров тоже вынуждены отказываться от Си и писать «на ассемблере», чтобы, например, сэкономить дефицитную память. Так, достаточно популярный микроконтроллер ATtiny4 имеет всего 16 байт оперативной памяти и 512 байт псевдопостоянной памяти для хранения кода программы; в таких условиях языку ассемблера практически нет альтернативы. Однако такие случаи редки даже в мире микроконтроллеров, большинство из которых предоставляет программисту далеко не столь жёсткие

условия. Мало кому из вас, изучающих ныне программирование на языке ассемблера, придётся хотя бы один раз за всю жизнь прибегнуть к этим навыкам на практике.

Так зачем же тратить время на изучение ассемблера? Ведь всё равно это никогда не пригодится? Так это выглядит лишь на первый взгляд; при более внимательном рассмотрении вопроса умение мыслить в терминах машинных команд не просто «пригодится», оно оказывается жизненно необходимо любому профессиональному программисту, даже если этот программист никогда не пишет на языке ассемблера. На каком бы языке вы ни писали свои программы, необходимо хотя бы примерно представлять, что конкретно будет делать процессор, чтобы исполнить вашу высочайшую волю. Если такого представления нет, программист начинает бездумно применять все доступные операции, не ведая, что *на самом деле* творит. Между тем одно присваивание на хорошо знакомом нам Паскале может выполниться за миллиардную долю секунды, но может и растянуться на заметное время, если, например, нам придёт в голову присваивать друг другу большие массивы строк. С более сложными языками программирования дела обстоят ещё интереснее: присваивание, записанное на языке Си++, может выполниться в одну машинную команду, а может повлечь *миллионы* команд[1]. Два таких присваивания записываются в программе совершенно одинаково (знаком равенства), но этот факт никак нам не поможет: мы не сможем адекватно оценить ресурсоёмкость той или иной операции, не понимая, как и что при этом делает процессор. Программист, не имеющий опыта работы на уровне команд процессора, попросту не ведает, что *на самом деле* творит; вставляя в программу на языке высокого уровня те или иные операции, он часто не догадывается, сколь сложную задачу ставит перед процессором. На выходе мы имеем огромные программы, обескураживающие своей низкой эффективностью — например, приложения для автоматизации офисного документооборота, которым «тесно» в четырёх гигабайтах оперативной памяти и для которых оказывается «слишком медленным» процессор, на много порядков превосходящий по быстродействию суперкомпьютеры восьмидесятых годов.

Опыт показывает, что профессиональный пользователь компьютеров, будь то программист или системный администратор, может себе позволить что-то *не знать*, но ни в коем случае не может позволить себе *не понимать*, как устроена вычислительная система на всех её уровнях, от электронных логических схем до громоздких прикладных программ. Не понимая чего-то, мы оставляем в своём тылу место для «ощущения магии»: на каком-то почти подсознательном уровне нам продолжает казаться, что что-то там нечисто и без парочки чародеев с волшебными палочками не обошлось. Такое ощущение для профес-

[1] Для знающих Си++ поясним: что будет, если применить операцию присваивания к объекту типа `list<string>`, содержащему две-три тысячи элементов?

сионала недопустимо категорически: напротив, профессионал обязан понимать (и интуитивно ощущать), что устройство, с которым он имеет дело, создано такими же людьми, как и он сам, и ничего «волшебного» или «непознаваемого» собой не представляет.

Если ставить целью достижение такого уровня понимания, оказывается совершенно не важно, какую конкретную архитектуру и язык какого конкретного ассемблера изучать. Зная один язык ассемблера, вы сможете начать писать на любом другом, потратив два-три часа (а то и меньше) на изучение справочной информации; но главное тут в том, что, умея мыслить в терминах машинных команд, вы всегда будете знать, что в действительности происходит при выполнении ваших программ.

Несмотря на всё вышесказанное, следует, по-видимому, пояснить выбор конкретной архитектуры. Материал нашей «ассемблерной» части книги основан на системе команд процессоров семейства x86, причём мы будем использовать 32-битный вариант этой архитектуры, так называемую систему команд i386. На момент написания этого текста 32-битные компьютеры семейства x86 уже почти полностью вытеснены компьютерами на основе 64-битных процессоров[2], но, к счастью, эти процессоры могут выполнять программы в 32-битном режиме. Саму 64-битную систему команд мы изучать не будем, и тому есть определённая причина. Все имеющиеся описания этой системы строятся по принципу перечисления её отличий от 32-битного случая; получается, что нам в любом случае сначала нужно изучить 32-битную систему команд. Но изучив её, мы уже достигнем своей цели — *получим опыт работы на языке ассемблера*; дальнейший переход к 64-битному случаю возможен, но для наших целей несколько избыточен. Даже 32-битную систему команд мы будем изучать далеко не во всём её (кошмарном) великолепии: нам хватит примерно десятой доли возможностей изучаемого процессора, чтобы иметь возможность писать программы.

Уместно будет сказать несколько слов относительно выбора конкретного ассемблера. Как известно, для работы с процессорами семейства x86 используется два основных подхода к синтаксису языка ассемблера — это синтаксис AT&T и синтаксис Intel. Одна и та же команда процессора представляется в этих синтаксических системах совершенно по-разному: например, команда, в синтаксисе Intel выглядящая как

```
mov eax, [a+edx]
```

в синтаксисе AT&T будет записываться следующим образом:

```
movl a(%edx), %eax
```

[2] Автор считает уместным заметить, что значительная часть этого текста готовилась к печати на EEEPC-901; этот нетбук оснащён одноядерным 32-битным процессором, которого, тем не менее, хватает с запасом для решения всех задач, возникающих у автора в обычных условиях.

В среде ОС Unix традиционно более популярен именно синтаксис AT&T, но в применении к поставленной учебной задаче это создаёт некоторые проблемы. Учебные пособия, ориентированные на программирование на языке ассемблера в синтаксисе Intel, всё-таки существуют, тогда как синтаксис AT&T описывается исключительно в специальной (справочной) технической литературе, не имеющей целью обучение. Кроме того, необходимо учитывать и многолетнее господство среды MS DOS в качестве платформы для аналогичных учебных курсов; всё это позволяет назвать синтаксис Intel существенно более привычным для преподавателей (да и для некоторых студентов, как ни странно, тоже) и лучше поддерживаемым. Для Unix доступно два основных ассемблера, поддерживающих синтаксис Intel: это NASM («Netwide Assembler»), разработанный Саймоном Тетхемом и Джулианом Холлом, и FASM («Flat Assembler»), созданный Томашем Гриштаром. Сделать однозначный выбор между этими ассемблерами достаточно сложно. В нашей книге рассматривается язык ассемблера NASM, в том числе и специфические для него макросредства; такой выбор не обусловлен никакими серьёзными причинами и попросту случаен.

3.1. Вводная информация

С понятием ***ассемблера*** мы уже знакомы из первого тома (см. §1.6.7); там же мы успели обсудить основные принципы устройства компьютера и то, как процессор обрабатывает программы. Возможно, сейчас самое время вернуться к этому параграфу и перечитать его; более того, мы рекомендовали бы освежить в памяти всю главу 1.6.

Прежде чем мы начнём программировать на выбранном языке ассемблера, нам придётся остановиться ещё на некоторых тонких моментах, без которых будет сложно понять происходящее. В этой главе мы обсудим особенности окружения, в котором будут выполняться наши программы, сделаем небольшой экскурс в историю используемого семейства процессоров, после чего, чтобы познакомиться с новым инструментом, напишем простую программу и заставим её заработать.

3.1.1. Классические принципы выполнения программ

Для начала напомним то, что уже обсуждалось во введении. Принципы построения вычислительных машин, известные как ***архитектурные принципы фон Неймана***, предполагают, что:

- основу компьютера составляют ***центральный процессор*** (электронная схема, выполняющая вычисления) и ***оперативная память*** (электронное устройство, обеспечивающее хранение ин-

формации и способное к непосредственному взаимодействию с центральным процессором);

- оперативная память состоит из ***ячеек памяти***; каждая ячейка способна хранить («помнить») число из определённого диапазона (в частности, подавляющее большинство современных компьютерных архитектур использует ячейки размером в 8 двоичных разрядов; такая ячейка способна хранить число от `0` до `255`); все ячейки памяти имеют одинаковое устройство и одинаковый размер (принцип однородности памяти);
- центральный процессор в любой момент может *записать* число из этого диапазона в любую из ячеек, а также *прочитать* содержимое любой ячейки, т. е. узнать, какое число там хранится (принцип прямого доступа к памяти);
- с точки зрения центрального процессора ячейки памяти различаются только номерами — так называемыми ***адресами*** (принцип линейности памяти);
- центральный процессор автоматически выполняет одну за другой операции, предписанные ***программой*** (принцип программного управления);
- программа хранится в ячейках оперативной памяти в виде ***машинных инструкций*** — чисел, представляющих собой кодовые обозначения операций, которые следует выполнить (принцип хранимой программы);
- ячейка памяти сама по себе «не знает», относится ли хранящееся в ней число к коду программы или это просто некие данные (принцип неразличимости команд и данных).

Неразличимость команд и данных позволяет трактовать программы как данные и создавать программы, для которых в роли обрабатываемой информации выступают *другие программы*. Более того, ещё четверть века назад на некоторых платформах программы могли *модифицировать сами себя* прямо во время исполнения, но современные вычислительные системы такую возможность исключают.

В составе центрального процессора присутствуют электронные схемы для хранения информации, подобные ячейкам памяти; они называются ***регистрами***. Обычно различают ***регистры общего назначения*** и ***служебные регистры***. Регистры общего назначения предназначены для краткосрочного размещения обрабатываемых данных; операции с ними выполняются на порядки быстрее, чем с ячейками памяти, но совокупный объём регистров может быть в миллионы, а в современных условиях — в миллиарды раз меньше объёма памяти, поэтому в регистрах обычно располагаются исходные данные и промежуточные результаты для расчётов, выполняемых прямо сейчас. По окончании того или иного расчёта данные переносят в оперативную память, чтобы освободить регистры для других целей.

Служебные регистры содержат информацию, необходимую самому процессору для организации выполнения программы. Важнейший из служебных регистров — ***указатель инструкции***, иногда называемый также ***счётчиком команд***[3]; в этом регистре содержится *адрес той ячейки памяти, откуда процессору нужно будет извлечь код следующего действия.*

Центральный процессор работает, бесконечно повторяя ***цикл выполнения команд***, состоящий из трёх шагов:

- извлечь код очередной машинной команды из памяти, начиная с ячейки, адрес которой сейчас находится в *указателе инструкции*;
- увеличить значение *указателя инструкции* на длину извлечённого кода[4], после чего регистр будет содержать адрес инструкции, *следующей за текущей*;
- дешифровать извлечённый из памяти код команды и выполнить соответствующее этому коду действие.

Почему-то многих студентов на экзамене ставит в тупик вопрос, откуда центральный процессор знает, какую именно машинную команду (из миллионов команд, находящихся в памяти) нужно выполнить прямо сейчас; правильный ответ совершенно тривиален — адрес нужной машинной команды находится в *указателе инструкции.* Процессор никоим образом не пытается вникнуть в логику выполняемой программы, в то, какими должны оказаться результаты программы в целом; он лишь следует раз и навсегда установленному циклу: считать код, увеличить указатель инструкции, выполнить инструкцию, начать сначала. Автоматическое увеличение адреса в регистре указателя инструкции приводит к тому, что машинные команды, составляющие программу, выполняются одна за другой в той последовательности, в которой они записаны в программе.

Когда указатель инструкции содержит адрес того или иного места в оперативной памяти, говорят, что в этом месте памяти (или, что то же самое, на данном участке программы) *находится управление.* Логика этого термина основана на том, что действия процессора подчинены машинным командам (*управляются* ими), при этом очередную команду процессор берёт из памяти по адресу из указателя инструкции.

Для организации хорошо знакомых нам ***ветвлений***, ***циклов*** и ***вызовов подпрограмм*** применяются машинные команды, принудительно изменяющие содержимое указателя инструкции, в результате чего последовательность команд нарушается, а выполнение программы продолжается с другого места — с той инструкции, чей адрес занесён в регистр. Это называется ***переходом*** или ***передачей управления***

[3]Как уже отмечалось в первом томе, термин не вполне удачен, поскольку на самом деле этот «счётчик» ничего не считает.

[4]В частности, коды команд процессора, который мы будем изучать, могут занимать от 1 до 15 ячеек.

(на другой участок машинного кода). Отметим, что рассмотренный выше цикл выполнения команды предполагает *сначала увеличить указатель инструкции, а потом уже выполнить команду*, так что если очередная команда производит передачу управления, она записывает в указатель инструкции новый адрес *поверх* уже находящегося там адреса, вычисленного в ходе автоматического увеличения.

Мы уже обсуждали переходы во вводной части нашей книги (см. т. 1, стр. 55). Там же говорилось, что команды передачи управления бывают ***безусловными*** и ***условными***: первые просто заносят заданный адрес в указатель инструкции, тогда как вторые сначала проверяют выполнение того или иного *условия*, и если оно не выполнено, не делают ничего, то есть при этом никакого перехода не происходит, а выполнение продолжается, как обычно, со следующей команды. Именно условные переходы позволяют организовать ветвления, а также циклы, длительность выполнения которых зависит от условий; команды безусловных переходов играют скорее вспомогательную роль, хотя и очень важную.

Большинство процессоров поддерживает также ***переход с запоминанием адреса возврата***, используемый для вызова подпрограмм. При выполнении такого перехода адрес, находящийся в указателе инструкции, сначала сохраняется где-то в памяти[5], и лишь после этого заменяется на новый, как правило — адрес начала машинного кода процедуры или функции. Поскольку к моменту выполнения команды (в данном случае — команды перехода с запоминанием возврата) в указателе инструкции находится уже адрес *следующей* команды, именно он и будет запомнен. Когда подпрограмма завершает работу, она осуществляет ***возврат управления***, то есть помещает в указатель инструкции то значение, которое было запомнено при её вызове, так что в вызвавшей части программы выполнение продолжается с команды, следующей за командой перехода с запоминанием. Обычно такую команду так и называют ***командой вызова***.

Повторим ещё раз, что **под передачей управления понимается принудительное изменение адреса, находящегося в регистре указателя инструкции (счётчика команд)**. Это стоит запомнить.

3.1.2. Особенности программирования под управлением мультизадачных операционных систем

Поскольку мы собираемся запускать написанные нами программы под управлением ОС Unix, не лишним будет заранее описать некоторые особенности таких систем с точки зрения выполняемых программ; эти особенности распространяются не только на Unix и никак не зависят

[5] Как мы увидим позже, для этого используется *аппаратный стек*.

от используемого языка программирования, но при работе на языке ассемблера становятся особенно заметны.

Практически все современные операционные системы позволяют запускать и исполнять несколько программ одновременно. Такой режим работы вычислительной системы, называемый ***мультизадачным***[6], порождает некоторые проблемы, требующие решения со стороны аппаратуры, прежде всего — центрального процессора.

Во-первых, нужно защитить выполняемые программы друг от друга и саму операционную систему от пользовательских программ. Если (пусть даже не по злому умыслу, а по ошибке) одна из выполняемых задач изменит что-то в памяти, принадлежащей другой задаче, скорее всего это приведёт к аварии этой второй задачи, причём найти причину такой аварии окажется принципиально невозможно. Если пользовательская задача (опять-таки по ошибке) внесёт изменения в память операционной системы, это приведёт уже к аварии всей системы, причём, опять-таки, без малейшей возможности разобраться в причинах. Поэтому центральный процессор должен поддерживать механизм ***защиты памяти***: каждой выполняющейся задаче выделяется определённая область памяти, и к ячейкам за пределами этой области задача обращаться не может.

Во-вторых, в мультизадачном режиме пользовательские задачи, как правило, не допускаются к прямой работе с внешними устройствами[7]. Если бы это правило не выполнялось, задачи постоянно конфликтовали бы за доступ к устройствам, и такие конфликты, разумеется, приводили бы к авариям. Чтобы ограничить возможности пользовательской задачи, создатели центрального процессора объявили часть имеющихся машинных инструкций ***привилегированными***. Процессор может работать либо в ***привилегированном режиме***, который также называют ***режимом суперпользователя***, либо в ***ограниченном режиме*** (он же ***режим задачи*** или ***пользовательский режим ЦП***)[8]. В ограниченном режиме привилегированные команды недоступны; в привилегированном режиме процессор может выполнять все имеющиеся инструкции, как обычные, так и привилегированные. Операционная система выполняется, естественно, в привилегиро-

[6]Термин «задача», строго говоря, довольно сложен, но упрощённо задачу можно понимать как программу, которая запущена на выполнение под управлением операционной системы; иначе говоря, при запуске программы в системе возникает задача.

[7]Из этого правила есть исключения, связанные, например, с отображением графической информации на дисплее, но в этом случае устройство должно быть закреплено за одной пользовательской задачей и строго недоступно для других задач.

[8]На самом деле процессор i386 и его потомки имеют не два, а четыре режима, называемые также ***кольцами защиты***, но реально операционные системы используют только нулевое кольцо (высший возможный уровень привилегий) и третье кольцо (низший уровень привилегий).

ванном режиме, а при передаче управления пользовательской задаче переключает режим в ограниченный. **Процессор может вернуться в привилегированный режим только при условии возврата управления операционной системе**; это исключает выполнение в привилегированном режиме кода пользовательских программ. К привилегированным относятся инструкции, осуществляющие взаимодействие с внешними устройствами; также в эту категорию попадают инструкции, используемые для настройки механизмов защиты памяти и некоторые другие команды, влияющие на работу системы в целом. Все такие «глобальные» действия являются прерогативой операционной системы. **При работе под управлением мультизадачной операционной системы пользовательской задаче разрешено лишь преобразовывать информацию в отведённой ей области оперативной памяти. Всё взаимодействие с внешним миром задача производит через обращения к операционной системе.** Даже просто вывести на экран строку задача самостоятельно не может, ей необходимо попросить об этом операционную систему. Такое обращение пользовательской задачи к операционной системе за теми или иными услугами называется ***системным вызовом***. Интересно, что *завершение задачи* тоже способна выполнить только операционная система; это становится очевидно, если вспомнить, что сама задача — это объект операционной системы, именно операционная система загружает в память код программы, выделяет память для данных, настраивает защиту, запускает задачу, обеспечивает выделение ей процессорного времени; при завершении задачи необходимо пометить её память как свободную, прекратить выделение этой задаче процессорного времени и т. п., и сделать это, разумеется, может только операционная система. Таким образом, корректной пользовательской задаче никак не обойтись без системных вызовов, ведь обратиться к операционной системе нужно даже для того, чтобы просто завершиться.

Ещё один важный момент, который необходимо упомянуть перед началом изучения конкретного процессора — это наличие в нашей операционной среде механизма ***виртуальной памяти***. Попробуем понять, что это такое. Как уже говорилось, оперативная память делится на одинаковые по своей ёмкости *ячейки* (в нашем случае каждая ячейка содержит 8 бит данных), и каждая такая ячейка имеет свой порядковый номер. Именно этот номер использует центральный процессор для работы с ячейками памяти через общую шину, чтобы отличать их одну от другой. Назовём этот номер ***физическим адресом*** ячейки памяти. Изначально никаких других адресов, кроме физических, у ячеек памяти не было. В машинном коде программ использовались именно физические адреса, которые называли просто «адресами», без уточняющего слова «физический». С развитием мультизадачного режима работы вычислительных систем оказалось, что в силу целого ряда причин использование

физических адресов *неудобно*. Например, программа в машинном коде, в которой используются физические адреса ячеек памяти, не сможет работать в другой области памяти — а ведь в мультизадачной ситуации может оказаться, что нужная нам область уже занята другой задачей. Есть и другие причины, которые обычно подробно рассматриваются в учебных курсах, посвящённых операционным системам.

В современных процессорах используется два вида адресов. Сам процессор работает с памятью, используя уже знакомые нам физические адреса, но в программах, которые на процессоре выполняются, используются совсем другие адреса — *виртуальные*. ***Виртуальный адрес*** — это число из некоего абстрактного ***виртуального адресного пространства***. На тех процессорах, с которыми мы будем работать, виртуальные адреса представляют собой 32-битные целые числа, то есть виртуальное адресное пространство есть множество целых чисел от 0 до $2^{32} - 1$; адреса обычно записываются в шестнадцатеричной системе, так что адрес может быть числом от `00000000` до `ffffffff`. Важно понимать, что виртуальный адрес совершенно не обязан соответствовать какой-то ячейке памяти. Точнее говоря, *некоторые* виртуальные адреса соответствуют физическим ячейкам памяти, некоторые — не соответствуют, а некоторые адреса и вовсе могут то соответствовать физической памяти, то не соответствовать. Такие соответствия задаются путём настройки центрального процессора, за которую отвечает операционная система. Центральный процессор, получив из очередной машинной инструкции виртуальный адрес, *преобразует* его в адрес физический, по которому обращается к оперативной памяти. Таким образом, мы в программах используем в качестве адресов не физические номера ячеек памяти, а виртуальные (абстрактные) адреса, которые потом уже сам процессор преобразует в настоящие номера ячеек. Устройство в составе процессора, преобразующее виртуальные адреса в физические, называется *memory management unit* (MMU, читается *эм-эм-ю*); это можно перевести как «устройство управления памятью», но обычно аббревиатуру «MMU» не переводят.

Наличие в процессоре MMU позволяет, в частности, каждой программе иметь своё собственное адресное пространство: действительно, никто не мешает операционной системе настроить преобразования адресов так, чтобы один и тот же виртуальный адрес в одной пользовательской задаче отображался на одну физическую ячейку, а в другой задаче — на совсем другую.

Вопросы, связанные с созданием новых операционных систем, мы в нашей книге рассматривать не будем. Вместо этого мы ограничимся рассмотрением возможностей процессора i386, доступных пользовательской задаче, работающей в ограниченном режиме. Более того, даже эти возможности мы рассмотрим не все; дело в том, что операционные системы семейства Unix выполняют пользовательские задачи в так назы-

ваемой ***плоской модели*** адресации памяти, в которой не используется часть регистров и некоторые виды машинных команд. На изучение этих регистров и команд мы не будем тратить время, поскольку всё равно не сможем их применить. Позже в нашем курсе мы подробно рассмотрим механизмы взаимодействия с операционной системой, включая и способы организации системного вызова для систем Linux и FreeBSD; однако пока нам эти механизмы не известны, мы будем осуществлять ввод/вывод и завершение программы с помощью готовых *макросов* — специальных идентификаторов, которые наш ассемблер развернёт в целые последовательности машинных команд и уже в таком виде оттранслирует. Отметим, что к концу этой части книги мы сами научимся при необходимости создавать такие макросы.

3.1.3. История платформы i386

В 1971 году корпорация Intel выпустила в свет семейство микросхем, получившее название MCS-4. Одна из этих микросхем, Intel 4004, которую мы уже упоминали во введении, представляла собой первый в мире законченный центральный процессор на одном кристалле, т. е., иначе говоря, первый в истории ***микропроцессор*** — во всяком случае, из доступных широкой публике. Машинное слово[9] этого процессора составляло четыре бита. Год спустя Intel выпустила восьмибитный процессор Intel 8008, а в 1974 году — более совершенный Intel 8080. Интересно, что 8080 использовал иные коды операций, но при этом программы, написанные на языке ассемблера для 8008, могли быть без изменений оттранслированы и для 8080. Аналогичную «совместимость по исходному коду» конструкторы Intel поддержали и для появившегося в 1978 году 16-битного процессора Intel 8086. Выпущенный годом позже процессор Intel 8088 представлял собой практически такое же устройство, отличающееся только разрядностью внешней шины (для 8088 она составляла 8 бит, для 8086 — 16 бит). Именно процессор 8088 был использован в компьютере IBM PC, давшем начало многочисленному и невероятно популярному[10] семейству машин, до сих пор называемых ***IBM PC-совместимыми*** или просто IBM-совместимыми.

Процессоры 8086 и 8088 не поддерживали защиты памяти и не имели разделения команд на обычные и привилегированные, так что запустить полноценную мультизадачную операционную систему на компьютерах

[9] Напомним, что машинным словом называется порция информации, обрабатываемая процессором в один приём.

[10] Популярность IBM-совместимых машин представляет собой явление весьма неоднозначное; многие другие архитектурные решения, имевшие существенно лучший дизайн, не смогли выжить на рынке, затопленном IBM-совместимыми компьютерами, более дешевыми из-за их массовости. Так или иначе, сейчас ситуация именно такова и существенных изменений пока не предвидится.

с этими процессорами было невозможно[11]. Так же обстояло дело и с процессором 80186, выпущенным в 1982 году. В сравнении со своими предшественниками этот процессор работал гораздо быстрее, поскольку в нём были аппаратно реализованы некоторые операции, выполнявшиеся в предыдущих процессорах микрокодом; тактовая частота тоже возросла. Процессор включал в себя некоторые подсистемы, которые ранее требовалось поддерживать с помощью дополнительных микросхем — такие как контроллер прерываний и контроллер прямого доступа к памяти. Кроме того, система команд процессора была расширена введением дополнительных команд; так, стало возможно с помощью одной команды занести в стек все регистры общего назначения. Адресная шина процессоров 8086, 8088 и 80186 была 20-разрядной, что позволяло адресовать не более 1 Mb оперативной памяти.

В том же 1982 году увидел свет процессор 80286, ставший последним 16-битным процессором в рассматриваемом ряду. Этот процессор поддерживал так называемый «защищённый» режим работы (protected mode) и сегментную модель виртуальной памяти, подразумевающую среди прочих возможностей защиту памяти; четыре ***кольца защиты*** позволили запретить пользовательским задачам выполнение действий, влияющих на систему в целом, что необходимо при работе мультизадачной операционной системы. Адресная шина получила четыре дополнительных разряда, увеличив максимальное количество непосредственно доступной памяти до 16 Mb.

Настоящие мультизадачные операционные системы были созданы лишь для следующего процессора в ряду, 32-разрядного Intel 80386, для краткости обозначаемого просто «i386». Этот процессор, массовый выпуск которого начался в 1986 году, отличался от своих предшественников увеличением регистров до 32 бит, существенным расширением системы команд, увеличением адресной шины до 32 разрядов, что позволяло непосредственно адресовать до 4 Gb физической памяти. Добавление поддержки ***страничной организации виртуальной памяти***, наилучшим образом пригодной для реализации мультизадачного режима работы, завершило картину. Именно с появлением i386 так называемые IBM-совместимые компьютеры наконец стали полноценными вычислительными системами. Вместе с тем i386 полностью сохранил совместимость с предшествующими процессорами своей серии, чем обусловлена достаточно странная на первый взгляд система регистров. Например, универсальные регистры процессоров 8086–80286 назывались `AX`, `BX`, `CX` и `DX` и содержали 16 бит данных каждый; в процессоре i386 и более поздних процессорах линейки имеются регистры,

[11] Мультизадачные системы для этих машин известны, но без защиты памяти, то есть в условиях, когда любая из выполняющихся программ может сделать со всей системой буквально что угодно, практическая применимость таких систем оставалась крайне сомнительной.

содержащие по 32 бита и называющиеся `EAX`, `EBX`, `ECX` и `EDX` (буква `E` означает слово «extended», т. е. «расширенный»), причём младшие 16 бит каждого из этих регистров сохраняют старые названия (соответственно, `AX`, `BX`, `CX` и `DX`). Большинство инструкций работает по-разному для операндов длиной 8 бит, 16 бит и 32 бита, и т. п.

Дальнейшее развитие семейства процессоров x86 вплоть до 2003 года было чисто количественным: увеличивалась скорость, добавлялись новые команды, но принципиальных изменений архитектуры не происходило. В 2001 году альянс компаний Hewllett Packard и Intel выпустил процессор Itanium (Merced), архитектура которого, получившая название IA-64, не имела ничего общего с x86, но включала *эмуляцию* выполнения команд архитектуры i386; эмуляция оказалась слишком медленной для практического применения, а создавать программы и операционные системы для новой архитектуры никто не торопился. В 2003 году компания AMD представила новый процессор, Opteron, архитектура которого стала 64-битным расширением архитектуры x86 подобно тому, как 32-битная i386 стала расширением исходной 16-битной архитектуры процессора 8086. Новая система команд получила название «x86_64». Появление Opteron окончательно добило архитектуру Itanium, которая так и не смогла получить серьёзного распространения, хотя процессоры этой архитектуры выпускаются до сих пор. Впрочем, и сам Intel уже в 2004 году выпустил процессор Xeon, имевший архитектуру x86_64.

Последовавшие за этим «многоядерные» архитектуры представляют собой не более чем количественное развитие, притом в направлении, практически не увеличивающем реальное быстродействие системы. Дело в том, что даже высоко загруженные серверные машины в основном «упираются» в своей производительности не в скорость работы процессора и тем более не в конкуренцию программ за единственный процессор, а скорее в скорость дисковых обменов и работы шины; ни на то, ни на другое «многоядерность» повлиять не в состоянии. Как правило, все ядра, кроме одного, в системе бóльшую часть времени просто простаивают.

Процессоры архитектуры x86_64 могут выполнять 32-битные программы, что существенно облегчает миграцию. В частности, скорее всего компьютер, на котором вы пытаетесь программировать, как раз 64-битный; при этом на нём может быть установлена 32-битная или 64-битная операционная система. Ваши собственные программы на языке ассемблера будут 32-битными, но это никоим образом не помешает их выполнять.

3.1.4. Знакомимся с инструментом

Чтобы писать программы на языке ассемблера, нужно изучить, во-первых, процессор, с которым мы будем работать (пусть даже не все его возможности, но хотя бы некоторую существенную их часть), и,

во-вторых, синтаксис языка ассемблера. К сожалению, здесь возникает определённая проблема: изучать эти две вещи одновременно не получается, но изучать систему команд процессора, не имея никакого представления о синтаксисе языка ассемблера, а равно и изучать синтаксис, не имея представления о системе команд — задача неблагодарная, так что с чего бы мы ни начали, результат получится несколько странный. Мы попробуем пойти иным путём: для начала составим хоть какое-то представление как о системе команд, так и о синтаксисе языка ассемблера, пусть даже это представление будет очень и очень поверхностным, а затем уже приступим к систематическому изучению того и другого.

Сейчас мы напишем работающую программу на языке ассемблера, оттранслируем её и запустим. Поначалу в тексте программы будет далеко не всё понятно; что-то мы объясним прямо сейчас, что-то оставим до более подходящего момента. Задачу мы для себя выберем очень простую: напечатать (т. е. вывести на экран, или, если говорить строго, *вывести в поток стандартного вывода*)[12] пять раз слово «Hello». Как мы уже говорили на стр. 22, для вывода строки на экран, а также для корректного завершения программы нам потребуется обращаться к операционной системе, но мы пока воспользуемся для этого уже готовыми *макросами*, которые описаны в отдельном файле. Ассемблер, сверяясь с этим файлом и с нашими указаниями, преобразует каждое использование такого макроса во фрагмент кода на языке ассемблера и сам же эти фрагменты затем оттранслирует.

Вводя в первом томе нашей книги понятие ассемблера (см. т. 1 §1.6), мы отметили, что машинные команды в языке ассемблера обозначаются удобными для запоминания короткими словами, так называемыми ***мнемониками***. Кроме мнемоник, в программах на языке ассемблера встречаются также ***директивы***, то есть прямые приказы ассемблеру, и ***макровызовы***, задействующие возможности макросов. В нашей первой программе будет не так много мнемоник, директивы и макровызовы займут в ней больше места. Итак, пишем:

```
%include "stud_io.inc"
global  _start

section .text
_start: mov     eax, 0
again:  PRINT   "Hello"
        PUTCHAR 10
        inc     eax
```

[12] Отметим, что процессор сам по себе ничего не знает о выводе на экран, все операции ввода-вывода требуют работы с внешними устройствами и организуются операционной системой, она же предоставляет нашей задаче абстрактные «стандартные потоки ввода-вывода».

```
cmp     eax, 5
jl      again
FINISH
```

Попробуем теперь кое-что объяснить. Первая строчка программы содержит директиву `%include`, которая предписывает ассемблеру вставить на место самой директивы всё содержимое некоторого файла, в данном случае — файла `stud_io.inc`. Этот файл также написан на языке ассемблера и содержит описания макросов `PRINT`, `PUTCHAR` и `FINISH`, которые мы будем использовать соответственно для печати строки, для перехода на следующую строку на экране и для завершения программы. Увидев и выполнив директиву `%include`, ассемблер прочитает файл с описаниями макросов, в результате чего мы сможем их использовать.

Важно отметить, что директива `%include` обязательно должна стоять в тексте программы *раньше*, чем там встретятся имена макросов. Ассемблер просматривает текст сверху вниз. Изначально он ничего не знает о макросах и не сможет их обработать, если ему о них не сообщить. Просмотрев файл, содержащий описания макросов, ассемблер запоминает эти описания и продолжает их помнить до окончания трансляции, так что мы можем их использовать в программе — но не раньше, чем о них узнает ассемблер. Именно поэтому мы поставили директиву `%include` в самое начало программы: теперь макросы можно использовать во всём её тексте.

После директивы `%include` мы видим строку со словом `global`; это тоже директива, но к ней мы вернёмся чуть позднее.

Следующая строка программы содержит директиву `section`. Исполняемый файл в ОС Unix устроен так, что в нём машинные команды хранятся в одном месте, а инициализированные (т. е. такие, которым прямо в программе задаётся начальное значение) данные — в другом, и, наконец, в третьем месте содержится информация о том, сколько программе потребуется памяти под неинициализированные данные. При загрузке исполняемого файла в память операционная система создаёт отдельные области памяти (так называемые секции, которые часто называют также сегментами, что не всегда верно) для машинного кода, для данных (здесь объединяются инициализированные и неинициализированные данные) и для стека.

Ассемблер на основе текста нашей программы формирует отдельные образы (то есть будущее содержимое памяти) для каждой из секций; мы должны наш исполняемый код поместить в одну секцию, описания областей памяти с заданным начальным значением — в другую секцию, описания областей памяти без задания начальных значений — в третью секцию. Соответствующие секции называются `.text`, `.data` и `.bss`. Секцию стека операционная система формирует без нашего участия, так что она в программах на языке ассемблера не упоминает-

ся. В нашей простой программе мы обходимся только секцией `.text`; рассматриваемая директива как раз приказывает ассемблеру приступить к формированию этой секции. В будущем при рассмотрении более сложных программ нам придётся встретиться со всеми тремя секциями.

Далее в программе мы видим строку

```
_start: mov     eax, 0
```

Словом `mov` обозначается команда, заставляющая процессор переслать некоторые данные из одного места в другое. После команды указаны два параметра, которые называют ***операндами***; для команды `mov` первый операнд задаёт, *куда* следует скопировать данные, а второй операнд указывает, *какие* данные следует туда скопировать. В данном конкретном случае команда требует занести число `0` (ноль) в регистр `EAX`[13]. Значение, хранимое в регистре `EAX`, мы будем использовать в качестве счётчика цикла, то есть оно будет означать, сколько раз мы уже напечатали слово «Hello»; ясно, что в начале выполнения программы этот счётчик должен быть равен нулю, поскольку мы пока не напечатали ничего.

Итак, рассматриваемая строка означает приказ процессору занести ноль в `EAX`; но что за загадочное «`_start:`» в начале строки?

Слово `_start` (знак подчёркивания в данном случае является частью слова) — это так называемая ***метка***. Попробуем сначала объяснить, что такое собой представляют эти метки «вообще», а потом расскажем, зачем нужна метка в данном конкретном случае.

Команду `mov eax,0` ассемблер преобразует в некий машинный код[14], который во время выполнения программы будет находиться в какой-то области оперативной памяти (в данном случае — в пяти ячейках, идущих подряд). В некоторых случаях нам нужно знать, какой адрес будет иметь та или иная область памяти; если говорить о командах, то адрес нам может потребоваться, например, чтобы в какой-то момент заставить процессор произвести в это место программы условный или безусловный переход.

Конечно, мы можем использовать оперативную память и для хранения данных, а не только команд. Области памяти, предназначенные для

[13] Читатель, уже имеющий опыт программирования на языке ассемблера, может заметить, что «правильнее» это сделать совсем другой командой: `xor eax, eax`, поскольку это позволяет достичь того же эффекта быстрее и с меньшими затратами памяти; однако для простейшего учебного примера такой трюк требует слишком длинных пояснений. Впрочем, позже мы к этому вопросу вернёмся и обязательно рассмотрим этот и другие подобные трюки.

[14] Отметим для наглядности, что машинный код этой команды состоит из пяти байтов: `b8 00 00 00 00`, первый из которых задаёт собственно действие «поместить заданное число в регистр», а также и номер регистра `EAX`. Остальные четыре байта (все вместе) задают то число, которое должно быть помещено в регистр; в данном случае это число 0.

данных, мы обычно называем ***переменными*** и даём им имена почти так же, как и в привычных нам языках программирования высокого уровня, в том числе в Паскале. Естественно, нам требуется знать, какой адрес имеет начало области памяти, отведённой под переменную. Адрес, как мы уже говорили, задаётся[15] числом из восьми шестнадцатеричных цифр, например, `18b4a0f0`. Запоминать такие числа неудобно, к тому же на момент написания программы мы ещё не знаем, в каком именно месте памяти в итоге окажется размещена та или иная команда или переменная. И здесь нам на помощь как раз и приходят метки. **Метка — это вводимое программистом слово (идентификатор), с которым ассемблер ассоциирует некоторое число, чаще всего — адрес в памяти**, но не всегда. В данном случае `_start` как раз и есть такая метка. Если ассемблер видит метку *перед* командой (или, как мы увидим позже, перед директивой, выделяющей память под переменную), он воспринимает это как указание завести в своих внутренних таблицах новую метку и связать с ней соответствующий адрес, если же метка встречается в параметрах команды, то ассемблер вспоминает, какой именно адрес (или просто число) связано с данной меткой и подставляет этот адрес (число) вместо метки в команду. Таким образом, с меткой `_start` в нашей программе будет связано число, представляющее собой адрес ячейки, начиная с которой в оперативной памяти будет размещён машинный код, соответствующий команде `mov eax,0` (код `b8 00 00 00 00`).

Важно понимать, что метки существуют только в памяти самого ассемблера и только во время трансляции программы. Готовая к исполнению программа, представленная в машинном коде, не будет содержать никаких меток, а только подставленные вместо них адреса.

После метки в обсуждаемой строке стоит символ двоеточия. Интересно, что мы могли бы его и не ставить. Некоторые ассемблеры отличают метки, снабженные двоеточиями, от меток без двоеточий; но наш NASM к таким не относится. Иначе говоря, мы сами решаем, ставить двоеточие после метки или нет. Обычно программисты ставят двоеточия после меток, которыми помечены машинные команды (то есть после таких меток, куда можно передать управление), но не ставят двоеточия после меток, помечающих данные в памяти (переменные). Поскольку метка `_start` помечает команду, после неё мы двоеточие решили поставить.

Внимательный читатель может обратить внимание, что никаких переходов на метку `_start` в нашей программе не делается. Зачем же она тогда нужна? Дело в том, что слово «`_start`» — это специальная метка, которой помечается ***точка входа*** в программу, то есть то место в программе, куда операционная система должна передать управление

[15] Во всяком случае, для того процессора и той системы, которые мы рассматриваем.

после загрузки программы в оперативную память; иначе говоря, метка `_start` обозначает место, с которого начнётся выполнение программы.

Вернёмся к тексту программы и рассмотрим следующую строчку:

```
again:  PRINT   "Hello"
```

Как несложно догадаться, слово `again` в начале строки — это ещё одна метка. Слово «again» по-английски означает «снова». Чтобы слово `Hello` оказалось напечатано пять раз, нам придётся ещё четыре раза вернуться в эту точку программы; отсюда и название метки. Стоящее далее в строке слово `PRINT` является ***именем макроса***, а строка `"Hello"` — ***параметром*** этого макроса. Сам макрос описан, как уже говорилось, в файле `stud_io.inc`. Увидев имя макроса и параметр, наш ассемблер подставит вместо них целый ряд команд и директив, исполнение которых приведёт в конечном итоге к выдаче на экран строки «Hello».

Очень важно понимать, что `PRINT` не имеет никакого отношения к возможностям центрального процессора. Мы уже несколько раз упоминали этот факт, но тем не менее повторим ещё раз: `PRINT` — это не имя какой-либо команды процессора, процессор как таковой не умеет ничего печатать. Рассматриваемая нами строчка программы представляет собой не команду, а директиву, также называемую ***макровызовом***. Повинуясь этой директиве, ассемблер сформирует фрагмент текста на языке ассемблера (отметим для наглядности, что в данном случае этот фрагмент будет состоять из 23 строк в случае применения ОС Linux и из 15 строчек — для ОС FreeBSD) и сам же оттранслирует этот фрагмент, получив последовательность машинных инструкций. Эти инструкции будут содержать, в числе прочего, обращение к операционной системе за услугой вывода данных (системный вызов `write`). Набор макросов, включающий в себя и макрос `PRINT`, введён для удобства работы на первых порах, пока мы ещё не знаем, как обращаться к операционной системе. Позже мы узнаем это, и тогда макросы, описанные в файле `stud_io.inc`, станут нам не нужны; более того, мы сами научимся создавать такие макросы.

Вернёмся к тексту нашего примера. Следующая строчка имеет вид

```
        PUTCHAR 10
```

Это тоже вызов макроса, называемого `PUTCHAR` и предназначенного для вывода на печать одного символа. В данном случае мы используем его для вывода символа с кодом `10`; как мы уже знаем, это символ *перевода строки*, то есть при выводе этого символа на печать курсор на экране перейдёт на следующую строку. Обратите внимание, что в этой и последующих строках присутствуют только команды и макровызовы, а меток нет. Они нам не нужны, поскольку ни на одну из последующих

команд мы не собираемся делать переходы, и, значит, нам не нужна информация об адресах в памяти, где будут располагаться эти команды.

Следующая строка в программе такая:

```
inc     eax
```

Здесь мы видим машинную команду `inc`, означающую приказ увеличить заданный регистр на 1. В данном случае увеличивается регистр `EAX`. Напомним, что в регистре `EAX` мы условились хранить информацию о том, сколько раз уже напечатано слово «Hello». Поскольку выполнение двух предыдущих строчек программы, содержащих вызовы макросов `PRINT` и `PUTCHAR`, привело в конечном счёте как раз к печати слова «Hello», следует отразить этот факт в регистре, что мы и делаем. Отметим, что машинный код этой команды оказывается очень коротким — всего один байт (шестнадцатеричное `40`, десятичное 64).

Далее в нашей программе идёт команда сравнения:

```
cmp     eax, 5
```

Машинная команда сравнения двух целых чисел обозначается мнемоникой `cmp` от английского *to compare* — «сравнивать». В данном случае сравниваются содержимое регистра `EAX` и число `5`. Результаты сравнения записываются в специальный регистр процессора, называемый ***регистром флагов***. Это позволяет, например, произвести ***условный переход*** в зависимости от результатов предшествующего сравнения, что мы в следующей строчке программы и делаем:

```
jl      again
```

Здесь `jl` (от слов «Jump if Lower») — это мнемоника для машинной команды условного перехода, который выполняется в случае, если предшествующее сравнение дало результат «первый операнд меньше второго», то есть, в нашем случае, если число в регистре `EAX` оказалось меньше, чем `5`. В терминах нашей задачи это означает, что слово «Hello» было напечатано меньше пяти раз, так что нужно продолжать его печатать, что и делается переходом (***передачей управления***) на команду, помеченную меткой `again`.

Если результат сравнения был любым, кроме «меньше», команда `jl` не произведёт никаких действий, так что процессор перейдёт к выполнению следующей по порядку команды. Это произойдёт в случае, если слово «Hello» уже было напечатано пять раз, то есть как раз тогда, когда цикл будет пора заканчивать. После окончания цикла наша исходная задача оказывается решена, так что программу тоже пора завершать. Для этого и предназначена следующая строка программы:

```
FINISH
```

Слово `FINISH` обозначает, как уже отмечалось, макрос, который разворачивается в последовательность команд, осуществляющих обращение к операционной системе с просьбой завершить выполнение нашей программы.

Нам осталось вернуться к началу программы и рассмотреть строку

```
global  _start
```

Слово `global` — это директива, которая требует от ассемблера считать некоторую метку «глобальной», то есть как бы видимой извне (если говорить строго, видимой извне объектного модуля; это понятие мы будем рассматривать позднее). В данном случае глобальной объявляется метка `_start`. Как мы уже знаем, это специальная метка, которой помечается ***точка входа в программу***, то есть то место в программе, куда операционная система должна передать управление после загрузки программы в оперативную память. Ясно, что эта метка должна быть видна извне, что и достигается директивой `global`.

Итак, наша программа состоит из трёх частей: подготовки, цикла, начало которого отмечено меткой `again`, и завершающей части, состоящей из одной строчки `FINISH`. Перед началом цикла мы заносим в регистр `EAX` число `0`, затем на каждой итерации цикла печатаем слово «Hello», делаем перевод строки, увеличиваем на единицу содержимое регистра `EAX`, сравниваем его с числом `5`; если в регистре `EAX` всё ещё содержится число, меньшее пяти, переходим снова к началу цикла (то есть на метку `again`), в противном случае выходим из цикла и завершаем выполнение программы.

Чтобы попробовать приведённую программу, как говорится, в деле, необходимо войти в систему Unix, вооружиться каким-нибудь редактором текстов, набрать вышеприведённую программу и сохранить её в файле с именем, заканчивающимся на «`.asm`» — именно так обычно называют файлы, содержащие исходный текст на языке ассемблера.

Допустим, мы сохранили текст программы в файле `hello5.asm`. Для получения исполняемого файла нужно выполнить два действия. Первое — запуск ассемблера NASM, который, используя заданный нами исходный текст, построит ***объектный модуль***. Объектный модуль — это ещё не исполняемый файл. Как мы уже знаем из части про Паскаль, большие программы обычно состоят из целого набора исходных файлов, называемых ***модулями***, плюс к тому мы можем захотеть воспользоваться чьими-то сторонними подпрограммами, объединёнными в ***библиотеки***. Каждый модуль компилируется отдельно, давая в результате объектный файл. Для получения исполняемого файла нам нужно будет соединить вместе все объектные файлы, полученные из модулей, и подключить к ним библиотеки; этим занимается ***компоновщик***, также называемый иногда ***редактором связей*** или ***линкером***.

Наша программа состоит всего из одного модуля и не нуждается ни в каких библиотеках, но стадии сборки (компоновки) это не исключает. Это и есть второе действие, нужное для построения исполняемого файла: запустить компоновщик, чтобы он из объектного файла построил файл исполняемый. Как раз на этой стадии будет использована метка `_start`; мы можем уточнить, что директива `global` не просто делает метку «видимой извне», а заставляет ассемблер вставить в объектный файл информацию об этой метке, видимую для компоновщика.

Итак, для начала вызываем ассемблер NASM:

```
avst@host:~/work$ nasm -f elf hello5.asm
```

Флажок «`-f elf`» указывает ассемблеру, что на выходе мы ожидаем объектный файл в формате ELF (*executable and linkable format* — «формат для исполняемых и собираемых файлов») — именно этот формат используется в нашей системе для исполняемых файлов[16]. Результатом запуска ассемблера станет файл `hello5.o`, содержащий объектный модуль. Теперь мы можем запустить компоновщик, который называется `ld`:

```
avst@host:~/work$ ld hello5.o -o hello5
```

Если вы работаете с 64-битной операционной системой, а в наше время это, скорее всего, так и есть, придётся добавить ещё один ключ для компоновщика, чтобы тот произвёл сборку 32-битного исполняемого файла; в частности, для GNU ld под Linux это будет выглядеть так:

```
avst@host:~/work$ ld -m elf_i386 hello5.o -o hello5
```

а при работе с FreeBSD — так:

```
avst@host:~/work$ ld -m elf_i386_fbsd hello5.o -o hello5
```

Узнать, с какой аппаратной платформой вы имеете дело, можно с помощью команды «`uname -a`». Эта команда выдаст одну довольно длинную строку текста, ближе к концу которой вы найдёте обозначение аппаратной архитектуры: `i386`, `i586`, `i686`, `x86` указывают на 32-битные процессоры, тогда как `x86_64`, `amd64` и т. п. — на 64-битные. Может оказаться и так, что ваш компьютер вообще не относится к семейству `i386`, о чём может свидетельствовать какое-нибудь `armv6l` (например, именно таково обозначение архитектуры Raspberry Pi), но в этом случае там совсем другая система команд, а ассемблера NASM, скорее всего, вообще нет. С этим ничего поделать нельзя, придётся найти другой компьютер.

Флажком `-o` (от слова *output* — вывод) мы задали *имя исполняемого файла* (`hello5`, на этот раз без суффикса). Запустим его на исполнение, дав команду «`./hello5`». Если мы нигде не ошиблись, мы увидим пять строчек `Hello`.

[16] Это верно по крайней мере для современных версий операционных систем Linux и FreeBSD. В других системах вам может потребоваться другой формат объектных и исполняемых файлов; сведения об этом обычно есть в технической документации.

3.1.5. Макросы из файла stud_io.inc

Макросы, описанные в файле stud_io.inc, нам неоднократно потребуются в дальнейшем, поэтому, чтобы не возвращаться к ним, ещё раз приведём описание их возможностей. Сам этот файл вы найдёте в архиве примеров, приложенном к нашей книге. Под Linux вы сможете использовать файл в том виде, в котором он находится в архиве.

Для работы под FreeBSD файл придётся слегка изменить. Для этого откройте файл в редакторе текстов, который вы используете для программирования, и найдите в самом его начале (после комментария-аннотации) следующие две строки:

```
%define STUD_IO_LINUX
;%define STUD_IO_FREEBSD
```

Символ точки с запятой означает здесь ***комментарий***, то есть первую из этих двух строк ассемблер видит, а вторую считает комментарием и игнорирует. Чтобы адаптировать файл для FreeBSD, вам нужно, наоборот, первую строчку из работы вывести, а вторую ввести. Для этого точку с запятой в начале второй строчки убираем, а в начале первой строчки — ставим. Получается вот так:

```
;%define STUD_IO_LINUX
%define STUD_IO_FREEBSD
```

После этой правки ваш файл stud_io.inc готов к работе под FreeBSD.

В программе, которую мы разобрали в предыдущем параграфе, мы использовали макросы PRINT, PUTCHAR и FINISH. Кроме этих трёх макросов наш файл stud_io.inc поддерживает ещё макрос GETCHAR, так что всего их четыре.

Макрос PRINT предназначен для печати строки; его аргументом должна быть строка в апострофах или двойных кавычках, ничего другого он печатать не умеет.

Макрос PUTCHAR предназначен для вывода на печать одного символа. В качестве аргумента он принимает код символа, записанный в виде числа или в виде самого символа, взятого в кавычки или апострофы; также можно в качестве аргумента этого макроса использовать однобайтовый регистр — AL, AH, BL, BH, CL, CH, DL или DH. **Использовать другие регистры в качестве аргумента PUTCHAR нельзя!** Наконец, аргументом этого макроса может выступать исполнительный адрес, заключённый в квадратные скобки — тогда код символа будет взят из ячейки памяти по этому адресу.

Макрос GETCHAR считывает символ из потока стандартного ввода (с клавиатуры). После считывания код символа записывается в регистр EAX; поскольку код символа всегда умещается в один байт, его можно извлечь из регистра AL, остальные разряды EAX будут равны нулю. Если символов больше нет (достигнута так называемая ***ситуация конца***

файла, которая в ОС Unix обычно имитируется нажатием Ctrl-D), в `EAX` будет занесено значение `-1` (шестнадцатеричное `FFFFFFFF`, то есть все 32 разряда регистра равны единицам). Никаких параметров этот макрос не принимает.

Макрос `FINISH` завершает выполнение программы. Этот макрос можно вызвать без параметров, а можно вызвать с одним числовым параметром, задающим так называемый ***код завершения процесса***; обычно используют код `0`, если наша программа отработала успешно, и код `1`, если в процессе работы возникли ошибки.

3.1.6. Правила оформления ассемблерных программ

Изучая язык Паскаль, мы много внимания уделяли правилам оформления текста программы. Свои правила существуют и для программ на языке ассемблера, причём они достаточно сильно отличаются; читатель уже мог заметить это по тексту примера, разобранного выше.

Своеобразный стиль, применяемый при работе на языке ассемблера, обусловлен двумя причинами. Во-первых, языки ассемблера относятся к достаточно немногочисленной группе языков программирования, в которых *строка исходного кода представляет собой основную синтаксическую единицу.* В наше время в большинстве языков конец строки рассматривается как один из пробельных символов, ничем принципиальным не отличающийся от пробела и табуляции. Кроме языков ассемблера, построчный синтаксис сейчас сохранился разве что в Фортране. Интересно, что именно Фортран (точнее, его ранние версии) заложил определённую традицию оформления кода, которая сейчас используется для языка ассемблера; что касается самого Фортрана, то для него в последние годы популярен так называемый свободный синтаксис, позволяющий использовать традиционные структурные отступы; встречаются, впрочем, и такие программисты, которые предпочитают писать на Фортране «по-старинке».

Традиция «фиксированного синтаксиса» восходит к тем временам, когда программа на Фортране представляла собой колоду перфокарт. Ранние версии Фортрана требовали, чтобы метка, которая в Фортране представляет собой целое число, записывалась в столбцах с первого по пятый, причём часто метку короче пяти цифр сдвигали вправо, оставляя первые позиции пустыми. В первой позиции можно было также поместить символ `C`, обозначающий комментарий; позже комментарий стало можно обозначать символами `*` или `!`. Текст оператора должен был начинаться строго с седьмой позиции, а перед ним в шестой позиции требовалось поместить пробел; непробельный символ в шестой позиции означал, что эта строка (точнее, перфокарта) является продолжением предыдущей, при этом пустыми должны были остаться первые пять позиций.

Существует и вторая причина, из-за которой ассемблерные программы не похожи на программы на том же Паскале: **программа на языке ассемблера представляет собой прообраз содержимого**

оперативной памяти, которая, согласно принципам фон Неймана, линейна и однородна. Именно по этой причине здесь, как уже, наверное, заметил читатель, не используются никакие структурные отступы и не делается попыток отойти от использования строк в качестве основных синтаксических единиц. Для выделения управляющих конструкций остаются только комментарии, и уж их следует использовать «на всю катушку», если только вы не хотите в итоге получить текст, в котором сами никогда не разберётесь. Подчеркнём ещё раз: **при написании программы на языке ассемблера пишите как можно более подробные комментарии!** В отличие от других языков, где комментариев может оказаться «слишком много», ассемблерную программу «перекомментировать» практически невозможно.

Современный текст на языке ассемблера обычно несколько напоминает программы на ранних версиях Фортрана. Общий принцип оформления ассемблерного кода довольно прост. Следует мысленно разделить горизонтальное пространство экрана на две части — область меток и область команд. Часто выделяют также область комментариев. Код пишется «в столбик» примерно так:

```
        xor ebx, ebx        ; zero ebx
        xor ecx, ecx        ; zero ecx
lp:     mov bl, [esi+ecx]   ; another byte from the string
        cmp bl, 0           ; is the string over?
        je lpquit           ; end the loop if so
        push ebx            ;
        inc ecx             ; next index
        jmp lp              ; repeat the loop
lpquit: jecxz done          ; finish if the string is empty
        mov edi, esi        ; point to the buffer's begin
lp2:    pop ebx             ; get a char
        mov [edi], bl       ; store the char
        inc edi             ; next address
        loop lp2            ; repeat ecx times
done:
```

Если метка не помещается в отведённое для меток пространство, её располагают на отдельной строке:

```
fill_memory:
        jecxz fm_q
fm_lp:  mov [edi], al
        inc edi
        loop fm_lp
fm_q:   ret
```

Обычно при работе на языке ассемблера **под метки выделяют столбец шириной в одну табуляцию и, естественно, именно**

символ табуляции (а не пробелы!) ставят перед каждой командой (в том числе и после меток). Некоторые программисты предпочитают отдать под метки две табуляции; это позволяет использовать более длинные метки без выделения для них отдельных строк:

```
fill_memory:    jecxz fm_q
fm_lp:          mov [edi], al
                inc edi
                loop fm_lp
fm_q:           ret
```

Довольно часто можно встретить стиль оформления, предполагающий отдельную колонку для обозначения команды. Выглядит это примерно так:

```
fill_memory:    jecxz   fm_q
fm_lp:          mov     [edi], al
                inc     edi
                loop    fm_lp
fm_q:           ret
```

К сожалению, этот стиль «ломается» при использовании, например, макросов с именами длиннее семи символов, поскольку имя такого макроса при макровызове следует записать, что вполне естественно, в колонку команды, но пространства в этой колонке не хватает. Впрочем, для макросов можно сделать исключение из правил.

Следует подчеркнуть, что целый ряд общих принципов оформления кода действует для языка ассемблера точно так же, как и для любого другого языка программирования. Попытаемся перечислить их.

Прежде всего напомним, что **текст программы должен состоять исключительно из ASCII-символов**; любые символы, не входящие в набор ASCII, недопустимы в тексте программы даже в комментариях, не говоря уже о строковых константах или тем более идентификаторах. Мы несколько раз упоминали этот момент в тексте первого тома; в частности, в сноске на стр. 236 было сказано, что комментарии следует писать на английском языке, в противном случае их вообще лучше не писать. Для языка ассемблера такая рекомендация не годится, обойтись без комментариев здесь совершенно невозможно; но следует из этого только одно: если у вас имеются какие-то проблемы с английским, их нужно срочно решать, а на первых порах пользоваться словарём.

Как и для любых программ, для текста на языке ассемблера действует *правило восьмидесятой колоноки*: **строки вашей программы не должны превышать 79 символов в длину**. Причины этого подробно обсуждались в первом томе (см. §2.15.4).

Естественно, не стоит забывать о *выборе осмысленных имён для меток*, тем более что в ассемблерных программах большинство вводимых

программистом идентификаторов — глобальные; правила разбивки кода на отдельные подпрограммы, а также на модули и подсистемы тоже совершенно не зависят от используемого языка и применимы к языку ассемблера точно так же, как и везде; здесь мы можем посоветовать снова вернуться к первому тому и перечитать §§ 2.6.4, 2.15.7, 2.17.3 и 2.17.4.

3.2. Основы системы команд i386

В этой главе мы приведём самые простые и самые необходимые сведения об архитектуре процессора i386: рассмотрим его систему регистров, команды для копирования информации, для выполнения целочисленных арифметических действий и для управления выполнением программы. Вопросам, связанным с организацией подпрограмм, а также арифметике чисел с плавающей точкой будут посвящены отдельные главы.

Команды процессора, которые можно задействовать только в привилегированном режиме и которыми, соответственно, пользуется только операционная система, мы не будем рассматривать вообще: рассказ о них занял бы слишком много места, а чтобы их попробовать в деле, пришлось бы написать свою операционную систему. Для поставленных учебных целей это не требуется, а при желании читатель может самостоятельно воспользоваться справочной литературой для более глубокого знакомства с возможностями процессора.

3.2.1. Система регистров

Регистром называют электронное устройство в составе центрального процессора, способное содержать в себе определённое количество данных в виде двоичных разрядов. В большинстве случаев (но не всегда) содержимое регистра трактуется как целое число, записанное в двоичной системе счисления. Регистры процессора i386 можно условно разделить на ***регистры общего назначения***, ***сегментные регистры*** и ***специальные регистры***. Каждый регистр имеет своё название[17], состоящее из двух-трёх латинских букв.

Сегментные регистры (`CS`, `DS`, `SS`, `ES`, `GS` и `FS`) в «плоской» модели памяти не используются. Точнее говоря, перед передачей управления пользовательской задаче операционная система заносит в эти регистры некоторые значения, которые задача теоретически может изменить, но ничего хорошего из этого не выйдет — скорее всего, произойдёт аварийное завершение. Мы принимаем во внимание существование этих регистров, но более к ним возвращаться не будем.

[17] Этим процессоры семейства x86 отличаются от многих других процессоров, в которых регистры имеют номера.

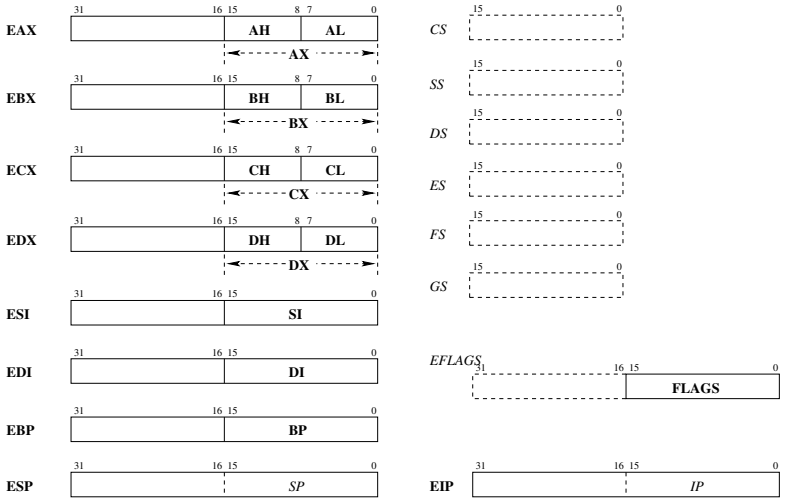

Рис. 3.1. Система регистров i386

Регистры общего назначения процессора i386 — это 32-битные регистры `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` и `ESP`. Как уже отмечалось на стр. 24, буква `E` в названии этих регистров означает слово «extended», которое появилось при переходе от 16-битных регистров старых процессоров к 32-битным регистрам процессора i386. Для совместимости с предыдущими процессорами семейства x86 в каждом 32-битном регистре выделяется обособленная младшая половина (младшие 16 бит), имеющая отдельное название, получаемое отбрасыванием буквы `E`; иначе говоря, мы можем работать также с 16-битными регистрами `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `BP` и `SP`, которые представляют собой младшие половины соответствующих 32-битных регистров.

Кроме того, регистры `AX`, `BX`, `CX` и `DX` также делятся на младшие и старшие части, теперь уже восьмибитные. Так, для регистра `AX` его младший байт имеет также название `AL`, а старший байт — `AH` (от слов «low» и «high»). Аналогично мы можем работать с регистрами `BL`, `BH`, `CL`, `CH`, `DL` и `DH`, которые представляют собой младшие и старшие байты регистров `BX`, `CX` и `DX`. Остальные регистры общего назначения таких обособленных однобайтовых подрегистров не имеют.

Почти все регистры общего назначения в некоторых случаях играют специфическую роль, частично закодированную в имени регистра. Так, в имени регистра `AX` буква `A` обозначает слово «accumulator»; на многих архитектурах, включая знаменитый IAS Джона фон Неймана, ***аккумулятором*** называли регистр, участвующий (по определению) во всех арифметических операциях, во-первых, в качестве одного из операндов, и, во-вторых, в качестве места, куда следует поместить результат. Связанная с этим особая роль регистров `AX` и `EAX` проявляется в командах целочисленного умножения и деления (см. §3.2.8).

В имени регистра BX буква B обозначает слово «base», но никакой особой роли в 32-битных процессорах этому регистру не отведено (хотя в 16-битных процессорах такая роль существовала).

В имени CX буква C обозначает слово «counter» (счётчик). Регистры ECX, CX, а в некоторых случаях даже CL используются во многих машинных командах, предполагающих (в том или ином смысле) определённое количество итераций.

Имя регистра DX символизирует слово «data» (данные). В особой роли регистр EDX (или DX, если выполняется шестнадцатиразрядная операция) выступает при выполнении операций целочисленного умножения (для хранения части результата, не поместившейся в аккумулятор) и целочисленного деления (для хранения старшей части делимого, а после выполнения операции — для хранения остатка от деления).

Имена регистров SI и DI означают, соответственно, «source index» и «destination index» (индекс источника и индекс назначения). Регистры ESI и EDI используются в командах, работающих с массивами данных, причём ESI хранит адрес текущей позиции в массиве-источнике (например, в области памяти, которую нужно куда-то скопировать), а EDI хранит адрес текущей позиции в массиве-цели (в области памяти, *куда* производится копирование).

Имя регистра BP обозначает «base pointer» (указатель базы). Как правило, регистр EBP используется для хранения базового адреса стекового фрейма при вызове подпрограмм, имеющих параметры и локальные переменные.

Наконец, имя регистра SP обозначает «stack pointer» (указатель стека). Несмотря на принадлежность регистра ESP к группе регистров общего назначения, в реальности он всегда используется именно в качестве указателя стека, то есть хранит адрес текущей позиции вершины аппаратного стека. Поскольку обойтись без стека тяжело, а другие регистры для этой цели не подходят, можно считать, что ESP никогда не выступает ни в какой иной роли. К группе регистров общего назначения его относят лишь на том основании, что его можно использовать в арифметических операциях наравне с другими регистрами этой группы.

К регистрам специального назначения мы отнесём ***счётчик команд*** EIP и ***регистр флагов*** FLAGS.

Регистр EIP, имя которого образовано от слов «extended instruction pointer», хранит в себе *адрес того места в оперативной памяти, откуда процессору следует извлечь следующую машинную инструкцию, предназначенную к выполнению.* После извлечения инструкции из памяти значение в регистре EIP автоматически увеличивается на длину прочитанной инструкции (отметим, что инструкция может занимать в памяти от одной до одиннадцати идущих подряд ячеек), так что регистр снова содержит адрес команды, которую нужно выполнить следующей.

Регистр флагов FLAGS — единственный из рассматриваемых нами регистров, который очень редко используется как единое целое и вовсе никогда не рассматривается как число. Вместо этого каждый двоичный разряд (бит) этого регистра представляет собой ***флаг***, имеющий собственное имя. Некоторые из этих флагов процессор сам устанавливает в ноль или единицу в зависимости от результата очередной выполненной команды; другие флаги устанавливаются в явном виде соответствующи-

ми инструкциями и в дальнейшем влияют на ход выполнения некоторых команд. В частности, флаги используются для выполнения ***условных переходов***: некая команда выполняет арифметическую или другую операцию, а следующая команда передаёт управление в другое место программы, но только если результат предыдущей операции удовлетворяет тем или иным условиям; условия как раз и проверяются по установленным флагам. Перечислим некоторые флаги:

- `ZF` — флаг нулевого результата (zero flag). Этот флаг устанавливается в ходе выполнения арифметических операций и операций сравнения: если в результате операции получился ноль, `ZF` устанавливается в единицу.
- `CF` — флаг переноса (carry flag). После выполнения арифметической операции над беззнаковыми числами этот флаг выставляется в единицу, если потребовался перенос из старшего разряда, то есть результат не поместился в регистр, либо потребовался заём из несуществующего разряда при вычитании, то есть вычитаемое оказалось больше, чем уменьшаемое (см. т. 1, §1.6.2). В противном случае флаг выставляется в ноль.
- `SF` — флаг знака (sign flag). Устанавливается равным старшему биту результата, который для знаковых чисел соответствует знаку числа (см. т. 1, стр. 170).
- `OF` — флаг переполнения (overflow flag). Выставляется в единицу, если при работе со знаковыми числами произошло переполнение (см. т. 1, стр. 171).
- `DF` — флаг направления (direction flag). Этот флаг можно установить командой `STD` и обнулить командой `CLD`; в зависимости от его значения *строковые операции*, которые мы будем рассматривать несколько позже, выполняются в прямом или в обратном направлении.
- `PF` и `AF` — флаг чётности (parity flag) и флаг полупереноса (auxiliary carry flag). Нам эти флаги не потребуются.
- `IF` и `TF` — флаги разрешения прерываний (interrupt flag) и ловушки (trap flag). Эти флаги нам *недоступны*, их можно изменять только в привилегированном режиме.

На самом деле такой набор флагов существовал до процессора i386; при переходе к процессору i386 регистр флагов, как и все остальные регистры, увеличился в размерах и поменял название на `EFLAGS`, но все новые флаги в ограниченном режиме недоступны, так что рассматривать их мы не будем.

3.2.2. Память пользовательской задачи. Секции

Ясно, что регистров центрального процессора заведомо не хватит для хранения всей информации, нужной в любой более-менее сложной программе. Поэтому регистры используются лишь для краткосрочного

хранения промежуточных результатов, которые вот-вот понадобятся снова. Кроме регистров, программа может воспользоваться для хранения информации *оперативной памятью*.

Один из основополагающих принципов, определяющих архитектуру фон Неймана, состоит в ***однородности памяти***: и сама программа (то есть составляющие её машинные команды), и все данные, с которыми она работает, располагаются в ячейках памяти, одинаковых по своему устройству и имеющих адреса из единого адресного пространства. В нашем случае каждая ячейка памяти способна хранить ровно один байт и имеет свой уникальный *адрес* — число из 32 бит (речь идёт, естественно, о виртуальных адресах, которые мы обсуждали на стр. 21).

Несмотря на то, что физически все ячейки памяти абсолютно одинаковы, операционная система может установить для пользовательской задачи разные возможности по доступу к различным областям памяти. Это достигается средствами аппаратной защиты памяти, которые мы уже упоминали. В частности, некоторые области памяти могут быть доступны задаче только для чтения, но не для изменения; кроме того, не всякую область памяти разрешается рассматривать как машинный код, то есть заносить адреса ячеек из этой области в регистр счётчика команд. Если задаче позволено рассматривать содержимое области памяти как фрагмент исполняемой машинной программы, говорят, что область памяти *доступна на исполнение*; область памяти, содержимое которой задача может модифицировать, называют *доступной на запись*. Часто можно встретить также термин *доступ на чтение*, но в применении к оперативной памяти отсутствие этого вида доступа обычно означает отсутствие какого-либо доступа вообще.

Обычно современные операционные системы выстраивают виртуальное адресное пространство пользовательской задачи, разделив его на четыре основные ***секции***. Первая из этих секций, называемая ***секцией кода***, создаётся для хранения исполняемого машинного кода, из которого, собственно говоря, и состоит исполняемая программа. Естественно, область памяти, выделенная под секцию кода, доступна задаче на исполнение. С другой стороны, **операционная система не позволяет пользовательским задачам модифицировать содержимое секции кода**; попытка задачи сделать такую модификацию рассматривается как нарушение защиты памяти. Сделано это по достаточно простой причине: если в системе одновременно запущено в виде задач несколько экземпляров одной и той же программы, операционная система обычно хранит в физической памяти только один экземпляр машинного кода такой программы. Это верно даже в случае, если запущенные задачи принадлежат разным пользователям и имеют разные полномочия в системе. Если одна из таких задач модифицирует «свою» секцию кода, очевидно, что это помешает работать остальным — ведь они используют (физически) ту же самую секцию кода. Однако на

чтение секция кода доступна, так что её можно использовать не только для кода как такового, но и для хранения *константных данных* — такой информации, которая не изменяется во время выполнения программы. В программах секция кода обозначается «`.text`»; точка перед названием секции обязательна и является частью названия.

Вторая и третья секции, имеющие собирательное название ***область данных***, предназначены для хранения глобальных и динамических переменных. Обе эти секции доступны задаче как на чтение, так и на запись; с другой стороны, операционная система обычно запрещает передачу управления внутрь этих секций, чтобы несколько затруднить «взлом» компьютерных программ. Первая из двух секций называется собственно ***секцией данных***, в программах обозначается «`.data`[18]» и содержит *инициализированные данные*, то есть такие глобальные переменные, для которых в программе задано начальное значение. Вторая секция из области данных называется ***секцией неинициализированных данных*** или ***секцией BSS***[19] и обозначается «`.bss`»; как ясно из названия, эта секция предназначена для переменных, для которых начальное значение не задано. Секция BSS отличается от секции данных двумя особенностями. Во-первых, поскольку содержимое секции данных на момент старта программы должно быть таким, как это задано программой, её образ необходимо хранить в исполняемом файле программы; для секции BSS в исполняемом файле достаточно хранить только её размер. Во-вторых, секция BSS может во время работы программы увеличиваться в объёме, что позволяет создавать новые переменные на этапе выполнения. В частности, именно там компилятор Паскаля располагает область ***динамической памяти*** («***кучу***»), где во время исполнения программы создаются (с помощью оператора `new`) динамические переменные.

Мы не будем рассматривать работу с динамической памятью на языке ассемблера, но для любознательных читателей сообщим, что в ОС Linux выделение дополнительной памяти производится системным вызовом `brk`, о котором можно узнать из технической документации по ядру. Выделение дополнительной памяти в ОС FreeBSD производится средствами системного вызова `mmap`, который, к сожалению, гораздо сложнее, особенно для использования в программах на языке ассемблера.

Четвёртая основная секция — это так называемая ***секция стека***; она нужна для хранения локальных переменных в подпрограммах и адресов возврата из подпрограмм. Подробный рассказ о стеке у нас ещё впереди, пока мы только отметим, что эта секция также доступна на запись; доступность её на исполнение зависит от конкретной операционной системы и даже от конкретной версии ядра: например, в

[18] Data (`англ.`) — данные; читается «дэйта».

[19] Изначально аббревиатура BSS обозначала Block Started by Symbol, что было обусловлено особенностями одного старого ассемблера. Сейчас программисты предпочитают расшифровывать BSS как Blank Static Storage.

большинстве версий Linux в секцию стека можно передавать управление, но специальный «патч»[20] к исходным текстам ядра эту возможность устраняет. Эта секция также может увеличиваться в размерах по мере необходимости, причём это происходит автоматически (в отличие от увеличения секции BSS, которое необходимо затребовать от операционной системы явно). Секция стека присутствует в пользовательской задаче всегда; её исходное содержимое зависит только от параметров запуска программы. Никакой информации о секции стека исполняемый файл не содержит. Во время написания программы мы не можем никак повлиять на секцию стека, так что ассемблер не имеет никакого специального обозначения для неё.

3.2.3. Директивы для отведения памяти

Содержимое этого параграфа не имеет прямого отношения к архитектуре процессора i386; здесь мы рассмотрим директивы, являющиеся особенностью конкретного ассемблера. Дело, однако, в том, что нам очень сложно будет обойтись без них при изучении дальнейшего материала.

Написанные нами условные обозначения машинных команд ассемблер транслирует в некий *образ области памяти* — массив чисел (данных), которые нужно будет записать в смежные ячейки оперативной памяти. Затем при запуске программы в эту область памяти будет передано управление (то есть адрес какой-то из этих ячеек будет записан в регистр `EIP`) и центральный процессор начнёт ***выполнение*** нашей программы, используя числа из созданного ассемблером образа в качестве кодов команд. Если мы пишем программу, которая будет потом выполняться в качестве задачи под управлением многозадачной операционной системы, то при загрузке исполняемого файла в память операционная система сформирует секцию кода (секцию `.text`) соответствующего размера и именно в ней расположит машинный код нашей программы, то есть скопирует в неё записанный в исполняемом файле образ памяти.

Аналогично можно использовать ассемблер для создания образа области памяти, содержащей данные, а не команды. Для этого нужно сообщить ассемблеру, сколько памяти нам необходимо под те или иные нужды, и при этом, возможно, задать те значения, которые в эту память будут помещены перед стартом программы.

[20] Английским словом *patch* программисты обозначают файл, содержащий формальное описание различий между двумя версиями исходных текстов программы; имея начальную версию исходных текстов и такой файл, можно с помощью специальной программы получить изменённую версию, что позволяет пересылать друг другу не всю программу целиком, а только файл, содержащий нужные изменения. Слово *patch* буквально переводится как «заплатка», но у программистов этот перевод не прижился. Устоявшегося русскоязычного термина, соответствующего английскому *patch*, так и не появилось, в большинстве случаев используется прямая транслитерация с английского — «патч».

Пользуясь нашими указаниями, ассемблер сформирует отдельно образ памяти, содержащей команды (образ секции `.text`), и отдельно образ памяти, содержащей инициализированные данные (образ секции `.data`), а кроме того, сосчитает, сколько нам нужно такой памяти, о начальном значении которой мы не беспокоимся, так что для неё не нужно формировать образ, а нужно лишь указать общий объём (*размер* секции `.bss`). Всё это ассемблер запишет в файл с объектным кодом, а компоновщик из таких файлов (возможно, нескольких) сформирует исполняемый файл, содержащий, кроме собственно машинного кода, во-первых, те данные, которые нужно записать в память перед стартом программы, и, во-вторых, указания на то, сколько программе понадобится ещё памяти, кроме той, что нужна под размещение машинного кода и исходных данных. Чтобы сообщить ассемблеру, в какой секции должен быть размещён тот или иной фрагмент формируемого образа памяти, мы в программе на языке ассемблера должны использовать директиву `section`; например, строчка

```
section .text
```

означает, что результат обработки последующих строк должен размещаться в секции кода, а строчка

```
section .bss
```

заставляет ассемблер перейти к формированию секции неинициализированных данных. Директивы переключения секций могут встречаться в программе сколько угодно раз — мы можем сформировать часть одной секции, затем часть другой, потом вернуться к формированию первой.

Сообщить ассемблеру о наших потребностях в оперативной памяти можно с помощью ***директив резервирования памяти***, которые делятся на два вида: директивы резервирования неинициализированной памяти и директивы задания исходных данных. Обычно перед директивами обоих видов ставится метка, чтобы можно было ссылаться с её помощью на адрес в памяти, где ассемблер отвёл для нас требуемые ячейки.

Директивы резервирования неинициализированной памяти сообщают ассемблеру, что необходимо зарезервировать заданное количество ячеек памяти, причём ничего, кроме количества, не уточняется. Мы не требуем от ассемблера заполнять отведённую память какими-либо конкретными значениями, нам достаточно, чтобы эта память вообще была в наличии. Для резервирования заданного количества однобайтовых ячеек используется директива `resb`, для резервирования памяти под определённое количество «слов[21]», то есть *двухбайтовых*

[21] Напомним, что такая терминология не совсем корректна, поскольку термином «слово» должна обозначаться порция информации, обрабатываемая процессором за

значений (например, коротких целых чисел) — директива `resw`, для «двойных слов» (то есть четырёхбайтных значений) используется `resd`; после директивы указывается (в качестве параметра) число, обозначающее количество значений, под которое мы резервируем память. Как уже говорилось, обычно перед директивой резервирования памяти ставится метка. Например, если мы напишем следующие строки:

```
string  resb 20
count   resw 256
x       resd 1
```

то по адресу, связанному с меткой `string`, будет расположен массив из 20 однобайтовых ячеек (такой массив можно, например, использовать для хранения строки символов); по адресу `count` ассемблер отведёт массив из 256 двубайтных «слов» (т. е. 512 ячеек), которые можно использовать, например, для каких-нибудь счётчиков; наконец, по адресу `x` будет располагаться одно «двойное слово», то есть четыре байта памяти, которые можно использовать для хранения достаточно большого целого числа.

Директивы второго типа, называемые ***директивами задания исходных данных***, не просто резервируют память, а указывают, какие значения в этой памяти должны находиться к моменту запуска программы. Соответствующие значения указываются после директивы через запятую; памяти отводится столько, сколько указано значений. Для задания однобайтовых значений используется директива `db`, для задания «слов» — директива `dw` и для задания «двойных слов» — директива `dd`. Например, строка

```
fibon   dw 1, 1, 2, 3, 5, 8, 13, 21
```

зарезервирует память под восемь двубайтных «слов» (то есть всего 16 байт), причём в первые два «слова» будет занесено число 1, в третье слово — число 2, в четвёртое — число 5 и т. д. С адресом первого байта отведённой и заполненной таким образом памяти будет ассоциирована метка `fibon`.

Числа можно задавать не только в десятичном виде, но и в шестнадцатеричном, восьмеричном и двоичном. Шестнадцатеричное число в ассемблере NASM можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$` (`$2af3`), либо поставив перед числом символы `0x`, как в языке Си (`0x2af3`). При использовании символа `$` необходимо следить, чтобы сразу после `$` стояла цифра, а не буква, так что если число начинается

один приём; начиная с i386, размер машинного слова на этих процессорах составлял четыре байта, а не два. Использование термина `word` в ассемблерах для обозначения двухбайтовых значений — пережиток тех времён, когда машинное слово составляло два байта.

с буквы, необходимо добавить `0` (например, `$0f9` вместо просто `$f9`). Аналогично нужно следить за первым символом и при использовании буквы `h`: например, `a21h` ассемблер воспримет как идентификатор, а не как число. Чтобы избежать проблемы, следует написать `0a21h`. С другой стороны, с числом `2fah` такой проблемы изначально не возникает, поскольку первый символ в его записи является цифрой. Восьмеричное число обозначается добавлением после числа буквы `o` или `q` (например, `634o`, `754q`). Наконец, двоичное число обозначается буквой `b` (`10011011b`).

Отдельного упоминания заслуживают коды символов и *текстовые строки*. Как мы уже знаем, для работы с текстовыми данными каждому символу приписывается ***код символа*** — небольшое целое положительное число. Мы уже знакомы с кодировочной таблицей ASCII (см. т. 1 §1.6.5). Чтобы программисту не нужно было запоминать коды, соответствующие печатным символам (буквам, цифрам и т. п.), ассемблер позволяет вместо кода написать сам символ, взяв его в апострофы или двойные кавычки. Так, директива

```
fig7    db '7'
```

разместит в памяти байт, содержащий число 55 — код символа «семёрка», а адрес этой ячейки свяжет с меткой `fig7`. Мы можем написать и сразу целую строку, например, вот так:

```
welmsg  db 'Welcome to Cyberspace!'
```

В этом случае по адресу `welmsg` будет располагаться строка из 22 символов (то есть массив однобайтовых ячеек, содержащих коды соответствующих символов). Как уже было сказано, кавычки можно использовать как одиночные (апострофы), так и двойные, так что следующая строка полностью аналогична предыдущей:

```
welmsg  db "Welcome to Cyberspace!"
```

Внутри двойных кавычек апострофы рассматриваются как обычный символ; то же самое можно сказать и о символе двойных кавычек внутри апострофов. Например, фразу «`So I say: "Don't panic!"`» можно задать следующим образом:

```
panic   db 'So I say: "Don', "'", 't panic"'
```

Здесь мы сначала воспользовались апострофом в качестве символа одиночных кавычек, так что символ двойных кавычек, обозначающий прямую речь, вошел в нашу строку как обычный символ. Затем, когда нам в строке потребовался апостроф, мы закрыли одиночные кавычки и воспользовались двойными, чтобы набрать символ апострофа.

В конце мы снова воспользовались апострофами, чтобы задать остаток нашей фразы, включая и заканчивающий прямую речь символ двойных кавычек.

Отметим, что строками в одиночных и двойных кавычках можно пользоваться не только с директивой `db`, но и с директивами `dw` и `dd`, однако при этом необходимо учитывать некоторые тонкости, которые мы рассматривать не будем.

При написании программ обычно директивы задания исходных данных располагают в секции `.data` (то есть перед описанием данных ставят директиву `section .data`), а директивы резервирования памяти выделяют в секцию `.bss`. Это обусловлено уже упоминавшимся различием в их природе: инициализированные данные нужно хранить в исполняемом файле, тогда как для неинициализированных достаточно указать их общее количество. Секция `.bss`, как мы помним, как раз и отличается от `.data` тем, что в исполняемом файле для неё хранится только указание размера; иначе говоря, размер исполняемого файла не зависит от размера секции `.bss`. Так, если мы добавим в секцию `.data` директиву

```
db "This is a string"
```

то размер исполняемого файла увеличится на 16 байт (надо же где-то хранить строку `"This is a string"`), тогда как если мы добавим в секцию `.bss` директиву

```
resb 16
```

ассемблер выделит 16 байт памяти, но размер исполняемого файла при этом вообще никак не изменится.

Расположить директивы задания исходных данных мы можем и в секции кода (секции `.text`), нужно только помнить, что тогда эти данные нельзя будет изменить во время работы программы. Но если в нашей программе есть большой массив, который не нужно изменять (какая-нибудь таблица констант, а чаще — некий текст, который наша программа должна напечатать), выгоднее разместить эти данные именно в секции кода, поскольку если пользователи запустят одновременно много экземпляров нашей программы, секция кода у них будет одна на всех и мы сэкономим память. Ясно, что такая экономия возможна только для неизменяемых данных. Помните, что попытка изменить во время выполнения содержимое секции кода приведёт к аварийному завершению программы!

Ассемблер позволяет использовать любые команды и директивы в любых секциях. В частности, мы можем в секцию данных поместить машинные команды, и они будут, как обычно, оттранслированы в соответствующий машиный код, но передать управление на этот код мы не сможем. Всё же в некоторых экзотических случаях такое может иметь смысл, поэтому ассемблер молча выполнит наши указания, сформировав машинный код, который никогда не выполняется. Встретив директивы резервирования памяти (`resb`, `resw` и др.) в секции `.data`,

ассемблер тоже сделает своё дело, но в этом случае будет выдано предупреждающее сообщение; действительно, ситуация несколько странная, поскольку без всякого толка увеличивает размер исполняемого файла, хотя и не приводит ни к каким фатальным последствиям. Ещё более странно будут выглядеть директивы резервирования неинициализированной памяти в секции кода: действительно, если начальное значение не задано, а изменить эту память мы не можем — значит, никакое осмысленное значение в такую память никогда не попадёт, и какой в таком случае от неё толк?! Тем не менее, ассемблер и в этом случае продолжит трансляцию, выдав только предупреждающее сообщение. Предупреждение будет выдано также и в случае, если в секции BSS встретится что-нибудь кроме директив резервирования неинициализированной памяти: ассемблер точно знает, что сформированный для этой секции образ ему будет некуда записывать. Несмотря на то, что во всех перечисленных случаях ассемблер, выдав предупреждение, продолжает работу, правильнее будет предположить, что вы ошиблись, и исправить программу.

3.2.4. Команда `mov` и виды операндов

Одна из самых часто встречающихся в программах на языке ассемблера команд — это команда пересылки данных из одного места в другое. Она называется `mov` (от слова *to move — перемещать*). Для нас эта команда интересна ещё и тем, что на её примере можно изучить целый ряд очень важных вопросов, таких как виды операндов, понятие длины операнда, прямую и косвенную адресацию, общий вид исполнительного адреса, работу с метками и т. д.

Итак, команда `mov` имеет два *операнда*, т. е. два параметра, записываемых после мнемокода команды (в данном случае — слова «`mov`») и задающих объекты, над которыми команда будет работать. Первый операнд задаёт то место, *куда* будут помещены данные, а второй операнд — то, *откуда* данные будут взяты. Так, уже знакомая нам по вводным примерам инструкция

```
mov eax, ebx
```

копирует данные из регистра `EBX` в регистр `EAX`. Важно отметить, что **команда `mov` только копирует данные, не выполняя никаких преобразований**. Для преобразования данных существуют другие команды.

В примерах, рассмотренных выше, мы встречали по меньшей мере два варианта использования команды `mov`:

```
mov eax, ebx
mov ecx, 5
```

Первый вариант копирует содержимое одного регистра в другой регистр, тогда как второй вариант *заносит в регистр некоторое число, заданное*

непосредственно в самой команде (в данном случае число 5). На этом примере наглядно видно, что *операнды бывают разных видов*. Если в роли операнда выступает название регистра, то говорят о ***регистровом операнде***; если же значение указано прямо в самой команде, такой операнд называется ***непосредственным операндом***.

На самом деле в рассматриваемом случае следует говорить даже не о различных типах операндов, а о *двух разных командах,* которые просто обозначаются одинаковой мнемоникой. Две команды `mov` из нашего примера переводятся в совершенно разные машинные коды, причём первая из них занимает в памяти два байта, а вторая — пять, четыре из которых тратятся на размещение непосредственного операнда.

Кроме непосредственных и регистровых операндов, существует ещё и третий вид операнда — ***адресный операнд***, называемый также ***операндом типа «память»***. В этом случае операнд тем или иным способом задаёт *адрес ячейки или области памяти, с которой надлежит произвести заданное командой действие.* Необходимо помнить, что **в языке ассемблера NASM операнд типа «память» абсолютно всегда обозначается квадратными скобками**, в которых и пишется собственно адрес. В простейшем случае адрес задаётся *в явном виде*, то есть в форме числа; обычно при программировании на языке ассемблера вместо чисел мы, как уже говорилось, используем метки. Например, мы можем написать:

```
section .data

; ...
count   dd 0
```

(символ «`;`» в языке ассемблера означает ***комментарий***), описав *область памяти размером в 4 байта, с адресом которой связана метка* `count` *и в которой исходно хранится число* `0`. Если теперь написать

```
section .text
; ...
        mov [count], eax
```

— эта команда `mov` будет обозначать копирование данных из регистра `EAX` в область памяти, помеченную меткой `count`, а, например, команда

```
        mov edx, [count]
```

будет, наоборот, обозначать копирование из памяти по адресу `count` в регистр `EDX`.

Чтобы понять роль квадратных скобок, рассмотрим команду

```
        mov edx, count
```

Вспомним, что метку (в данном случае `count`), как мы уже говорили на стр. 28, ассемблер просто заменяет на некоторое *число*, в данном случае — адрес области памяти. Например, если область памяти `count` расположена в ячейках, адреса которых начинаются с `40f2a008`, то вышеприведённая команда — это абсолютно то же самое, как если бы мы написали

```
mov edx, 40f2a008h
```

Теперь очевидно, что это просто уже знакомая нам форма команды `mov` с непосредственным операндом, т. е. эта команда *заносит в регистр* ***EDX*** *число* ***40f2a008***, не вникая, является ли это число адресом какой-либо ячейки памяти или нет. Если же мы добавим квадратные скобки, речь пойдёт уже об *обращении к памяти* по заданному адресу, то есть число будет использовано как адрес области памяти, где размещено значение, с которым надо работать (в данном случае — поместить в регистр `EDX`).

3.2.5. Косвенная адресация; исполнительный адрес

Задать адрес области памяти в виде числа или метки возможно не всегда. Во многих случаях нам приходится тем или иным способом *вычислять* адрес и уже затем обращаться к области памяти по такому вычисленному адресу. Например, именно так будут обстоять дела, если нам потребуется заполнить все элементы какого-нибудь массива заданными значениями: адрес начала массива нам наверняка известен, но нужно будет организовать цикл (по элементам массива) и на каждом шаге цикла выполнять копирование заданного значения в *очередной* (каждый раз другой) элемент массива. Самый простой способ исполнить это — перед входом в цикл задать некий адрес равным адресу начала массива и на каждой итерации увеличивать его.

Важное отличие от простейшего случая, рассмотренного в предыдущем параграфе, состоит в том, что *адрес, используемый для доступа к памяти, будет вычисляться во время исполнения программы*, а не задаваться при её написании. Таким образом, вместо указания процессору «обратись к области памяти по такому-то адресу» нам нужно потребовать действия более сложного: «возьми там-то (например, в регистре) значение, используй это значение в качестве адреса и по этому адресу обратись к памяти». Такой способ обращения к памяти называют ***косвенной адресацией*** (в отличие от ***прямой адресации***, при которой адрес задаётся явно).

Процессор i386 позволяет для косвенной адресации использовать только значения, хранимые в регистрах процессора. Простейший вид косвенной адресации — это обращение к памяти по адресу, хранящемуся в одном из регистров общего назначения. Например, команда

```
mov ebx, [eax]
```

означает «возьми значение в регистре EAX, используй это значение в качестве адреса, по этому адресу обратись к памяти, возьми оттуда 4 байта и занеси эти 4 байта в регистр EBX», тогда как команда

```
mov ebx, eax
```

означала, как мы уже видели, просто «скопируй содержимое регистра EAX в регистр EBX».

Рассмотрим небольшой пример. Пусть у нас есть массив из однобайтовых элементов, предназначенный для хранения строки символов, и нам необходимо в каждый элемент этого массива занести код символа '@'. Посмотрим, с помощью какого фрагмента кода мы можем это сделать (воспользуемся командами, уже знакомыми нам из примера на стр. 25):

```
section .bss
array   resb 256        ; массив размером 256 байт

section .text
; ...
        mov ecx, 256    ; кол-во элементов -> в счётчик (ECX)
        mov edi, array  ; адрес массива -> в EDI
        mov al, '@'     ; нужный код -> в однобайтовый AL
again:  mov [edi], al   ; заносим код в очередной элемент
        inc edi         ; увеличиваем адрес
        dec ecx         ; уменьшаем счётчик
        jnz again       ; если там не ноль, повторяем цикл
```

Здесь мы использовали регистр ECX для хранения числа итераций цикла, которые ещё осталось выполнить (изначально 256, на каждой итерации уменьшаем на единицу, достигнув нуля — заканчиваем цикл), а для хранения адреса мы воспользовались регистром EDI, в который перед входом в цикл занесли адрес начала массива array и на каждой итерации увеличивали его на единицу, переходя, таким образом, к следующей ячейке.

Внимательный читатель может заметить, что фрагмент кода написан не совсем рационально. Во-первых, можно было бы использовать лишь один изменяемый регистр, либо сравнивая его не с нулём, а с числом 256, либо просматривая массив с конца. Во-вторых, не совсем понятно, зачем для хранения кода символа использовался регистр AL, ведь можно было использовать непосредственный операнд прямо в команде, заносящей значение в очередной элемент массива.

Всё это действительно так, но тогда нам пришлось бы воспользоваться, во-первых, явным указанием размера операнда, а это мы ещё не обсуждали; и, во вторых, пришлось бы использовать команду cmp либо усложнить команду

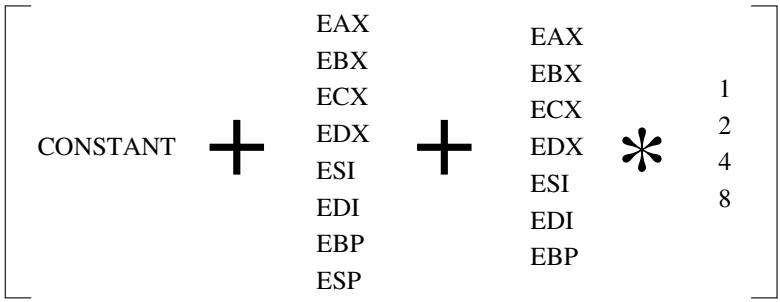

Рис. 3.2. Общий вид исполнительного адреса

присваивания начального значения адреса. Применив здесь не совсем рациональный код, мы смогли ограничиться наименьшим количеством пояснений, отвлекающих внимание от основной задачи.

Итак, адрес для обращения к памяти не всегда задан заранее; мы можем вычислить адрес уже во время выполнения программы, занести результат вычислений в регистр процессора и воспользоваться косвенной адресацией. Адрес, по которому очередная машинная команда произведёт обращение к памяти (неважно, задан этот адрес явно или вычислен) называется ***исполнительным адресом***. Выше мы рассматривали ситуации, когда адрес вычислен, результат вычислений занесён в регистр и именно значение, хранящееся в регистре, используется в качестве исполнительного адреса. Для удобства программирования процессор i386 позволяет задавать исполнительный адрес так, чтобы он вычислялся *уже в ходе выполнения команды*.

Если говорить точнее, мы можем потребовать от процессора взять некоторое заранее заданное значение (возможно, равное нулю), прибавить к нему значение, хранящееся в одном из регистров, а затем взять значение, хранящееся в другом регистре, умножить на 1, 2, 4 или 8 и прибавить результат к уже имеющемуся адресу. Например, мы можем написать

```
mov eax, [array+ebx+2*edi]
```

Выполняя эту команду, процессор сложит число, заданное меткой `array`[22], с содержимым регистра `EBX` и удвоенным содержимым регистра `EDI`, результат сложения использует в качестве исполнительного адреса, извлечёт из области памяти по этому адресу 4 байта и скопирует их в регистр `EAX`. Каждое из трёх слагаемых, используемых в исполнительном адресе, является необязательным, то есть мы можем использовать только два слагаемых или всего одно — как, собственно, мы и поступали до сих пор.

[22] Напомним, что метка есть не что иное, как обозначение некоторого числа, в данном случае, скорее всего, *адреса* начала какого-то массива.

Важно понимать, что выражение в квадратных скобках никоим образом не может быть произвольным. Например, мы не можем взять три регистра, не можем умножить один регистр на 2, а другой на 4, не можем умножать на иные числа, кроме 1, 2, 4 и 8, не можем, например, перемножить два регистра между собой или вычесть значение регистра, вместо того чтобы прибавлять его. Общий вид исполнительного адреса показан на рис. 3.2; как можно заметить, в качестве регистра, подлежащего домножению на коэффициент, мы не можем использовать `ESP`; при этом в качестве регистра, значение которого просто добавляется к заданному адресу, можно использовать любой из восьми регистров общего назначения.

С другой стороны, ассемблер допускает определённые вольности с записью адреса, если только он при этом может корректно преобразовать адрес в машинную команду. Во-первых, слагаемые можно расположить в произвольном порядке. Во-вторых, можно использовать не одну константу, а две: ассемблер сам сложит их и результат запишет в получающуюся машинную команду. Наконец, можно умножить регистр на 3, 5 или 9: если вы напишете, например, `[eax*5]`, ассемблер «переведёт» это как `[eax+eax*4]`. Конечно, если вы попытаетесь написать `[eax+ebx*5]`, ассемблер выдаст ошибку, ведь нужное ему слагаемое вы уже использовали.

Чтобы понять, зачем может понадобиться такой сложный вид исполнительного адреса, достаточно представить себе *двумерный* массив, состоящий, например, из 10 строк, каждая из которых содержит 15 четырёхбайтных целых чисел. Назовём этот массив `matrix`, поставив перед его описанием соответствующую метку:

```
matrix  resd 10*15
```

Для доступа к элементам N-й строки такого массива мы можем вычислить смещение от начала массива до начала этой N-й строки (для этого нужно умножить N на длину строки, составляющую $15 * 4 = 60$ байт), занести результат вычислений, скажем, в `EAX`, затем в другой регистр (например, в `EBX`) занести номер нужного элемента в строке — и исполнительный адрес вида `[matrix+eax+4*ebx]` в точности укажет нам место в памяти, где расположен нужный элемент.

Возможности процессора по вычислению исполнительного адреса можно при желании задействовать отдельно от обращения к памяти. Для этого предусмотрена команда `lea` (название образовано от слов «load effective address»). Команда имеет два операнда, причём первый из них обязан быть регистровым (размером 2 или 4 байта), а второй — операндом типа «память». При этом никакого обращения к памяти команда не делает; вместо этого в регистр, указанный первым операндом, заносится *адрес*, вычисленный обычным способом для второго операнда. Если первый операнд — двухбайтный регистр, то в него будут записаны младшие 16 бит вычисленного адреса. Например, команда

```
lea eax, [1000+ebx+8*ecx]
```

возьмёт значение регистра `ECX`, умножит его на `8`, прибавит к этому значение регистра `EBX` и число `1000`, а полученный результат занесёт в регистр `EAX`. Разумеется, вместо числа можно использовать и метку. Ограничения на выражение в скобках точно такие же, как и в других случаях использования операнда типа «память» (см. рис. 3.2 на стр. 52).

Подчеркнём ещё раз, что **команда `lea` только вычисляет адрес, не обращаясь к памяти**, несмотря на использование операнда типа «память».

3.2.6. Размеры операндов и их допустимые комбинации

Напомним, что мы ввели три типа операндов:

- непосредственные, задающее значение прямо в команде;
- регистровые, предписывающие взять значение из заданного регистра и/или поместить значение в этот регистр;
- операнды типа «память», задающие адрес, по которому в памяти находится нужное значение и/или по которому в память нужно записать результат работы команды.

В разных ситуациях могут действовать определённые ограничения на тип операндов. Например, очевидно, что непосредственный операнд нельзя использовать в качестве первого аргумента команды `mov`, ведь этот аргумент должен задавать место, *куда* производится копирование данных; мы можем копировать данные в регистр или в область оперативной памяти, однако непосредственные операнды ни того, ни другого не задают. Имеются и другие ограничения, налагаемые, как правило, устройством самого процессора как электронной схемы. Так, ни в команде `mov`, ни в других командах нельзя использовать сразу два операнда типа «память». Если необходимо, например, скопировать значение из области памяти `x` в область памяти `y`, делать это придётся через регистр:

```
mov eax, [x]
mov [y], eax
```

Команда `mov [y],[x]` будет отвергнута ассемблером как ошибочная, поскольку ей не соответствует никакой машинный код: процессор попросту *не умеет* выполнять такое копирование за одну инструкцию.

Все остальные комбинации типов операндов для команды `mov` являются допустимыми, то есть за одну команду `mov` мы можем:

- скопировать значение из регистра в регистр;
- скопировать значение из регистра в память;
- скопировать значение из памяти в регистр;

- задать (непосредственным операндом) значение регистра;
- задать (непосредственным операндом) значение ячейки или области памяти.

Последний вариант заслуживает особого рассмотрения. До сих пор во всех командах, которые мы использовали в примерах, хотя бы один из операндов был регистровым; это позволяло не думать о *размере* операндов, то есть о том, являются ли наши операнды отдельными байтами, двухбайтовыми «словами» или четырёхбайтовыми «двойными словами». Отметим, что команда `mov` не может пересылать данные между операндами разного размера (например, между однобайтовым регистром `AL` и двухбайтовым регистром `CX`); поэтому всегда, если хотя бы один из операндов является регистровым, можно однозначно сказать, какого размера порция данных подлежит обработке (в данном случае — простому копированию). Однако же в варианте, когда первый операнд команды `mov` задаёт адрес в памяти, куда нужно записать значение, а второй является непосредственным (то есть записываемое значение задано прямо в команде), ассемблер не знает и не имеет оснований предполагать, какого конкретно размера порцию данных нужно переслать, или, иначе говоря, сколько байт памяти, начиная с заданного адреса, должно быть записано. Поэтому, например, команда

```
mov [x], 25     ; ОШИБКА!!!
```

будет отвергнута как ошибочная: непонятно, имеется в виду *байт* со значением `25`, *«слово»* со значением `25` или *«двойное слово»* со значением `25`. Тем не менее, команда, подобная вышеприведённой, вполне может понадобиться, и процессор умеет такую команду выполнять. Чтобы воспользоваться такой командой, нам нужно просто указать ассемблеру, что конкретно мы имеем в виду. Это делается указанием ***спецификатора размера*** перед любым из операндов; в качестве такого спецификатора может выступать слово `byte`, `word` или `dword`, обозначающие, соответственно, байт, слово или двойное слово (т. е. размер 1, 2 или 4 байта). Так, например, если мы хотели записать число `25` в четырёхбайтную область памяти, находящуюся по адресу `x`, мы можем написать

```
mov [x], dword 25
```

или

```
mov dword [x], 25
```

Сделаем одно важное замечание. Различные машинные команды, выполняющие схожие действия, могут обозначаться одной и той же мнемоникой. Так,

```
mov eax, 2
mov eax, [x]
mov [x], eax
mov [x], al
```

представляют собой четыре совершенно разные машинные команды, они имеют **разные значения машинного кода** и даже занимают разное количество байтов в памяти. Вместе с тем команды

```
mov eax, 2
mov eax, x
```

используют один и тот же машинный код операции и различаются только значением второго операнда, который в обоих случаях непосредственный; действительно, ведь метка `x` будет заменена на адрес, то есть просто число.

3.2.7. Целочисленное сложение и вычитание

Операции сложения и вычитания над целыми числами производятся соответственно командами `add` и `sub`. Обе команды имеют по два операнда, причём первый из них задаёт и одно из чисел, участвующих в операции, и место, куда следует записать результат; второй операнд задаёт второе число для операции (второе слагаемое, либо вычитаемое). Первый операнд должен быть регистровым или типа «память», а второй операнд у обеих команд может быть любого типа, но нельзя в одной команде использовать два операнда типа «память»; таким образом, для этих команд возможны те же пять форм, как и для команды `mov`. Например, команда

```
add eax, ebx
```

означает «взять значение из регистра `EAX`, прибавить к нему значение из регистра `EBX`, а результат записать обратно в регистр `EAX`». Команда

```
sub [x], ecx
```

означает «взять четырёхбайтное число из памяти по адресу `x`, вычесть из него значение из регистра `ECX`, результат записать обратно в память по тому же адресу». Команда

```
add edx, 12
```

увеличит на `12` содержимое регистра `EDX`, а команда

```
add dword [x], 12
```

сделает то же самое с четырёхбайтной областью памяти по адресу `x`; обратите внимание, что нам пришлось явно указать размер операнда (см. §3.2.6, стр. 55).

Интересно, что команды `add` и `sub` не заботятся о том, считаем ли мы их операнды числами знаковыми или беззнаковыми[23]. Сложение и вычитание знаковых и беззнаковых чисел с точки зрения реализации выполняется абсолютно одинаково, так что **при сложении и вычитании процессор может не знать (и не знает), со знаковыми или с беззнаковыми числами он работает**. Помнить о том, какие числа имеются в виду — обязанность программиста.

В соответствии с полученным результатом команды `add` и `sub` выставляют значения *флагов* `OF`, `CF`, `ZF` и `SF` (см. стр. 40). Флаг `ZF` устанавливается в единицу, если в результате последней операции получился ноль, в противном случае флаг сбрасывается; ясно, что значение этого флага осмысленно как для знаковых, так и для беззнаковых чисел, ведь представление нуля для них одинаково.

Флаги `SF` и `OF` имеет смысл рассматривать только при работе со знаковыми числами. `SF` устанавливается в единицу, если получено отрицательное число, иначе он сбрасывается в ноль. Процессор производит установку этого флага, копируя в него старший бит результата; для знаковых чисел старший бит, как мы знаем, соответствует знаку числа. Флаг `OF` устанавливается, если произошло *переполнение*, что означает, что знак полученного результата не соответствует тому, который должен был получиться исходя из математического смысла операций — например, если в результате сложения двух положительных получилось отрицательное или наоборот. Ясно, что этот флаг не имеет никакого смысла для беззнаковых чисел.

Наконец, флаг `CF` устанавливается, если (в терминах беззнаковых чисел) произошел перенос из старшего разряда, либо произошел заём из несуществующего разряда. По смыслу этот флаг является аналогом `OF` в применении к беззнаковым числам (результат не поместился в размер операнда, либо получился отрицательным). Для знаковых чисел этот флаг смысла не имеет.

Не зная, какие числа имеются в виду, процессор по результатам выполнения команд `add` и `sub` устанавливает все четыре флага; программист должен использовать те из них, которые соответствуют смыслу проведённой операции.

Наличие флага переноса позволяет организовать сложение и вычитание чисел, не помещающихся в регистры, способом, напоминающим

[23] Знаковость и беззнаковость целых чисел мы обсуждали в первом томе, §1.6.2; если вы не чувствуете уверенности в обращении с этими терминами, обязательно перечитайте этот параграф и разберитесь в вопросе; при необходимости найдите кого-нибудь, кто сможет вам всё объяснить. В противном случае вы рискуете ничего не понять во всей оставшейся части книги.

школьное сложение и вычитание «в столбик» — так называемое ***сложение и вычитание с переносом***. Для этого в процессоре i386 предусмотрены команды `adc` и `sbb`. По своей работе и свойствам они полностью аналогичны командам `add` и `sub`, но отличаются от них тем, что учитывают значение флага переноса (`CF`) на момент начала выполнения операции. Команда `adc` *добавляет* к своему итоговому результату значение флага переноса, команда `sbb`, напротив, *вычитает* значение флага переноса из своего результата. После того как результат сформирован, обе команды заново выставляют все флаги, включая и `CF`, уже в соответствии с новым результатом.

Приведём пример. Пусть у нас есть два 64-битных целых числа, причём первое записано в регистры `EDX` (старшие 32 бита) и `EAX` (младшие 32 бита), а второе точно так же записано в регистры `EBX` и `ECX`. Тогда сложить эти два числа можно командами

```
add eax, ecx   ; складываем младшие части
adc edx, ebx   ; теперь старшие, с учётом переноса
```

если же нам понадобится произвести вычитание, то это делается командами

```
sub eax, ecx   ; вычитаем младшие части
sbb edx, ebx   ; теперь старшие, с учётом заёма
```

К операциям сложения и вычитания следует отнести ещё несколько команд. Увеличение и уменьшение целого числа на единицу представляет собой настолько часто встречающийся частный случай, что для него предусмотрены специальные ***команды инкремента и декремента*** `inc` и `dec`. Эти команды, с которыми мы уже сталкивались в ранее приведённых примерах, имеют всего один операнд (регистровый или типа «память») и производят соответственно увеличение и уменьшение на единицу. Обе команды устанавливают флаги `ZF`, `OF` и `SF`, но не затрагивают флаг `CF`. Отметим, что при использовании этих команд с операндом типа «память» указание размера операнда оказывается *обязательным*: действительно, для ассемблера нет другого способа понять, какого размера область памяти имеется в виду.

Команда `neg`, также имеющая один операнд, обозначает *смену знака*, то есть операцию «унарный минус». Обычно её применяют к знаковым числам; тем не менее она устанавливает все четыре флага `ZF`, `OF` и `SF` и `CF`, как если бы операнд вычитался из нуля.

Наконец, команда `cmp` (от слова «compare» — «сравнить») производит точно такое же вычитание, как и команда `sub`, за исключением того, что результат никуда не записывается. Команда вызывается ради установки флагов, обычно сразу после неё следует команда условного перехода.

3.2.8. Целочисленное умножение и деление

В отличие от сложения и вычитания, умножение и деление схематически реализуется сравнительно сложно[24], так что команды умножения и деления могут показаться организованными очень неудобно для программиста. Причина этого, по-видимому, в том, что создатели процессора i386 и его предшественников действовали здесь прежде всего из соображений удобства реализации самого процессора.

Надо сказать, что умножение и деление доставляет некоторые сложности не только разработчикам процессоров, но и программистам, и отнюдь не только в силу неудобности соответствующих команд, но и по самой своей природе. Во-первых, в отличие от сложения и вычитания, умножение и деление для знаковых и беззнаковых чисел производится совершенно по-разному, так что небходимы и различные команды.

Во-вторых, интересные вещи происходят с размерами операндов. При умножении размер (количество значащих битов) результата может быть *вдвое* больше, чем размер исходных операндов, так что, если мы не хотим потерять информацию, то одним флажком, как при сложении и вычитании, не обойдёмся: нужен дополнительный регистр для хранения старших битов результата. С делением ситуация ещё интереснее: если модуль делителя превосходит `1`, размер результата будет меньше размера делимого (если точнее, *количество значащих битов* результата двоичного деления не превосходит $n - m + 1$, где n и m — количество значащих битов делимого и делителя соответственно), так что желательно иметь возможность задавать делимое более длинное, чем делитель и результат. Кроме того, целочисленное деление даёт в качестве результата не одно, а два числа: частное и остаток. Разделять между собой операции нахождения частного и остатка нежелательно, поскольку может привести к двухкратному выполнению (на уровне электронных схем) одних и тех же действий.

Все команды целочисленного умножения и деления имеют только один операнд[25], задающий второй множитель в командах умножения и делитель в командах деления, причём этот операнд может быть регистровым или типа «память», но не непосредственным. В роли первого множителя и делимого, а также места для записи результата используются ***неявный операнд***, в качестве которого выступают регистры `AL`, `AX`, `EAX`, а при необходимости — и регистровые пары `DX:AX`

[24]На некоторых процессорах, даже современных, этих операций вообще нет, и причина этого — исключительно сложность их аппаратной реализации. На таких процессорах выполнять умножение приходится «вручную», двоичным столбиком; обычно для такого умножения создают подпрограмму.

[25]На самом деле из этого правила есть исключение: команда целочисленного умножения знаковых чисел `imul` имеет двухместную и даже трёхместную формы, но рассматривать эти формы мы не будем: пользоваться ими ещё сложнее, чем обычной одноместной формой.

Таблица 3.1. Расположение неявного операнда и результатов для операций целочисленного деления и умножения в зависимости от разрядности явного операнда

разрядн. (бит)	умножение		деление		
	неявный множитель	результат умножения	делимое	частное	остаток
8	AL	AX	AX	AL	AH
16	AX	DX:AX	DX:AX	AX	DX
32	EAX	EDX:EAX	EDX:EAX	EAX	EDX

и EDX:EAX (напомним, что буква A означает слово «аккумулятор»; это и есть особая роль регистра EAX, о которой говорилось на стр. 38).

Для умножения беззнаковых чисел применяют команду mul, для умножения знаковых — команду imul. В обоих случаях в зависимости от разрядности операнда (второго множителя) первый множитель берётся из регистра AL (для однобайтной операции), либо AX (для двухбайтной операции), либо EAX (для четырёхбайтной), а результат помещается в регистр AX (если операнды были однобайтными), либо в регистровую пару DX:AX (для двухбайтной операции), либо в регистровую пару EDX:EAX (для четырёхбайтной операции). Это можно более наглядно представить в виде таблицы (см. табл. 3.1).

Команды mul и imul устанавливают флаги CF и OF в ноль, если старшая половина результата фактически не используется, то есть все значащие биты результата уместились в младшей половине. Для mul это означает, что все разряды старшей половины результата содержат нули, для imul — что все разряды старшей половины результата равны старшему биту младшей половины результата, то есть весь результат целиком, будь то регистр AX или регистровые пары DX:AX, EDX:EAX, представляет собой *знаковое расширение* своей младшей половины (соответственно регистры AL, AX или EAX). В противном случае CF и OF устанавливаются в единицу. Значения остальных флагов после выполнения mul и imul *не определены*; это значит, что ничего осмысленного сказать об их значениях нельзя, причём разные процессоры могут устанавливать их по-разному, и даже в результате выполнения той же команды на том же процессоре с теми же значениями операндов флаги могут (по крайней мере, теоретически) получить другие значения.

Для деления (и нахождения остатка от деления) целых чисел применяют команду div (для беззнаковых) и idiv (для знаковых). Единственный операнд команды, как уже говорилось выше, задаёт *делитель*. В зависимости от разрядности этого делителя (1, 2 или 4 байта) делимое берётся из регистра AX, регистровой пары DX:AX или регистровой пары EDX:EAX, частное помещается в регистр AL, AX или EAX, а остаток от деления — в регистры AH, DX или EDX соответственно (см. табл. 3.1). Частное

всегда округляется в сторону нуля (для беззнаковых и положительных — в меньшую, для отрицательных — в бóльшую сторону). Знак остатка, вычисляемого командой `imul`, всегда совпадает со знаком делимого, а абсолютная величина (модуль) остатка всегда строго меньше модуля делителя. Значения флагов после выполнения целочисленного деления не определены.

Отдельного рассмотрения заслуживает ситуация, когда в делителе на момент выполнения команды `div` или `idiv` находится число `0`. Делить на ноль, как известно, нельзя, а собственных средств, чтобы сообщить об ошибке, у процессора нет. Поэтому процессор инициирует так называемое ***исключение***, называемое также ***внутренним прерыванием***, в результате которого управление получает операционная система; в большинстве случаев она сообщает об ошибке и завершает текущую задачу как аварийную. То же самое произойдёт и в случае, если результат деления не уместился в отведённые ему разряды: например, если мы занесём в `EDX` число `10h`, а в `EAX` — любое другое, даже просто `0`, и попытаемся поделить это (то есть шестнадцатеричное 1000000000, или 2^{36}), скажем, на 2 (записав его, например, в `EBX`, чтобы сделать деление 32-разрядным), то результат (2^{35}) в 32 разряда «не влезет», и процессору придётся инициировать исключение. Подробнее об исключениях (внутренних прерываниях) мы расскажем в §3.6.3.

При целочисленном делении знаковых чисел часто приходится перед делением *расширять делимое*: если мы работали с однобайтными числами, из однобайтного делимого, находящегося в `AL`, следует сначала сделать двухбайтное, находящееся в `AX`, для чего в старшую половину `AX` нужно занести `0`, если число неотрицательное, и $\texttt{FF}_{16}$, если отрицательное. Иначе говоря, фактически нужно заполнить старшую половину `AX` знаковым битом от `AL`. Это можно сделать с помощью команды `cbw`. Аналогичным образом команда `cwd` расширяет число в регистре `AX` до регистровой пары `DX:AX`, то есть заполняет разряды регистра `DX`. Команда `cwde` расширяет тот же регистр `AX` до регистра `EAX`, заполняя старшие 16 разрядов этого регистра. Наконец, команда `cdq` расширяет `EAX` до регистровой пары `EDX:EAX`, заполняя разряды регистра `EDX`. В принципе, область применения этих команд не ограничивается целочисленным делением, особенно если говорить о `cwde`. Команды `cbw`, `cwd`, `cwde` и `cdq` не имеют операндов, поскольку всегда работают с одними и теми же регистрами.

Заметим, что при делении беззнаковых чисел нет необходимости прибегать к специальным командам для расширения разрядности числа: достаточно просто обнулить старшую часть делимого, будь то `AH`, `DX` или `EDX`.

3.2.9. Условные и безусловные переходы

Как уже отмечалось, в обычное последовательное выполнение команд можно вмешаться, выполнив ***передачу управления***, называемую также ***переходом***; команда передачи управления принудительно запи-

сывает новый адрес в регистр `EIP`, заставляя процессор продолжить выполнение программы с другого места. Различают команды ***безусловных переходов***, выполняющие передачу управления в другое место программы без всяких проверок, и команды ***условных переходов***, которые могут в зависимости от результата проверки некоторого условия либо выполнить переход в заданную точку, либо не выполнять его — в этом случае выполнение программы, как обычно, продолжится со следующей команды.

Прежде чем обсуждать имеющиеся в системе команд процессора i386 средства для выполнения переходов, нам для начала придётся отметить, что в системе команд процессора i386 все команды передачи управления подразделяются, в зависимости от «дальности» такой передачи, на *три типа*.

- ***Дальние*** (`far`) переходы подразумевают передачу управления во фрагмент программы, расположенный *в другом сегменте*. Поскольку под управлением ОС Unix мы используем «плоскую» модель памяти, в которой разделение на сегменты отсутствует (точнее, имеет место лишь один сегмент, «накрывающий» всё наше виртуальное адресное пространство), такие переходы нам понадобиться не могут: у нас попросту нет других сегментов.
- ***Близкие*** (`near`) переходы — это передача управления в произвольное место внутри одного сегмента; фактически такие переходы представляют собой явное изменение значения `EIP`. В «плоской» модели памяти это именно тот вид переходов, с помощью которого мы можем «прыгнуть» в произвольное место в нашем адресном пространстве.
- ***Короткие*** (`short`) переходы используются для оптимизации в случае, если точка, куда надлежит «прыгнуть», отстоит от текущей команды не более чем на 127 байт вперёд или 128 байт назад. В машинном коде такой команды смещение задаётся всего одним байтом, отсюда соответствующее ограничение.

При написании команды перехода мы можем явно указать вид перехода, поставив после команды слово `short` или `near` (ассемблер понимает, разумеется, и слово `far`, но нам это не нужно). Если этого не сделать, ассемблер выбирает тип перехода *по умолчанию*, причём для безусловных переходов это `near`, что нас обычно устраивает, а вот для условных переходов по умолчанию используется `short`, что создаёт определённые сложности.

Команда безусловного перехода называется `jmp` (от слова *jump*, которое буквально переводится как «прыжок»). У команды предусмотрен один операнд, определяющий собственно адрес, куда следует передать управление. Чаще всего используется форма команды `jmp` с непосредственным операндом, то есть адресом, указанным прямо в команде; естественно, указываем мы не числовой адрес, которого обычно просто

не знаем, а метку. Также возможно использовать регистровый операнд (в этом случае переход производится по адресу, взятому из регистра) или операнд типа «память» (адрес читается из двойного слова, расположенного в заданной позиции в памяти); такие переходы называют ***косвенными***, в отличие от ***прямых***, для которых адрес задаётся явно. Приведём несколько примеров:

```
jmp cycle   ; переход на метку cycle
jmp eax     ; переход по адресу из регистра EAX
jmp [addr]  ; переход по адресу, содержащемуся
            ; в памяти, которая помечена меткой addr
jmp [eax]   ; переход по адресу, прочитанному из
            ; памяти, расположенной по адресу,
            ; взятому из регистра EAX
```

Здесь первая команда задаёт прямой переход, а остальные — косвенный.

Когда в команде `jmp` используется непосредственный операнд, на самом деле ассемблер вычисляет *разницу* адресов между той меткой, куда мы хотим перейти, и адресом команды, непосредственно следующей за `jmp`; именно эта разница и служит *настоящим* непосредственным операндом в получаемой в итоге машинной команде. Говорят, что при выполнении перехода на явным образом заданный адрес используется ***относительная адресация***. При создании программы на языке ассемблера об этом моменте можно не задумываться и даже вообще его не знать, ведь в тексте программы мы в командах просто указываем метки, а дальнейшее — забота ассемблера. Отметим, что всё это верно только для команды прямого перехода, а косвенные переходы выполняются по «настоящему» (абсолютному) адресу.

Если метка, на которую нужно перейти, находится достаточно близко к текущей позиции, можно попытаться оптимизировать машинный код, применив слово `short`:

```
mylabel:
        ; ...
        ; небольшое количество команд
        ; ...
        jmp short mylabel
```

На глаз обычно тяжело определить, действительно ли метка находится достаточно близко, тем более что макросы (например, `GETCHAR`) могут сгенерировать целый ряд команд, иногда слабо предсказуемый по длине. Но об этом можно не беспокоиться: если расстояние до метки окажется больше допустимого, ассемблер выдаст ошибку примерно такого вида:

```
file.asm:35: error: short jump is out of range
```

и останется только найти строку с указанным номером (в данном случае 35) и убрать «несработавшее» слово `short`.

Таблица 3.2. Простейшие команды условного перехода

команда	условие перехода	команда	условие перехода
jz	ZF=1	jnz	ZF=0
js	SF=1	jns	SF=0
jc	CF=1	jnc	CF=0
jo	OF=1	jno	OF=0
jp	PF=1	jnp	PF=0

В противоположность командам безусловного перехода, команды условного перехода ассемблер по умолчанию считает «короткими», если не указать тип перехода явно.

Такой странный подход к командам переходов обусловлен историческими причинами: на ранних процессорах линейки x86 условные переходы были только короткими, других не было. Процессор i386 и все более поздние, конечно же, поддерживают и близкие условные переходы; дальние условные переходы до сих пор не поддерживаются, но нам они всё равно не нужны.

Простейшие команды условного перехода производят переход по указанному адресу в случае, если один из флагов равен нулю (сброшен) или единице (установлен). Имена этих команд образуются из буквы `J` (от слова *jump*), первой буквы названия флага (например, `Z` для флага `ZF`) и, возможно, вставленной между ними буквы `N` (от слова «not»), если переход нужно произвести при условии равенства флага нулю. Все эти команды приведены в табл. 3.2. Напомним, что смысл каждого из флагов мы рассмотрели на стр. 40.

Такие команды условного перехода обычно ставят непосредственно после арифметической операции (например, сразу после команды `cmp`, см. стр. 58). Например, две команды

```
cmp eax, ebx
jz are_equal
```

можно прочитать как приказ «сравнить значения в регистрах `EAX` и `EBX` и если они равны, перейти на метку `are_equal`».

Отметим один важный момент: **все команды условных переходов допускают только непосредственный операнд** (обычно это просто метка). Ни из регистра, ни из памяти взять адрес для такого перехода нельзя. Обычно такое не нужно, но если всё же потребуется, можно сделать переход по противоположному условию на две команды вперёд, а следующей командой поставить безусловный переход; получится, что через этот безусловный переход мы благополучно перепрыгнем, если исходное условие перехода не выполнено, и, наоборот, выполним переход, если условие было выполнено.

Как и для безусловных переходов, при трансляции команд условных переходов в машинный код в качестве операнда вставляется не адрес как таковой, а разница

между позицией в памяти, куда следует «прыгать», и инструкцией, следующей за текущей, то есть используется относительная адресация.

3.2.10. Переходы по результатам сравнений

Если нам нужно сравнить два числа на *равенство*, всё довольно просто: достаточно, как в предыдущем примере, воспользоваться флагом `ZF`. Но что делать, если нас интересует, например, условие $a < b$? Сначала мы, естественно, применим команду

`cmp` a, b

(в качестве a и b могут быть любые операнды, нужно только помнить, что они не могут быть оба одновременно операндами типа «память»). Команда выполнит сравнение своих операндов — точнее, вычтет из a значение b и соответствующим образом выставит значения флагов. Но вот дальнейшее, как мы сейчас увидим, оказывается несколько сложнее.

Если числа a и b — знаковые, то на первый взгляд всё просто: вычитание $a - b$ при условии $a < b$ даёт число строго отрицательное, так что флаг знака (`SF`, sign flag) должен быть установлен, и мы можем воспользоваться командой `js` или `jns`. Но ведь результат мог и не поместиться в длину операнда (например, в 32 бита, если мы сравниваем 32-разрядные числа), то есть могло возникнуть переполнение! В этом случае значение флага `SF` окажется противоположным истинному знаку результата, зато будет взведён флаг `OF` (overflow flag). Таким образом, условие $a < b$ выполняется в двух случаях: если `SF`=1, но `OF`=0 (то есть переполнения не было, число получилось отрицательное), либо если `SF`=0, но `OF`=1 (число получилось положительное, но это результат переполнения, а на самом деле результат отрицательный). Иначе говоря, нас интересует, чтобы флаги `SF` и `OF` *не были равны друг другу*: `SF`≠`OF`. Для такого случая в процессоре i386 предусмотрена команда `jl` (от слов *jump if less than*, «прыгнуть, если менее чем»), обозначаемая также мнемоникой `jnge` (*jump if not greater or equal*, «прыгнуть, если не больше или равно»).

Рассмотрим теперь ситуацию, когда числа a и b — беззнаковые. Как мы уже обсуждали в §3.2.7 (см. стр. 57), по итогам арифметических операций над беззнаковыми числами флаги `OF` и `SF` рассматривать не имеет смысла, но зато осмысленным становится рассмотрение флага `CF` (carry flag), который выставляется в единицу, если по итогам арифметической операции произошел перенос из старшего разряда (при сложении) либо заём из несуществующего разряда (для вычитания). Именно это нам здесь и нужно: если a и b рассматриваются как беззнаковые и $a < b$, то при вычитании $a - b$ как раз и произойдёт такой заём. Таким образом, нам достаточно воспользоваться значением флага `CF`, то есть выполнить

Таблица 3.3. Команды условного перехода по результатам арифметического сравнения (`cmp` a, b)

имя ком.	jump if...	выр. $a \vee b$	условие перехода	синоним
равенство				
`je` `jne`	equal not equal	$a = b$ $a \neq b$	`ZF`$= 1$ `ZF`$= 0$	`jz` `jnz`
неравенства для знаковых чисел				
`jl` `jnge`	less not greater or equal	$a < b$	`SF`$\neq$`OF`	
`jle` `jng`	less or equal not greater	$a \leqslant b$	`SF`$\neq$`OF` или `ZF`$= 1$	
`jg` `jnle`	greater not less or equal	$a > b$	`SF`$=$`OF` и `ZF`$= 0$	
`jge` `jnl`	greater or equal not less	$a \geqslant b$	`SF`$=$`OF`	
неравенства для беззнаковых чисел				
`jb` `jnae`	below not above or equal	$a < b$	`CF`$= 1$	`jc`
`jbe` `jna`	below or equal not above	$a \leqslant b$	`CF`$= 1$ или `ZF`$= 1$	
`ja` `jnbe`	above not below or equal	$a > b$	`CF`$= 0$ и `ZF`$= 0$	
`jae` `jnb`	above or equal not below	$a \geqslant b$	`CF`$= 0$	`jnc`

команду `jc`, которая специально для данной ситуации имеет синонимы `jb` (*jump if below*, «прыгнуть, если ниже») и `jnae` (*jump if not above or equal*, «прыгнуть, если не выше или равно»).

Когда нас интересуют соотношения «больше» и «меньше либо равно», необходимо включить в рассмотрение и флаг `ZF`, который (как для знаковых, так и для беззнаковых чисел) обозначает равенство аргументов предшествующей команды `cmp`.

Все команды условных переходов по результату арифметического сравнения приведены в табл. 3.3.

3.2.11. О построении ветвлений и циклов

Новички часто теряются, пытаясь с помощью команд условных и безусловных переходов построить привычные по тому же Паскалю конструкции ветвления (оператор `if`–`else`) или цикла с предусловием (оператор `while`). Секрет их построения состоит в том, что условный

переход в большинстве случаев приходится делать по **противоположному** условию; например, если в Паскале мы бы написали цикл с заголовком вроде «`while a = 0 do`», то на языке ассемблера нам придётся сначала сравнить `a` с нулём (например, командой `cmp dword [a], 0`), а затем сделать переход командой `jnz`, то есть *jump if* ***not*** *zero*.

Вообще обычный паскалевский цикл с предусловием

`while` *условие* `do` *тело*

средствами машинных команд реализуется по следующей схеме:

```
cycle:  вычисление условия
        JNx cycle_quit          ; выход
        выполнение тела
        JMP cycle               ; повтор
cycle_quit:
```

а ветвление в его полном варианте

`if` *условие* `then` *ветвь1* `else` *ветвь2*

превращается в

```
        вычисление условия
        JNx else_branch         ; на ветку else
        выполнение ветви1
        JMP if_quit             ; обход ветки else
else_branch:
        выполнение ветви2
if_quit:
```

В обоих случаях «мнемоникой» `JNx` мы обозначили условный переход по *невыполнению* условия, то есть переход, который выполняется, если условие оказалось ложно. Это вполне понятно, если учесть, что сразу за условием в первом случае идёт тело цикла, во втором — ветка *then*, то есть именно те действия, которые нужно сделать, если условие выполнено (истинно); но в таком случае нам не нужно никуда «прыгать», мы уже находимся, где надо. «Прыжок» нужно выполнить, если тело, предназначенное к исполнению, необходимо пропустить не исполняя — то есть если условие оказалось ложным.

При программировании на языке ассемблера можно заметить, что условные переходы по *невыполнению* условия встречаются гораздо чаще, чем переходы по его *выполнению*. Стоит сказать, что в таких случаях при наличии выбора лучше применять мнемоники с буквой `n`, такие как `jnb`, `jna`, `jnge`, `jnle` и т. п., чтобы подчеркнуть: переход совершается по *противоположному* условию; это добавит ясности вашей программе. Напомним, что буква `n` во всех этих мнемониках означает *not*.

3.2.12. Условные переходы и регистр ECX; циклы

Как уже говорилось, некоторые регистры общего назначения в некоторых случаях играют особую роль. В частности, регистр ECX лучше других приспособлен к роли *счётчика цикла*: в системе команд процессора i386 имеются специальные команды для построения циклов с ECX в роли счётчика, а для других регистров таких команд нет.

Одна из таких команд называется loop и предназначена для организации циклов с заранее известным количеством итераций. В качестве счётчика цикла она использует регистр ECX, в который перед началом цикла следует занести число нужных итераций. Сама команда loop выполняет два действия: уменьшает на единицу значение в регистре ECX и, если в результате значение не стало равным нулю, производит переход на заданную метку. Отметим, что команда loop имеет одно важное ограничение: она выполняет только «короткие» переходы, то есть с её помощью невозможно осуществить переход на метку, отстоящую от самой команды более чем на 128 байт.

Пусть, например, у нас есть массив из 1000 двойных слов, заданный с помощью директивы

```
array   resd 1000
```

и мы хотим посчитать сумму его элементов. Это можно сделать с помощью следующего фрагмента кода:

```
        mov ecx, 1000   ; кол-во итераций
        mov esi, array  ; адрес первого элемента
        mov eax, 0      ; начальное значение суммы
lp:     add eax, [esi]  ; прибавляем число к сумме
        add esi, 4      ; адрес следующего элемента
        loop lp         ; уменьшаем счётчик;
                        ; если нужно - продолжаем
```

Здесь мы использовали фактически две переменные цикла — регистр ECX в качестве счётчика и регистр ESI для хранения адреса текущего элемента массива.

Конечно, можно произвести аналогичное действие и для любого другого регистра общего назначения, воспользовавшись двумя командами. Например, мы можем уменьшить на единицу регистр EAX и осуществить переход на метку lp при условии, что полученный в EAX результат не равен нулю; это будет выглядеть так:

```
        dec eax
        jnz lp
```

Точно так же можно записать две команды и для регистра ECX:

```
        dec ecx
        jnz lp
```

Преимущество команды `loop lp` перед этими двумя командами состоит в том, что её машинный код занимает меньше памяти, хотя, как ни странно, на большинстве процессоров работает медленнее.

В примере с массивом можно обойтись и без ESI, одним только счётчиком:

```
        mov ecx, 1000
        mov eax, 0
lp:     add eax, [array+4*ecx-4]
        loop lp
```

Здесь есть два интересных момента. Во-первых, массив мы вынуждены проходить от конца к началу. Во-вторых, исполнительный адрес в команде add имеет несколько странный вид. Действительно, регистр ECX пробегает значения от 1000 до 1 (для нулевого значения цикл уже не выполняется), тогда как адреса элементов массива пробегают значения от array+4*999 до array+4*0, так что умножать на 4 следовало бы не ECX, а (ecx-1). Однако этого мы сделать не можем и просто вычитаем 4. На первый взгляд это противоречит сказанному в §3.2.5 относительно общего вида исполнительного адреса (слагаемое в виде константы должно быть одно либо ни одного), но на самом деле ассемблер NASM прямо во время трансляции вычтет значение 4 из значения array и уже в таком виде оттранслирует, так что в итоговом машинном коде константное слагаемое будет одно.

Рассмотрим теперь две дополнительные команды условного перехода. Команда `jcxz` (jump if CX is zero) производит условный переход, *если в регистре* ***CX*** *содержится ноль*. Флаги при этом не учитываются. Аналогичным образом команда `jecxz` производит переход, если ноль содержится в регистре ECX. Как и для команды `loop`, этот переход всегда короткий. Чтобы понять, зачем введены эти команды, представьте, что на момент входа в цикл в регистре ECX *уже* содержится ноль. Тогда сначала выполнится тело цикла, а потом команда `loop` уменьшит счётчик на единицу, в результате чего счётчик окажется равен максимально возможному целому беззнаковому числу (двоичная запись этого числа состоит из всех единиц), так что тело цикла будет выполнено 2^{32} раз, тогда как по смыслу его, скорее всего, не следовало выполнять вообще. Чтобы избежать таких неприятностей, перед циклом можно поставить команду `jecxz`:

```
        ; заполняем ecx
        jecxz lpq
lp:     ; тело цикла
        ; ...
        loop lp
lpq:
```

В заключение упомянем две модификации команды `loop`. Команда `loope`, называемая также `loopz`, производит переход, если в регистре `ECX` *после его уменьшения на единицу* — не ноль и при этом флаг `ZF` установлен, тогда как команда `loopne` (или, что то же самое, `loopnz`) — если в регистре `ECX` не ноль и флаг `ZF` сброшен. Уменьшение `ECX` регистра эти команды производят в любом случае. Как можно догадаться, буква «`e`» здесь означает *equal*, а буква «`z`» — *zero*.

3.2.13. Побитовые операции

Информацию, записанную в регистры и память в виде байтов, слов и двойных слов можно рассматривать не только как представление целых чисел, но и как строки, состоящие из отдельных и (в общем случае) никак не связанных между собой битов.

Для работы с такими битовыми строками используются специальные команды ***побитовых операций***. Простейшими из них являются двухместные команды `and`, `or` и `xor`, выполняющие соответствующую логическую операцию («и», «или», «исключающее или») отдельно над первыми битами обоих операндов, отдельно над вторыми битами и т. д.; результат, представляющий собой битовую строку той же длины, что и операнды, заносится, как обычно для арифметических команд, в регистр или область памяти, определяемую первым операндом. Ограничения на используемые операнды у этих команд такие же, как и у двухместных арифметических команд: первый операнд должен быть либо регистровым, либо типа «память», второй операнд может быть любого типа; нельзя использовать операнд типа «память» одновременно для первого и второго операнда; если ни один из операндов не является регистровым, необходимо указать разрядность операции с помощью одного из слов `byte`, `word` и `dword`. Осуществить побитовое отрицание (инверсию) можно с помощью команды `not`, имеющей один операнд. Операнд может быть регистровый или типа «память»; в последнем случае, естественно, необходимо задать длину операнда словом `byte`, `word` или `dword`. Все эти команды устанавливают флаги `ZF`, `SF` и `PF` в соответствии с результатом; обычно используется только флаг `ZF`.

В программах на языке ассемблера очень часто встречается команда `xor`, оба операнда которой представляют собой один и тот же регистр, например,

```
xor eax, eax
```

Это означает *обнуление* указанного регистра, т. е. то же самое, что и

```
mov eax, 0
```

Команду `xor` для этого используют, потому что она занимает меньше места (2 байта против 5 для команды `mov`) и работает на несколько тактов быстрее. Некоторые программисты вместо `mov eax,-1` предпочитают использовать две

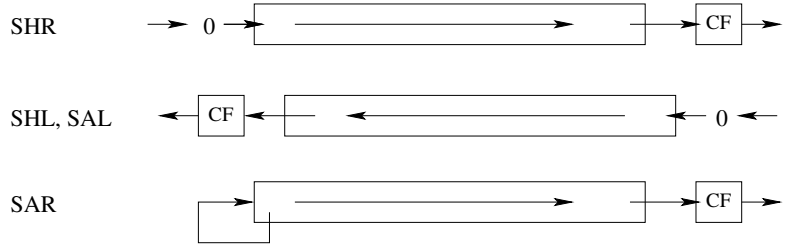

Рис. 3.3. Схема работы команд побитового сдвига

команды `xor eax,eax` и `not eax`, хотя выигрыш тут уже не столь заметен (4 байта кода против 5), а по времени исполнения можно и проиграть.

Можно привести и другие примеры подобного использования побитовых операций. Так, с помощью команды `and` можно получить остаток от деления беззнакового числа на степень двойки (от 1 до 32 степени); например, «`and eax,3`» оставит в `eax` остаток от деления его исходного значения на 4, а «`and eax,1fh`» — от остаток от деления на 32.

В случае, если необходимо просто проверить наличие в числе одного из заданных битов, может оказаться удобной команда `test`, которая работает так же, как и команда `and`, то есть выполняет побитовое «и» над своими операндами, но результат никуда не записывает, а только выставляет флаги.

В частности, для проверки на равенство нулю вместо

```
cmp eax, 0
```

часто используют команду

```
test eax, eax
```

которая занимает меньше памяти и работает быстрее.

Кроме команд, работающих над каждым битом операнда (операндов) и реализующих логические операции, часто приходится применять ***операции побитового сдвига***, которые работают над всеми битами операнда сразу, попросту *сдвигая* их. Команды *простого побитового сдвига* `shr` (*shift right*) и `shl` (*shift left*) имеют два операнда, первый из который указывает, *что* сдвигать, а второй — на сколько битов производить сдвиг. Первый операнд может быть регистровым или типа «память» (во втором случае обязательно указание разрядности). Второй операнд может быть либо непосредственным, то есть числом от 1 до 31 (на самом деле можно указать любое число, но от него будут использоваться только младшие пять разрядов), либо *регистром* `CL`; никакие другие регистры использовать нельзя. При выполнении этих команд с регистром `CL` в качестве второго операнда процессор игнорирует все разряды `CL`, кроме пяти младших.

Схема сдвига на 1 бит следующая. При сдвиге влево старший бит сдвигаемого числа переносится во флаг `CF`, остальные биты сдвигаются влево (то есть бит с номером[26] n получает значение, которое до операции имел бит с номером $n-1$), в младший бит записывается ноль. При сдвиге вправо, наоборот, во флаг `CF` заносится младший бит, все биты сдвигаются вправо (то есть бит с номером n получает значение, которое до операции имел бит с номером $n+1$), в старший бит записывается ноль.

Отметим, что для *беззнаковых* чисел сдвиг на n бит влево эквивалентен умножению на 2^n, а сдвиг вправо — целочисленному делению на 2^n с отбрасыванием остатка. Интересно, что для *знаковых* чисел ситуация со сдвигом влево абсолютно аналогична, а вот сдвиг вправо для любого отрицательного числа даст положительное, ведь в знаковый бит будет записан ноль. Поэтому наряду с командами простого сдвига вводятся также и команды ***арифметического побитового сдвига*** `sal` (*shift arithmetic left*) и `sar` (*shift arithmetic right*). Команда `sal` делает то же самое, что и команда `shl` (на самом деле это одна и та же машинная команда). Что касается команды `sar`, то она работает аналогично команде `shr`, за исключением того, что в старшем бите значение сохраняется таким же, каким оно было до операции; таким образом, если рассматривать сдвигаемую битовую строку как запись знакового целого числа, то операция `sar` не изменит знак числа (положительное останется положительным, отрицательное — отрицательным). Иначе говоря, операция арифметического сдвига вправо эквивалентна делению на 2^n с отбрасыванием остатка *для знаковых целых чисел*. Операции простых и арифметических сдвигов схематически показаны на рис. 3.3.

Команды побитовых сдвигов работают гораздо быстрее, чем команды умножения и деления; кроме того, обращаться с ними существенно легче: можно использовать любые регистры, так что не нужно думать о высвобождении аккумулятора. Поэтому при умножении и делении на степени двойки программисты почти всегда используют именно команды побитовых сдвигов. Более того, как правило, компиляторы языков высокого уровня при трансляции арифметических выражений тоже по возможности стараются использовать сдвиги вместо умножения и деления.

Кроме рассмотренных, процессор i386 поддерживает также команды «сложных» побитовых сдвигов `shrd` и `shld`, работающих через два регистра; команды ***циклического побитового сдвига*** `ror` и `rol`; команды циклического сдвига через флаг `CF` — `rcr` и `rcl`. Могут оказаться очень полезны команды, работающие с отдельными битами своих операндов — `bt`, `bts`, `btc`, `btr`, `bsf` и `bsr`. Все эти команды мы рассматривать

[26] По традиции мы предполагаем, что биты занумерованы справа налево, начиная с нуля, то есть, например, в 32-битном числе младший бит имеет номер 0, а старший — номер 31.

не будем; при желании читатель может освоить их самостоятельно, используя справочники.

Рассмотрим пример ситуации, в которой целесообразно применение битовых операций. Битовые строки удобно использовать для представления подмножеств из конечного числа исходных элементов; попросту говоря, у нас имеется конечное множество объектов (например, сотрудники какого-нибудь предприятия, или тумблеры на каком-нибудь пульте управления, или просто числа от `0` до `N`) и нам в программе нужна возможность представлять *подмножество* этого множества: какие из сотрудников сейчас находятся на работе; какие из тумблеров на пульте установлены в положение «включено»; какие из `N` спортсменов, участвующих в марафоне, прошли очередной контрольный пункт; и т. п. Наиболее очевидное представление для подмножества множества `N` элементов — это область памяти, содержащая `N` двоичных разрядов (так, если в множество могут входить числа от 0 до 511, нам потребуется 512 разрядов, то есть 64 однобайтовых ячейки), где каждому из `N` возможных элементов приписывается один разряд, и этот разряд будет равен единице, если соответствующий элемент входит в подмножество, и нулю в противном случае. Говорят, что каждому из `N` объектов присвоен один из двух *статусов*: либо «входит в множество» (`1`), либо «не входит в множество» (`0`).

Итак, пусть нам потребовалось подмножество множества из 512 элементов; это могут быть совершенно произвольные объекты, нас интересует только то, что у каждого из них есть уникальный номер — число от 0 до 511. Чтобы хранить такое множество, мы опишем массив из 16 двойных слов (напомним, что двойное слово содержит 32 бита и может, соответственно, хранить статус 32 разных объектов). Как обычно, элементы массива будем считать занумерованными (или имеющими *индексы*) от 0 до 15. Элемент массива с индексом 0 будет хранить статус объектов с номерами от 0 до 31, элемент с индексом 1 — статус объектов с номерами от 32 до 63 и т. д. При этом внутри самого элемента будем считать биты занумерованными справа налево, то есть самый младший разряд будет иметь номер 0, самый старший — номер 31. Например, статус объекта с номером 17 будет храниться в 17-м бите нулевого элемента массива; статус объекта с номером 37 — в 5-м бите первого элемента; статус объекта с номером 510 — в 29-м бите 15-го элемента массива. Вообще, чтобы по номеру объекта `X` узнать, в каком бите какого элемента массива хранится его статус, достаточно разделить `X` на 32 (количество бит в каждом элементе) с остатком. Частное будет соответствовать номеру элемента в массиве, остаток — номеру бита в этом элементе. Это можно было бы сделать с помощью команды `div`, но лучше вспомнить, что число 32 есть степень двойки (2^5), так что если взять младшие пять бит числа `X`, мы получим остаток от его деления на 32, а если выполнить для него побитовый сдвиг вправо на 5 позиций —

результат будет равен искомому частному. Например, пусть число X занесено в регистр EBX, и нам нужно узнать номер элемента и номер бита в элементе. Оба номера не превосходят 255 (точнее, номер элемента не превосходит 15, а номер бита не превосходит 32), так что результат мы можем разместить в однобайтовых регистрах; пусть это будут BL (для номера бита) и BH (для номера элемента массива). Поскольку занесение любых новых значений в BL и BH испортит содержимое регистра EBX как целого, логично будет сначала скопировать число куда-то ещё, например в EDX, потом в EBX обнулить все биты, кроме пяти младших. При этом и значение EBX как целого, и значение его младшего байта — регистра BL станут равны искомому остатку от деления; потом в EDX мы выполним сдвиг вправо и реультат, который полностью уместится в младшем байте регистра EDX, то есть в регистре DL, скопируем в BH:

```
mov     edx, ebx
and     ebx, 11111b  ; взяли 5 младших разрядов
shr     edx, 5       ; разделили остальное на 32
mov     bh, dl
```

Однако то же самое можно сделать и короче, без использования дополнительных регистров, ведь все нужные биты у нас с самого начала находятся в EBX. Младшие пять разрядов числа X — это нужный нам остаток от деления, а нужное нам частное — это *следующие* несколько (в данном случае — не более четырёх) разрядов. Когда в EBX занесли число X, эти разряды оказались в позициях начиная с пятой, а нам нужно, чтобы они оказались в регистре BH, который есть не что иное, как второй байт регистра EBX, так что достаточно сдвинуть всё содержимое EBX *влево* на три позиции, и нужный нам результат деления аккуратно «впишется» в BH; после этого содержимое BL мы сдвинем обратно на те же три бита, что заодно и очистит нам его старшие биты:

```
shl     ebx, 3
shr     bl, 3
```

Научившись преобразовывать номер объекта в номер элемента массива и номер разряда в элементе, вернёмся к исходной задаче. Для начала опишем массив:

```
section .bss
set512  resd    16
```

Теперь у нас есть подходящая область памяти, и с адресом её начала связана метка set512. Где-то в начале программы (а возможно, и не только в начале) нам, видимо, понадобится операция очистки множества, то есть такой набор команд, после которого статус всех элементов оказывается нулевой (в множество не входит ни один элемент). Для этого достаточно занести нули во все элементы массива, например, так:

```
section .text

; ...

        xor     eax, eax          ; eax := 0
        mov     ecx, 15
        mov     esi, set512
lp:     mov     [esi+4*ecx], eax
        loop    lp
```

Пусть теперь у нас в регистре EBX имеется номер элемента X, и нам необходимо внести элемент в множество, то есть установить соответствующий бит в единицу. Для этого мы сначала найдём номер бита в элементе массива и вычислим *маску* — такое число, в котором только один бит (как раз нужный нам) равен единице, а в остальных разрядах нули. Затем мы найдём нужный элемент массива и применим к нему и к маске операцию «или», результат которой занесём обратно в элемент массива. При этом нужный нам бит в элементе окажется равен единице, а остальные не изменятся. Для вычисления маски мы возьмём единицу и сдвинем её на нужное количество разрядов влево. Напомним, что из регистров только CL может быть вторым аргументом команд побитовых сдвигов, так что номер бита имеет смысл сразу вычислять в CL. Итак, пишем:

```
; внести в множество set512 элемент,
; номер которого находится в EBX
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        or      [set512+4*edx], eax ; применяем маску
```

Аналогично решается и задача по исключению элемента из множества, только маска на этот раз будет инвертирована (0 в нужном разряде, единицы во всех остальных), а применять мы её будем с командой and (логическое «и»), в результате чего нужный бит обнулится, остальные не изменятся:

```
; убрать из множества set512 элемент,
; номер которого находится в EBX
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        not     eax                 ; инвертируем маску
```

```
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        and     [set512+4*edx], eax ; применяем маску
```

Узнать, входит ли элемент с заданным номером в множество, можно тоже с помощью маски (единица в нужном разряде, нули в остальных) и команды `test`. Результат покажет флаг `ZF`: если он будет взведён — значит, соответствующего элемента в множестве не было, и наоборот:

```
; узнать, входит ли в множество set512 элемент,
; номер которого находится в EBX
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        test    [set512+4*edx], eax ; применяем маску
; теперь ZF=1 означает, что элемент в множестве
;   отсутствовал, а ZF=0 - что присутствовал
```

Рассмотрим ещё один пример. Пусть нам потребовалось сосчитать, сколько элементов входит в множество. Для этого придётся просмотреть все элементы массива и в каждом из них сосчитать единичные биты. Проще всего это сделать, загрузив значение из элемента массива в регистр, а потом сдвигая значение вправо на один бит и каждый раз проверяя, единица ли в младшем разряде; это можно делать ровно 32 раза, но проще закончить, когда в регистре останется ноль. Массив мы будем просматривать с конца, индексируя по `ECX`: это позволит нам применить команду `jecxz`. В качестве счётчика результата воспользуемся регистром `EBX`, а для анализа элементов массива применим `EAX`.

```
; сосчитать элементы в множестве set512
        xor     ebx, ebx    ; EBX := 0
        mov     ecx, 15     ; последний индекс
lp:     mov     eax, [set512+4*ecx] ; загрузили элемент
lp2:    test    eax, 1      ; единица в младшем разряде?
        jz      notone      ;     если нет, прыгаем
        inc     ebx         ; если да, увеличиваем счётчик
notone: shr     eax, 1      ; сдвинули EAX
        test    eax, eax    ; там ещё что-то осталось?
        jnz     lp2         ; если да, продолжаем
                            ;     внутренний цикл
        jecxz   quit        ; если в ECX ноль, заканчиваем
        dec     ecx         ;     иначе уменьшаем его
        jmp     lp          ;     и продолжаем внешний цикл
quit:
; теперь результат подсчёта находится в EBX
```

3.2.14. Строковые операции

Для удобства работы с массивами (непрерывными областями памяти) процессор i386 вводит несколько команд, объединяемых в категорию ***строковых операций***. Именно эти команды используют регистры `ESI` и `EDI` в их особой роли, обсуждавшейся на стр. 39. Общая идея строковых команд состоит в том, что чтение из памяти выполняется по адресу из регистра `ESI`, запись в память — по адресу из регистра `EDI`, а затем эти регистры увеличиваются (или уменьшаются) в зависимости от команды на 1, 2 или 4. Некоторые команды производят чтение в регистр или запись в память из регистра; в этом случае используется регистр «аккумулятор» соответствующего размера, то есть регистр `AL`, `AX` или `EAX`. Строковые команды не имеют операндов, всегда используя одни и те же регистры.

«Направление» изменения адресов (движения вдоль строк) определяется флагом `DF` (напомним, его имя означает «direction flag», т. е. «флаг направления»). Если этот флаг сброшен, адреса увеличиваются (то есть строковая операция выполняется слева направо), если флаг установлен — адреса уменьшаются (работаем справа налево). Установить `DF` можно командой `std` (*set direction*), а сбросить — командой `cld` (*clear direction*).

Самые простые из строковых команд — команды `stosb`, `stosw` и `stosd`, которые записывают в память по адресу `[edi]` соответственно байт, слово или двойное слово из регистра `AL`, `AX` или `EAX`, после чего увеличивают или уменьшают (в зависимости от значения `DF`) регистр `EDI` на 1, 2 или 4. Например, если у нас есть массив

```
buf     resb 1024
```

и нам нужно заполнить его нулями, мы можем применить следующий код:

```
        xor al, al    ; обнуляем al
        mov edi, buf  ; адрес начала массива
        mov ecx, 1024 ; длина массива
        cld           ; работаем в прямом направлении
lp:     stosb         ; al -> [edi], увел. edi
        loop lp
```

Эти и другие строковые команды удобно использовать с ***префиксом*** `rep`. Команда, снабжённая таким префиксом, будет выполнена столько раз, какое число было в регистре `ECX` (кроме команды `stosw`: если её снабдить префиксом, то будет использоваться регистр `CX`; это обусловлено историческими причинами). С помощью префикса `rep` мы можем переписать вышеприведённый пример без использования метки:

```
        xor al, al
        mov edi, buf
        mov ecx, 1024
        cld
        rep stosb
```

Команды `lodsb`, `lodsw` и `lodsd`, наоборот, считывают байт, слово или двойное слово из памяти по адресу, находящемуся в регистре `ESI`, и помещают прочитанное в регистр `AL`, `AX` или `EAX`, после чего увеличивают или уменьшают значение регистра `ESI` на 1, 2 или 4. Использование этих команд с префиксом `rep` обычно бессмысленно, поскольку мы не сможем между последовательными исполнениями строковой команды вставить ещё какие-то действия, обрабатывающие значение, прочитанное и помещённое в регистр. Использование команд серии `lods` без префикса, напротив, может оказаться весьма полезным. Пусть, например, у нас есть массив четырёхбайтных чисел

```
array   resd 256
```

и нам необходимо сосчитать сумму его элементов. Это можно сделать следующим образом:

```
        xor ebx, ebx    ; обнуляем сумму
        mov esi, array
        mov ecx, 256
        cld
lp:     lodsd
        add ebx, eax
        loop lp
```

Часто оказывается удобным сочетание команд серии `lods` с соответствующими командами `stos`. Пусть, например, нам нужно увеличить на единицу все элементы того же самого массива. Это можно сделать так:

```
        mov esi, array
        mov edi, esi
        mov ecx, 256
        cld
lp:     lodsd
        inc eax
        stosd
        loop lp
```

Если же необходимо просто скопировать данные из одной области памяти в другую, очень удобны оказываются команды `movsb`, `movsw` и `movsd`. Эти команды копируют байт, слово или двойное слово из памяти по адресу `[esi]` в память по адресу `[edi]`, после чего увеличивают (или уменьшают) сразу оба регистра `ESI` и `EDI` соответственно на 1, 2 или 4. Например, если у нас есть два строковых массива

```
buf1    resb 1024
buf2    resb 1024
```

и нужно скопировать содержимое одного из них в другой, можно сделать это так:

```
mov ecx, 1024
mov esi, buf1
mov edi, buf2
cld
rep movsb
```

Благодаря возможности изменять направление работы (с помощью DF), мы можем производить копирование *частично перекрывающихся* областей памяти. Пусть, например, в массиве buf1 содержится строка "This is a string" и нам нужно перед словом "string" вставить слово "long". Для этого сначала нужно скопировать область памяти, начиная с адреса [buf1+10], на пять байт вперёд, чтобы освободить место для слова "long" и пробела. Ясно, что производить такое копирование мы можем только из конца в начало, иначе часть букв будет затёрта до того, как мы их скопируем. Таким образом, если слово "long " (вместе с пробелом) содержится в буфере buf2, то вставить его во фразу, находящуюся в buf1, мы можем так:

```
std
mov edi, buf1+17+5
mov esi, buf1+17
mov ecx, 8
rep movsb
mov esi, buf2+4
mov ecx, 5
rep movsb
```

Кроме перечисленных, процессор i386 реализует команды cmpsb, cmpsw и cmpsd (*compare string* — «сравнить строку»), а также scasb, scasw и scasd (*scan string* — «сканировать строку»). Команды серии scas сравнивают аккумулятор (соответственно AL, AX или EAX) с байтом, словом или двойным словом по адресу [edi], устанавливая флаги подобно команде cmp, и увеличивают/уменьшают EDI. Команды серии cmps сравнивают байты, слова или двойные слова, находящиеся в памяти по адресам [esi] и [edi], устанавливают флаги и увеличивают/уменьшают оба регистра.

Кроме префикса rep, можно воспользоваться также префиксами repz и repnz (также называемыми repe и repne), которые, кроме уменьшения и проверки регистра ECX (или CX, если команда двухбайтная) также проверяют значение флага ZF и продолжают работу только если

этот флаг установлен (repz/repe) или сброшен (repnz/repne). Обычно эти префиксы используют как раз в сочетании с командами серий scas и cmps. Например, если нам нужно найти букву 'a' в массиве символов mystr, имеющем размер mystr_len, мы можем действовать следующим образом:

```
mov esi, mystr
mov ecx, mystr_len
mov al, 'a'
cld
repnz scasb
```

Последняя строчка будет циклически сравнивать байт [esi] с регистром AL, в который мы занесли код нужной нам буквы, и остановится в двух случаях: если ECX дошел до нуля или если очередное сравнение показало равенство (взведён флаг ZF). Если после этого ECX будет равен нулю, то нужного символа в массиве не нашлось (мы дошли до конца массива). Также можно проверить, чему равно значение ячейки по адресу [esi].

3.2.15. Ещё несколько интересных команд

Прежде чем пойти дальше, рассмотрим ещё несколько команд.

На стр. 77 мы ввели команды std и cld, с помощью которых можно установить и сбросить флаг DF (флаг направления). Точно так же можно поступить с флагом переноса (CF): команда stc устанавливает его, а команда clc — сбрасывает; иногда это используется для передачи информации между разными частями программы. Как ни странно, для остальных известных нам флагов аналогичных команд не предусмотрено; существуют команды для управления привилегированными флагами, например cli и sti для IF (*interrupt flag*, но воспользоваться ими в ограниченном режиме нельзя.

Команда lahf копирует содержимое регистра флагов в регистр AH: флаг CF копируется в младший бит регистра (бит №0), флаг PF — в бит №2, флаг AF — в бит №4, флаг ZF — в бит №6 и, наконец, SF — в бит №7, то есть самый старший. Остальные биты остаются неопределёнными.

Команда xchg позволяет обменять местами значения двух своих операндов. В качестве одного из них выступает регистр AX или EAX, в качестве второго — регистр либо операнд типа «память», имеющий тот же размер. Операнд AX или EAX может быть указан первым или вторым — как легко догадаться, это ни на что не влияет. Когда оба операнда команды — регистры, в принципе никаких серьёзных возможностей эта команда не даёт, но если одним из операндов выступает область памяти, использование xchg позволяет *за одно неделимое действие занести в эту память некоторое значение, а то значение, которое там было раньше, сохранить в AX/EAX*. Почему это так важно, мы узнаем

из той части нашей книги, которая будет посвящена параллельному программированию.

Команды `movsx` (*move signed extension*, «перемещение со знаковым расширением») и `movzx` (*move zero extension*, «перемещение с расширением нулями») позволяют совместить копирование с увеличением разрядности. Обе команды имеют по два операнда, причём первый обязан быть регистровым, а второй может быть регистром или иметь тип «память», и в любом случае длина первого операнда должна быть вдвое больше длины второго, то есть можно копировать из байта в слово или из слова в двойное слово. Недостающие разряды команда `movzx` заполняет нулями, а команда `movsx` — значением старшего бита исходного операнда.

Довольно любопытна доступная на процессорах Pentium и более поздних команда `cpuid`, с помощью которой можно узнать, на какой модели процессора выполняется наша программа и какие возможности этот процессор поддерживает; подробное описание команды можно найти в Интернете или справочниках, здесь мы его приводить не будем, но о самом факте существования этой команды помнить полезно.

Также упомянем без подробного рассмотрения команды `xlat` (удобна при перекодировке текстовых данных через перекодировочную таблицу), `bswap` (позволяет переставить байты заданного 32-битного регистра в обратном порядке, впервые появилась в процессоре 80486), `aaa`, `aad`, `aam` и `aas` (позволяют производить арифметические операции над двоично-десятичными числами, в которых каждый полубайт представляет десятичную, а не шестнадцатеричную цифру).

Рассмотрение системы команд не может считаться законченным без команды `nop`. Она выполняет очень важное действие: *не делает ничего*. Само её название образовано от слов *no operation*.

3.3. Стек, подпрограммы, рекурсия

3.3.1. Понятие стека и его предназначение

Как мы уже знаем, под *стеком* в программировании подразумевают структуру данных, построенную по принципу «последний вошел — первый вышел» (англ. *last in first out*, LIFO), т. е. такой объект, над которым определены операции «добавить элемент» и «извлечь элемент», причём элементы, которые были добавлены, извлекаются в обратном порядке. В применении к низкоуровневому программированию понятие стека существенно у́же: здесь под стеком понимается непрерывная область памяти, для которой в специальном регистре хранится ***адрес вершины стека***; память в рассматриваемой области выше вершины (т. е. с адресами, меньшими адреса вершины) считается *свободной*, а

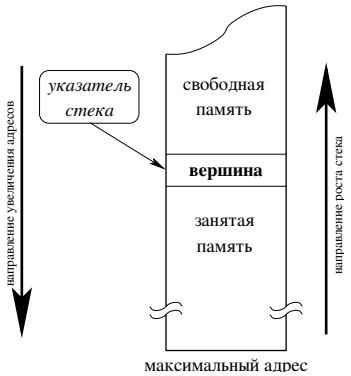

Рис. 3.4. Стек

память от вершины до конца области (до старших адресов), включая и саму вершину, считается *занятой*; регистр, хранящий адрес вершины, называется ***указателем стека*** (см. рис. 3.4). Операция добавления в стек некоторого значения уменьшает адрес вершины, сдвигая тем самым вершину вверх (то есть в направлении мéньших адресов) и в новую вершину записывает добавляемое значение; операция извлечения считывает значение с вершины стека и сдвигает вершину вниз, увеличивая её адрес.

Вообще говоря, направление роста стека зависит от конкретного процессора; на тех машинах, которые мы рассматриваем, и вообще на большинстве существующих архитектур стек «растёт вниз», то есть в направлении убывания адресов, но можно найти и такие процессоры, на которых стек «растёт вверх», и такие, где направление роста стека можно выбирать, и даже такие, где стек организован циклически.

Стек можно использовать, например, для временного хранения значений регистров; если некоторый регистр хранит важное для нас значение, а нам нужно временно задействовать этот регистр для хранения другого значения, то самый простой способ выйти из положения — сохранить значение регистра в стеке, затем использовать регистр под другие нужды, и, наконец, восстановить исходное значение регистра путём извлечения этого значения из стека обратно в регистр. Но гораздо более важно другое: **стек используется при вызовах подпрограмм для хранения адресов возврата, для передачи фактических параметров в подпрограммы и для хранения локальных переменных**. Именно использование стека позволяет реализовать механизм рекурсии, при котором подпрограмма может прямо или косвенно вызвать сама себя.

3.3.2. Организация стека в процессоре i386

Большинство существующих процессоров поддерживают работу со стеком на уровне машинных команд, и i386 в этом плане не исключение. Команды работы со стеком позволяют заносить в стек и извлекать из него слова и двойные слова; отдельные байты записывать в стек нельзя, так что адрес вершины стека всегда остаётся чётным. Более того, при работе в 32-битном режиме желательно всегда использовать в стеке двойные слова, сохраняя адрес вершины кратным четырём; работать всё будет и без этого, но команды работы со стеком станут выполняться медленнее.

Как уже говорилось (см. стр. 39), регистр `ESP`, формально относящийся к группе регистров общего назначения, тем не менее практически никогда не используется ни в какой иной роли, кроме роли ***указателя стека***; название этого регистра как раз и означает *stack pointer*. Считается, что адрес, содержащийся в `ESP`, указывает на вершину стека, то есть на ту область памяти, где хранится последнее занесённое в стек значение. Стек «растёт» в сторону уменьшения адресов, то есть при занесении в стек нового значения `ESP` уменьшается, при извлечении значения — увеличивается.

Занесение значения в стек производится командой `push`, имеющей один операнд. Этот операнд может быть непосредственным, регистровым или типа «память» и иметь размер `word` или `dword`; если операнд не регистровый, то размер придётся указать явно. Для извлечения значения из стека используется команда `pop`, операнд которой может быть регистровым или типа «память»; естественно, операнд должен иметь размер `word` или `dword`. Подчеркнём ещё раз, что двухбайтные операнды при работе со стеком использовать не следует; тем не менее, необходимо помнить про указание размера операнда.

Команды `push` и `pop` совмещают копирование данных (на вершину стека или с неё) со сдвигом самой вершины, то есть изменением значения регистра `ESP`. При необходимости можно обратиться к значению на вершине стека, не извлекая его из стека — применив (в любой команде, допускающей операнд типа «память») операнд `[esp]`. Например, команда

```
mov eax, [esp]
```

скопирует четырёхбайтное значение с вершины стека в регистр `EAX`.

Как говорилось выше, стек очень удобно использовать для временного хранения значений из регистров:

```
push eax   ; запоминаем eax
;  ... используем eax под посторонние нужды ...
pop eax    ; восстанавливаем eax
```

Рассмотрим более сложный пример. Пусть регистр ESI содержит адрес некоторой строки символов в памяти, причём известно, что строка заканчивается байтом со значением 0 (но не известно, какова длина строки) и нам необходимо «обратить» эту строку, то есть записать составляющие её символы в обратном порядке в том же месте памяти; нулевой байт, играющий роль ограничителя, естественно, остаётся при этом на месте и никуда не копируется. Один из способов сделать это — последовательно записать коды символов в стек, а затем снова пройти строку с начала в конец, извлекая из стека символы и записывая их в ячейки, составляющие строку.

Поскольку записывать в стек однобайтовые значения нельзя, а двухбайтовые можно, но нежелательно, мы будем записывать значения четырёхбайтовые, причём будем использовать только младший байт. Конечно, можно сделать всё более рационально, но нам сейчас важнее наглядность нашей иллюстрации. Для промежуточного хранения будем использовать регистр EBX, при этом только его младший байт (BL) будет содержать полезную информацию, но записывать в стек и извлекать из стека мы будем весь EBX целиком. Задача будет решена в два цикла. Перед первым циклом мы занесём ноль в регистр ECX, потом на каждом шаге будем извлекать байт по адресу [esi+ecx] и помещать этот байт (в составе двойного слова) в стек, а ECX увеличивать на единицу, и так до тех пор, пока очередной извлечённый байт не окажется нулевым, что по условиям задачи означает конец строки. В итоге все ненулевые элементы строки окажутся в стеке, а в регистре ECX будет длина строки.

Поскольку для второго цикла заранее известно количество его итераций (длина строки) и оно уже содержится в ECX, мы организуем этот цикл с помощью команды loop. Перед входом в цикл мы проверим, не пуста ли строка (то есть не равен ли ECX нулю), и если строка была пуста, сразу же перейдём в конец нашего фрагмента. Поскольку значение в ECX будет уменьшаться, а строку нам нужно пройти в прямом направлении — наряду с ECX мы воспользуемся регистром EDI, который в начале установим равным ESI (то есть указывающим на начало строки), а на каждой итерации будем его сдвигать. Итак, пишем:

```
        xor ebx, ebx        ; обнуляем ebx
        xor ecx, ecx        ; обнуляем ecx
lp:     mov bl, [esi+ecx]   ; очередной байт из строки
        cmp bl, 0           ; конец строки?
        je lpquit           ; если да - конец цикла
        push ebx            ; bl в составе ebx
        inc ecx             ; следующий индекс
        jmp lp              ; повторить цикл
lpquit: jecxz done          ; если строка пустая - конец
        mov edi, esi        ; опять с начала буфера
lp2:    pop ebx             ; извлекаем
```

```
        mov [edi], bl      ; записываем
        inc edi            ; следующий адрес
        loop lp2           ; повторять ecx раз
done:
```

3.3.3. Дополнительные команды работы со стеком

При необходимости можно занести в стек значение всех регистров общего назначения одной командой; эта команда называется `pushad` (*push all doublewords*, «затолкать все двойные слова»). Уточним, что эта команда заносит в стек содержимое регистров `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` и `EDI` (в указанном порядке), причём значение `ESP` заносится в том виде, в котором оно было *до* выполнения команды. Соответствующая команда извлечения из стека называется `popad` (*pop all doublewords*, «вытолкать все двойные слова»). Она извлекает из стека восемь четырёхбайтных значений и заносит эти значения в регистры в порядке, обратном приведённому для команды `pushad`, при этом значение, соответствующее регистру `ESP`, игнорируется (то есть из стека извлекается, но в регистр не заносится).

Регистр флагов (EFLAGS) может быть занесён в стек командой `pushfd` и извлечён командой `popfd`, однако при этом, если мы работаем в ограниченном режиме, только некоторые флаги (а именно — флаги, доступные к изменению в ограниченном режиме) могут быть изменены, на остальные команда `popfd` никак не повлияет.

Существуют аналогичные команды для 16-битных регистров, поддерживаемые для совместимости со старыми процессорами; они называются `pushaw`, `popaw`, `pushfw` и `popfw` и работают полностью аналогично, но вместо 32-битных регистров используют соответствующие 16-битные. Команды `pushaw` и `popaw` практически не используются, что касается команд `pushfw` и `popfw`, то их использование могло бы иметь смысл, если учесть, что в «расширенной» части регистра `EFLAGS` нет ни одного флага, значение которого мы могли бы поменять в ограниченном режиме работы; в реальности, впрочем, эти команды тоже не применяются, поскольку так можно нарушить выравнивание стека на адреса, кратные четырём, отчего работа со стеком замедлится.

3.3.4. Подпрограммы: общие принципы

Напомним, что ***подпрограммой*** называется обособленная часть программного кода, которая может быть *вызвана* из главной программы (или из другой подпрограммы); под ***вызовом*** понимается временная передача управления подпрограмме с тем, чтобы, когда подпрограмма сделает свою работу, она вернула управление в точку, откуда её вызвали. Мы уже встречались с подпрограммами в виде процедур и функций Паскаля.

При вызове подпрограммы нужно запомнить ***адрес возврата***, то есть адрес машинной команды, следующей за командой вызова подпрограммы, причём сделать это так, чтобы сама вызываемая подпрограмма могла, когда закончит свою работу, воспользоваться этим сохранённым адресом для возврата управления. Кроме того, подпрограммы часто получают ***параметры***, влияющие на их работу, и используют в работе ***локальные переменные***. Подо всё это требуется выделить оперативную память (или регистры). Самым простым решением было бы выделить каждой подпрограмме свою собственную область памяти под хранение всей локальной информации, включая и адрес возврата, и параметры, и локальные переменные. Тогда вызов подпрограммы потребует прежде всего записать в принадлежащую подпрограмме область памяти (в заранее оговорённые места) значения параметров и адрес возврата, а затем передать управление в начало подпрограммы.

Интересно, что когда-то давно именно так с подпрограммами и поступали, но с развитием методов и приёмов программирования возникла потребность в ***рекурсии*** — таком построении программы, при котором некоторые подпрограммы могут прямо или косвенно вызывать сами себя, притом потенциально неограниченное[27] число раз. Ясно, что при каждом рекурсивном вызове требуется новый экземпляр области памяти для хранения адреса возврата, параметров и локальных переменных, причём чем позже такой экземпляр будет создан, тем раньше соответствующий вызов закончит работу, то есть рекурсивные вызовы подпрограмм в определённом смысле подчиняются правилу «последний пришел — первый ушел». Совершенно логично из этого вытекает идея использования при вызовах подпрограмм уже знакомого нам стека.

В современных вычислительных системах перед вызовом подпрограммы в стек помещаются значения параметров вызова, затем производится собственно вызов, то есть передача управления, которая совмещена с сохранением в том же стеке адреса возврата. Наконец, когда подпрограмма получает управление, она резервирует в стеке определённое количество памяти для хранения локальных переменных, обычно просто сдвигая адрес вершины вниз на соответствующее количество ячеек. Область стековой памяти, содержащую связанные с одним вызовом значения параметров, адрес возврата и локальные переменные, называют ***стековым фреймом***.

3.3.5. Вызов подпрограмм и возврат из них

Вызов подпрограммы, как уже ясно из вышесказанного,— это передача управления по адресу начала подпрограммы с одновременным запоминанием в стеке адреса возврата, то есть адреса машинной команды, непосредственно следующей за командой вызова. Процессор i386

[27] Точнее говоря, ограниченное только объемом памяти.

предусматривает для этой цели команду `call`; аналогично команде `jmp`, аргумент команды `call` может быть непосредственным (адрес перехода задан непосредственно в команде, например, меткой; как и для команды `jmp`, в машинном коде используется *расстояние* от текущей позиции, т. е. применяется относительная адресация), регистровым (адрес передачи управления находится в регистре) и типа «память» (переход нужно осуществить по адресу, прочитанному из заданного места памяти). Команда `call` не имеет «короткой» формы; поскольку «дальняя» форма нам, как обычно, не требуется в силу отсутствия сегментов, остаётся только одна форма — «близкая» (`near`), которую мы всегда и используем.

Возврат из подпрограммы производится командой `ret` (от слова *return* — «возврат»). В своей простейшей форме эта команда не имеет аргументов. Выполняя эту команду, процессор извлекает четыре байта с вершины стека и записывает их в регистр `EIP`, в результате чего управление передаётся по адресу, который находился в памяти на вершине стека.

Рассмотрим простой пример. Допустим, в нашей программе часто приходится заполнять каким-то однобайтовым значением области памяти разной длины. Такое действие вполне можно оформить в виде подпрограммы. Для простоты картины примем соглашение, что адрес нужной области памяти передаётся через регистр `EDI`, количество однобайтовых ячеек, которые нужно заполнить — через регистр `ECX`, ну а само значение, которое надо записать во все эти ячейки — через регистр `AL`. Код соответствующей подпрограммы может выглядеть, например, так:

```
; fill memory (edi=address, ecx=length, al=value)
fill_memory:
        jecxz   fm_q
fm_lp:  mov     [edi], al
        inc edi
        loop fm_lp
fm_q:   ret
```

Обратиться к такой подпрограмме можно, например, так:

```
        mov edi, my_array
        mov ecx, 256
        mov al, '@'
        call fill_memory
```

В результате такого вызова 256 байт памяти, начиная с адреса, заданного меткой `my_array`, окажутся заполнены кодом символа `'@'` (число 64).

3.3.6. Организация стековых фреймов

Подпрограмма, приведённая в качестве примера в предыдущем параграфе, фактически не использовала механизм стековых фреймов, сохраняя в стеке только адрес возврата. Этого оказалось достаточно, поскольку подпрограмме не требовались локальные переменные, а параметры мы передали через регистры. Как показывает практика, подпрограммы редко бывают такими простыми. В более сложных случаях нам наверняка потребуются локальные переменные, поскольку регистров на всё не хватит. Кроме того, передача параметров через регистры тоже может оказаться неудобна: во-первых, их не всегда хватает, а во-вторых, подпрограмме могут быть долго нужны значения, переданные через регистры, и это фактически лишит её возможности использовать под свои внутренние нужды те из регистров, которые были задействованы при передаче параметров. Кроме того, передача параметров через регистры (а равно и через какую-либо фиксированную область памяти) лишает нас возможности использовать рекурсию, что тоже, разумеется, плохо.

Поэтому обычно, в особенности при трансляции программы с какого-либо языка высокого уровня, с того же Паскаля или Си, параметры в функции передаются через стек, и в стеке же размещаются локальные переменные. Как было сказано выше, параметры в стеке размещает вызывающая программа, затем при вызове подпрограммы в стек заносится адрес возврата, а затем уже сама вызванная подпрограмма резервирует место в стеке под локальные переменные. Всё это вместе как раз и образует стековый фрейм. К содержимому стекового фрейма можно обращаться, используя адреса, «привязанные» к адресу, по которому содержится адрес возврата; иначе говоря, ту ячейку памяти, начиная с которой в стек был занесён адрес возврата, используют в качестве своего рода реперной точки. Так, если в стек занести три четырёхбайтных параметра, а потом вызвать процедуру, то адрес возврата будет лежать в памяти по адресу `[esp]`, ну а параметры, очевидно, окажутся доступны по адресам `[esp+4]`, `[esp+8]` и `[esp+12]`. Если же разместить в стеке локальные четырёхбайтные переменные, то они окажутся доступны по адресам `[esp-4]`, `[esp-8]` и т. д.

Заметим, что использовать для доступа к параметрам регистр `ESP` оказывается не слишком удобно, ведь в самой процедуре нам тоже может потребоваться стек — как для временного хранения данных, так и для вызова других подпрограмм. Поэтому первым же своим действием подпрограмма обычно сохраняет значение регистра `ESP` в каком-то другом регистре (чаще всего `EBP`) и именно его использует для доступа к параметрам и локальным переменным, ну а регистр `ESP` продолжает играть свою роль указателя стека, изменяясь по мере необходимости; перед возвратом из подпрограммы его обычно восстанавливают в ис-

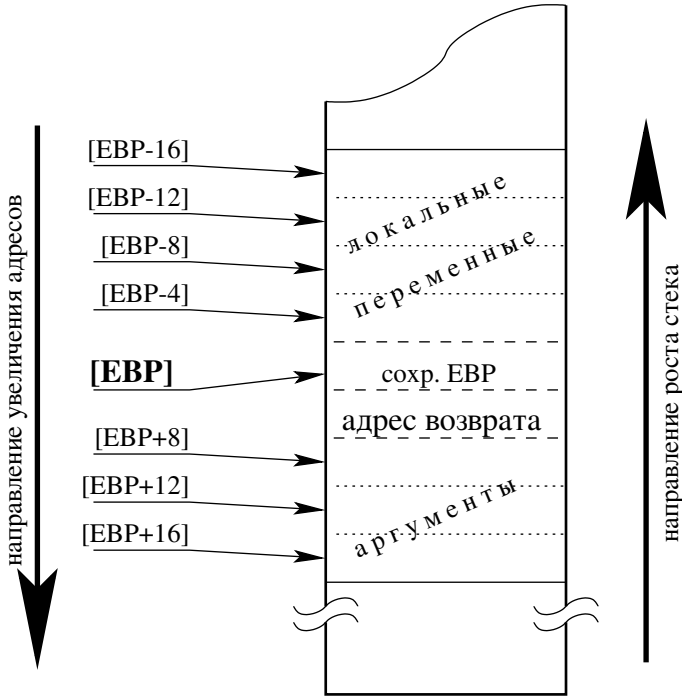

Рис. 3.5. Структура стекового фрейма

ходном значении, попросту пересылая в него значение из `EBP`, чтобы он снова указывал на адрес возврата.

Наконец, возникает ещё один вопрос: а что если другие подпрограммы тоже используют регистр `EBP` для тех же целей? Ведь в этом случае первый же вызов другой подпрограммы испортит нам всю работу. Можно, конечно, сохранять `EBP` в стеке перед вызовом каждой подпрограммы, но поскольку в программе обычно гораздо больше *вызовов подпрограмм*, чем самих подпрограмм, экономнее оказывается следовать простому правилу: *каждая подпрограмма должна сама сохранить старое значение `EBP` и восстановить его перед возвратом управления*. Естественно, для сохранения значения `EBP` тоже используется стек; сохранение выполняется простой командой `push ebp` сразу после получения управления. Таким образом, старое значение `EBP` помещается в стек непосредственно после адреса возврата из подпрограммы, и в качестве «точки привязки» используется в дальнейшем именно этот адрес вершины стека. Для этого следующей командой выполняется `mov ebp,esp`. В результате регистр `EBP` указывает на то место в стеке, где находится его же, `EBP`, сохранённое значение; если теперь обратиться к памяти по адресу `[ebp+4]`, мы обнаружим там адрес возврата

из подпрограммы, ну а параметры, занесённые в стек перед вызовом подпрограммы, оказываются доступны по адресам `[esp+8]`, `[esp+12]`, `[esp+16]` и т. д. Память под локальные переменные выделяется путём простого вычитания нужной длины из текущего значения `ESP`; так, если под локальные переменные нам требуется 16 байт, то сразу после сохранения `EBP` и копирования в него содержимого `ESP` нужно выполнить команду `sub esp, 16`; если (для простоты картины) все наши локальные переменные тоже занимают по 4 байта, они окажутся доступны по адресам `[ebp-4]`, `[ebp-8]` и т. д. Структура стекового фрейма с тремя четырёхбайтными параметрами и четырьмя четырёхбайтными локальными переменными показана на рис. 3.5.

Повторим, что в начале своей работы, согласно нашим договорённостям, каждая подпрограмма должна выполнить

```
push ebp
mov ebp, esp
sub esp, 16   ; вместо 16 подставьте объём
              ; памяти под локальные переменные
```

Завершение подпрограммы теперь должно выглядеть так:

```
mov esp, ebp
pop ebp
ret
```

Интересно, что процессор i386 поддерживает даже специальные команды для обслуживания стековых фреймов. Так, в начале подпрограммы вместо трёх команд, приведённых выше, можно было бы дать одну команду «`enter 16, 0`», а вместо двух команд перед `ret` можно было бы написать `leave`. Проблема, как ни странно, в том, что команды `enter` и `leave` работают *медленнее*, чем соответствующий набор простых команд, так что их практически никогда не используют; если дизассемблировать машинный код, сгенерированный компилятором языка Си или Паскаль, мы, скорее всего, обнаружим в начале любой процедуры или функции именно такие команды, как показано выше, и ничего похожего на `enter`. Единственным оправданием существования команд `enter` и `leave` может служить их короткая запись (например, машинная команда `leave` занимает в памяти всего один байт), но в наше время об экономии памяти на машинном коде обычно никто не задумывается; быстродействие, как правило, оказывается важнее.

Сделаем ещё одно важное замечание. При работе под управлением ОС Unix мы можем не беспокоиться ни о наличии стека, ни о задании его размера. Операционная система создаёт стек автоматически при запуске любой задачи, а уже во время её исполнения при необходимости увеличивает размер доступной для стека памяти: по мере того как вершина стека продвигается по виртуальному адресному пространству «вверх» (то есть в сторону уменьшения адресов), операционная система ставит в соответствие виртуальным адресам всё новые и новые страницы

физической памяти. Именно поэтому на рис. 3.4 и 3.5 мы изобразили верхний край стека как нечто нечёткое.

3.3.7. Основные конвенции вызовов подпрограмм

Несмотря на подробное описание механизма стековых фреймов, данное в предыдущем параграфе, в некоторых вопросах остаётся возможность для манёвра. Так, например, в каком порядке следует заносить в стек значения параметров подпрограммы? Если мы пишем программу на языке ассемблера, этот вопрос, собственно говоря, не встаёт; однако он оказывается неожиданно принципиальным при создании компиляторов языков программирования высокого уровня.

Создатели классических компиляторов языка Паскаль[28] обычно шли «очевидным» путём: вызов процедуры или функции транслировался с Паскаля в виде серии команд занесения в стек значений, причём значения заносились в естественном (для человека) порядке — слева направо; затем в код вставлялась команда `call`. Когда такая процедура получает управление, значения фактических параметров располагаются в стеке снизу вверх, то есть *последний* параметр оказывается размещён ближе других к реперной точке фрейма (доступен по адресу `[ebp+8]`). Из этого, в свою очередь, следует, что **для доступа к первому (а равно и к любому другому) параметру процедура или функция языка Паскаль должна знать общее количество этих параметров**, поскольку расположение n-го параметра в стековом фрейме получается зависящим от общего количества. Так, если у процедуры три четырёхбайтных параметра, то первый из них окажется в стеке по адресу `[ebp+16]`, если же их пять, то первый придётся искать по адресу `[ebp+24]`. Именно поэтому язык Паскаль не допускает создания процедур или функций с переменным числом аргументов, так называемых ***вариадических*** подпрограмм (что вполне нормально для учебного языка, но не совсем приемлемо для языка профессионального). Входящие в язык Паскаль псевдопроцедуры с переменным числом аргументов, такие как `WriteLn`, на самом деле являются частью самого языка Паскаль; компилятор трансформирует их вызовы в нечто весьма далёкое от вызова подпрограммы на уровне машинного кода. Так или иначе, программист не может на Паскале описать свою процедуру подобного рода.

Создатели языка Си пошли иным путём. При трансляции вызова функции языка Си параметры помещаются в стек *в обратном порядке* и оказываются размещёнными во фрейме в порядке сверху вниз, так

[28] Следует отметить, что компиляторы Паскаля не обязаны действовать именно так; например, Free Pascal, который мы рассматривали в первом томе, старается передать параметры через регистры, и только если регистров не хватает — размещает оставшиеся параметры в стеке, но при этом порядок действительно используется «прямой».

что первый параметр всегда оказывается доступен по адресу `[ebp+8]`, второй — по адресу `[ebp+12]` и т. д., вне всякой зависимости от общего количества параметров (конечно, параметры по крайней мере должны присутствовать, то есть если функция, например, вызвана вообще без параметров, никакого первого параметра в стеке не будет). Это, с одной стороны, позволяет создание вариадических функций; в частности, в сам по себе язык Си не входит вообще ни одной функции, что же касается таких функций, как `printf`, `scanf` и др., то они реализуются в *библиотеке*, а не в самом языке; более того, сами эти функции тоже написаны на Си (как сказано выше, на Паскале так сделать не получается).

С другой стороны, отсутствие в Паскале вариадических подпрограмм позволяет возложить заботы об очистке стека *на вызываемого*. Действительно, подпрограмма языка Паскаль всегда знает, сколько места занимают фактические параметры в её стековом фрейме (поскольку для каждой подпрограммы это количество задано раз и навсегда и не может измениться) и, соответственно, может принять на себя заботу об очистке стека. Как уже говорилось, вызовов подпрограмм в любой программе больше, чем самих подпрограмм, так что перекладыванием заботы об очистке стека с вызывающего на вызываемого достигается определённая экономия памяти (количества машинных команд). При использовании соглашений языка Си такая экономия невозможна, поскольку подпрограмма в общем случае[29] не знает и не может знать, сколько параметров ей передали, так что забота об очистке стека от параметров остаётся на вызывающем; обычно это делается простым увеличением значения `ESP` на число, равное совокупной длине фактических параметров. Например, если подпрограмма `proc1` принимает на вход три четырёхбайтных параметра (назовём их `a1`, `a2` и `a3`), её вызов будет выглядеть примерно так:

```
push dword a3  ; заносим в стек параметры
push dword a2
push dword a1
call proc1     ; вызываем подпрограмму
add esp, 12    ; убираем параметры из стека
```

В случае же использования соглашений языка Паскаль последняя команда (`add`) оказывается не нужна, обо всём позаботится вызываемый. Процессор i386 даже имеет для этого специальную форму команды `ret` с одним операндом (выше в примерах мы использовали `ret` без операндов). Этот операнд, который может быть только непосредственным

[29] В разных ситуациях используются различные способы фиксации количества параметров; так, функция `printf` узнаёт, сколько параметров нужно извлечь из стека, путём анализа форматной строки, а функция `execlp` извлекает аргументы, пока не наткнётся на нулевой указатель, но и то и другое — лишь частные случаи.

и всегда имеет длину два байта («слово»), задаёт количество памяти (в байтах), занятой параметрами функции. Например, процедуру, принимающую через стек три четырёхбайтных параметра, компилятор Паскаля закончит командой

```
ret 12
```

Эта команда, как и обычная команда `ret`, извлечёт из стека адрес возврата и передаст по нему управление, но кроме этого (одновременно с этим) увеличит значение `ESP` на заданное число (в данном случае 12), избавляя, таким образом, вызвавшего от обязанности по очистке стека.

3.3.8. Локальные метки

Прежде чем мы приведём пример подпрограммы, выполняющей рекурсивный вызов, необходимо рассмотреть ещё одно важное средство, предоставляемое ассемблером NASM — ***локальные метки***. Суть и основное достоинство подпрограмм состоит в их *обособленности*. Иначе говоря, в процессе написания одной подпрограммы мы обычно не помним, как изнутри устроены другие подпрограммы и воспринимаем каждую из подпрограмм, кроме одной (той, что пишется прямо сейчас) в виде этакой одной большой команды. Это позволяет не держать в голове лишних деталей и сосредоточиться на реализации конкретного фрагмента программы, а по окончании такой реализации выкинуть её детали из головы и перейти к другому фрагменту.

Проблема в том, что в теле любой мало-мальски сложной подпрограммы нам обязательно понадобятся метки, и нужно сделать так, чтобы при выборе имён для таких меток нам не нужно было вспоминать, есть ли уже где-нибудь (в другой подпрограмме) метка с таким же именем.

Ассемблер NASM для этого предусматривает специальные *локальные* метки. Синтаксически эти метки отличаются от обычных тем, что начинаются с точки. Ассемблер локализует такие метки во фрагменте программы, ограниченном с обеих сторон обычными (нелокальными) метками. Иначе говоря, локальную метку ассемблер рассматривает не саму по себе, а как нечто подчинённое последней (ближайшей сверху) нелокальной метке. Например, в следующем фрагменте:

```
first_proc:
        ; ... ...
.cycle:
        ; ... ...
second_proc:
        ; ... ...
.cycle:
```

```
        ; ... ...
third_proc:
```

первая метка `.cycle` подчинена метке `first_proc`, а вторая — метке `second_proc`, так что между собой они не конфликтуют. Если метка `.cycle` встретится в параметрах той или иной команды между метками `first_proc` и `second_proc`, ассемблер будет знать, что имеется в виду именно первая из меток `.cycle`, если она встретится после `second_proc`, но перед `third_proc` — то задействуется вторая, тогда как появление метки `.cycle` до `first_proc` или после `third_proc` будет рассматриваться как ошибка. Если каждую подпрограмму начинать с обычной метки, а внутри подпрограммы использовать только локальные метки, то в разных подпрограммах мы можем использовать локальные метки с одинаковыми именами, и ассемблер в них не запутается.

На самом деле ассемблер достигает такого эффекта не очень честным путём: видя метку, имя которой начинается с точки, он просто добавляет к ней спереди имя последней встречавшейся ему метки без точки. Так, в примере выше речь идёт не о двух одинаковых метках `.cycle`, а о двух *разных* метках `first_proc.cycle` и `second_proc.cycle`. Полезно помнить об этом и не применять в программе в явном виде метки, содержащие точку, хотя ассемблер это допускает.

3.3.9. Пример

Приведём пример подпрограммы, использующей рекурсию. Одна из простейших классических задач, решаемых рекурсивно — это сопоставление строки с образцом, её мы и используем. Отметим, что на Паскале мы эту задачу уже решали; подробное описание алгоритма решения дано в первом томе, §2.14.3, так что здесь мы ограничимся краткими замечаниями.

Уточним задачу с учётом использования «низкоуровневых» строк. Даны две строки символов, длина которых заранее неизвестна, но известно, что каждая из них ограничена нулевым байтом (отметим, что в Паскале строки были устроены иначе). Первую строку мы рассматриваем как *сопоставляемую*, вторую — как *образец*. В образце символ `'?'` может сопоставляться с произвольным символом, символ `'*'` — с произвольной *подцепочкой символов* (возможно, даже пустой), остальные символы обозначают сами себя и сопоставляются только сами с собой. Требуется определить, соответствует ли (целиком) заданная строка заданному образцу, и вернуть результат `0`, если не соответствует, и результат `1`, если соответствует.

Как мы убедились, решив задачу на Паскале, рекурсивный алгоритм для неё оказывается довольно простым. На каждом шаге (точнее, на каждом рекурсивном вызове) мы рассматриваем *оставшуюся часть* строки и образца; сначала эти оставшиеся части совпадают со строкой

и образцом, затем, по мере продвижения алгоритма, от них отбрасываются символы, стоящие в начале, и мы предполагаем, что для уже отброшенных символов сопоставление прошло успешно.

В зависимости от первого символа образца наш алгоритм работает по одному из трёх основных сценариев. Если вместо первого символа мы видим в образце ограничительный ноль, то есть образец у нас кончился, то алгоритм на этом заканчивается и немедленно выдаёт результат: «истину», если сопоставляемая строка тоже кончилась, в противном случае «ложь»; в самом деле, с пустым образцом можно сопоставить только пустую строку.

Если образец ещё не кончился и первым символом в нём находится любой символ, кроме `'*'`, то необходимо произвести сопоставление первого символа образца с первым символом строки с учётом того, что строка должна быть непустая, а символ `'?'` в образце успешно сопоставляется с любым символом в строке. Если сопоставление не проходит (то есть либо в строке первым символом ограничительный ноль, либо в образце **не** знак вопроса, и при этом символ в образце не равен символу в строке), то возвращаем «ложь»; в противном случае сопоставляем остатки строки и образца, отбросив первые символы, и результат сопоставления возвращаем в качестве своего результата.

Наконец, если первый символ образца оказался символом `'*'`, нужно последовательно перебрать возможности сопоставления этой «звёздочки» с пустой подцепочкой строки, с одним символом строки, с двумя символами и т. д., пока не кончится сама строка. Делаем мы это следующим образом. Заводим целочисленную переменную `I`, которая будет у нас обозначать текущий рассматриваемый вариант. Присваиваем этой переменной ноль (начинаем рассмотрение с пустой цепочки). Теперь для каждой рассматриваемой альтернативы отбрасываем от образца один символ (звёздочку), а от строки — столько символов, какое сейчас число в переменной `I`. Полученные остатки пытаемся сопоставить, используя для этого рекурсивный вызов «самих себя». Если результат вызова — «истина», завершаем работу, тоже вернув истину; если же результат — «ложь», проверяем, можно ли ещё увеличивать переменную `I` (не выйдем ли мы при этом за пределы сопоставляемой строки). Если увеличиваться уже некуда, завершаем работу, вернув «ложь»; в противном случае возвращаемся к началу цикла и рассматриваем следующее возможное значение `I`.

Программа, написанная нами ранее на Паскале, использовала паскалевское представление строк и из-за этого сильно отличалась от того решения, которое мы сейчас напишем на языке ассемблера: например, там приходилось использовать четыре параметра для подпрограммы, тогда как здесь параметров будет всего два. Забегая вперёд, отметим, что решение на языке Си будет повторять «ассемблерное» практически слово в слово; соответствующую программу на Си можно найти в §4.7.2.

Реализацию на языке ассемблера мы выполним в виде подпрограммы, которую назовём `match`. Подпрограмма будет предполагать, что ей передано два параметра — адрес строки (`[ebp+8]`) и адрес образца (`[ebp+12]`); сама подпрограмма будет использовать одну четырёхбайтную переменную для хранения `I`; под неё будет выделяться место в стековом фрейме, так что она будет располагаться по адресу `[ebp-4]`. Для увеличения скорости работы наша подпрограмма в самом начале скопирует адреса из параметров в регистры `ESI` (адрес строки) и `EDI` (адрес образца). Кроме того, для выполнения арифметических действий подпрограмма будет использовать регистр `EAX`. Через него же она будет возвращать результат своей работы: число `0` как обозначение логической лжи (соответствие не найдено) или число `1` как обозначение логической истины (соответствие найдено).

«Отбрасывание» символов из начала строк мы будем производить простым увеличением рассматриваемого адреса строки: действительно, если по адресу `string` находится строка, мы можем считать, что по адресу `string+1` находится та же строка, кроме первой буквы.

Подпрограмма будет рекурсивно вызывать саму себя, и, будучи вызванной рекурсивно, должна будет выполнять работу над значениями, отличающимися от тех, что были заданы в предыдущем вызове. При этом регистры в качестве хранилища локальных данных понадобятся как исходному вызову подпрограммы, так и «вложенному» (рекурсивному), но в процессоре только один набор регистров, и нужно сделать так, чтобы разные «экземпляры» работающей подпрограммы друг другу не мешали.

Возможны различные соглашения о том, какие из регистров подпрограмма имеет право «испортить», а какие должна после себя оставить в том виде, в котором они находились на момент её вызова. В качестве крайних можно рассматривать варианты, когда подпрограмме разрешается портить *все* регистры, либо когда ей не разрешается портить никакие регистры, кроме, возможно, `EAX`, через который возвращается результат. Первый случай вынуждает нас в любом месте программы при вызове любых подпрограмм сначала сохранять в стеке все регистры, в которых у нас хранилось что-нибудь важное; это не совсем хороший вариант, поскольку большинству небольших подпрограмм *все* регистры не требуются, так что наши сохранения и восстановления окажутся лишней работой, снижающей эффективность программы. С другой стороны, наложение на все подпрограммы требования по восстановлению всех регистров тоже при внимательном рассмотрении оказывается избыточным, хотя и не настолько.

Довольно удачным оказывается компромисс, известный как ***конвенция CDECL*** и используемый большинством компиляторов Си на платформе x86. Согласно этой конвенции подпрограмма имеет право портить `EAX`, `EDX` и `ECX`, а все остальные регистры общего назначения

должна либо не трогать, либо сохранить и перед возвратом управления восстановить. Такой выбор регистров не случаен. Дело в том, что EAX портится то и дело, поскольку многие операции можно производить только через него; для долгосрочного хранения данных его поэтому никто не применяет, и, кстати, именно по этой причине EAX используется для возврата значения из функций. Регистр EDX часто задействуется совместно с EAX, либо в качестве «старшей части» 64-битного числа, либо для хранения остатка от деления; любая операция умножения или деления испортит этот регистр, так что и он не подходит для длительного хранения информации. Наконец, ECX используется в качестве счётчика во всевозможных циклах, которые тоже встречаются очень часто. С учётом сказанного именно эти три регистра приходится портить чаще остальных и, как следствие, потеря находящихся в них значений создаёт наименьшее количество проблем для вызывающего.

Наша функция будет работать в соответствии с CDECL: «портить» она будет значения регистров EBP, ESI, EDI и EAX, но EAX в любом случае «испортится», поскольку через него мы возвращаем итоговое значение, так что сохранять нужно только ESI, EDI и, конечно, EBP. На первый взгляд, мы могли бы слегка сэкономить, используя вместо ESI/EDI, например, ECX/EDX, которые согласно CDECL имеем право испортить, но ведь вызываем-то мы сами себя, так что нам всё равно пришлось бы эти регистры сохранять, но не в начале процедуры, а перед рекурсивным обращением к самим себе.

Текст нашей подпрограммы получается таким:

```
match:                          ; НАЧАЛО ПОДПРОГРАММЫ
        push ebp                ; организуем стековый фрейм
        mov ebp, esp
        sub esp, 4              ; локальная переменная I
                                ; будет по адресу [ebp-4]
        push esi                ; сохраняем регистры ESI и EDI
        push edi                ;     (EAX всё равно изменится)
        mov esi, [ebp+8]        ; загружаем параметры: адреса
        mov edi, [ebp+12]       ;     строки и образца
.again:                         ; сюда мы вернёмся, когда
                                ;     сопоставим очередной
                                ;     символ и сдвинемся
        cmp byte [edi], 0       ; образец кончился?
        jne .not_end            ;     если нет, прыгаем
        cmp byte [esi], 0       ; образец кончился, а строка?
        jne near .false         ; если нет, то вернуть ЛОЖЬ
        jmp .true               ; кончились одновременно: ИСТИНА
.not_end:                       ; если образец не кончился...
        cmp byte [edi], '*'     ; не звёздочка ли в его начале?
        jne .not_star           ; если нет, прыгаем отсюда
                                ; звёздочка! организуем цикл
```

```
        mov dword [ebp-4], 0  ; I := 0
.star_loop:
                              ; готовимся к рекурс. вызову
        mov eax, edi          ; сначала второй аргумент:
        inc eax               ;      образец со след. символа
        push eax
        mov eax, esi          ; теперь первый аргумент:
        add eax, [ebp-4]      ;      строка с I-го символа
        push eax              ;  (напомним, [ebp-4] - это I)
        call match            ; вызываем сами себя, но
                              ;      с новыми параметрами
        add esp, 8            ; после вызова очищаем стек
        test eax, eax         ; что нам вернули?
        jnz .true             ; вернули не ноль, т.е. ИСТИНУ
                              ; значит, остаток строки
                              ; сопоставился с остатком
                              ; образца => вернём ИСТИНУ
        add eax, [ebp-4]      ; вернули 0, т.е. ЛОЖЬ
                              ; надо попробовать больше
                              ; символов "списать" на
                              ; эту звёздочку
        cmp byte [esi+eax], 0 ; но, быть может, строка
                              ;      уже кончилась?
        je .false             ; тогда пробовать больше нечего
        inc dword [ebp-4]     ; иначе пробуем: I := I + 1
        jmp .star_loop        ;      и в начало цикла по I
.not_star:                    ; сюда мы попадаем, если обр.
        mov al, [edi]         ;      не пуст и не нач. с '*'
        cmp al, '?'           ; может быть, там знак '?'
        je .quest             ; если да, прыгаем отсюда
        cmp al, [esi]         ; если нет, символы в начале
                              ;      строки и образца должны
                              ;      совпадать; если строка
                              ;      кончилась, эта проверка
                              ;      тоже не пройдёт
        jne .false            ; не совпали (или кон. строки)
                              ;      => возвращаем ЛОЖЬ
        jmp .goon             ; совпали - продолжаем
                              ;      просмотр
.quest:                       ; образец начинается с '?'
        cmp byte [esi], 0     ; надо только, чтобы строка не
        jz .false             ;      кончилась (иначе ЛОЖЬ)
.goon:  inc esi               ; символы сопоставились =>
        inc edi               ;      сдвигаемся по строке и
        jmp .again            ;      образцу и продолжаем
.true:                        ; сюда мы прыгали, чтобы
        mov eax, 1            ;      вернуть ИСТИНУ
        jmp .quit
```

```
.false:                         ; а сюда прыгали, чтобы
        xor eax, eax            ;       вернуть ЛОЖЬ
.quit:                          ; всё, конец работы
        pop edi                 ;       приводим всё в
        pop esi                 ;       порядок перед
        mov esp, ebp            ;       возвратом управления
        pop ebp                 ; результат у нас в EAX
        ret                     ; Возвращаем управление
                                ; КОНЕЦ ПРОЦЕДУРЫ
```

Если, например, ваша строка располагается в памяти, помеченной меткой `string`, а образец — в памяти, помеченной меткой `pattern`, то вызов подпрограммы `match` будет выглядеть вот так:

```
push dword pattern
push dword string
call match
add esp, 8
```

После этого результат сопоставления (0 или 1) окажется в регистре `EAX`. Текст этого примера вместе с главной программой, использующей параметры командной строки, читатель найдёт в файле `match.asm`.

Обратите внимание, что в начале подпрограммы при попытке перейти на метку `.false` мы были вынуждены явно указать, что переход является «ближним» (`near`). Дело в том, что метка `.false` оказалась чуть дальше от команды перехода, чем это допустимо для «короткого» перехода. См. обсуждение на стр. 63.

3.4. Основные особенности ассемблера NASM

Ранее мы использовали ассемблер NASM, ограничиваясь лишь общими замечаниями и изредка отвлекаясь, чтобы описать некоторые его возможности, без которых не могли обойтись. Так, в §3.1.4 было дано ровно столько пояснений, чтобы можно было понять одну простейшую программу. Позже нам потребовалось использовать память для хранения данных, и пришлось посвятить §3.2.3 директивам резервирования памяти и меткам. Прежде чем привести в §3.3.9 пример сложной подпрограммы, мы вынуждены были в §3.3.8 рассказать про локальные метки.

Эту главу мы целиком посвятим изучению ассемблера NASM, начав с краткого описания ключей командной строки, используемых при его запуске, и продолжив более формальным, чем раньше, описанием синтаксиса его языка. После этого мы отдельную главу посвятим макропроцессору.

3.4.1. Ключи и опции командной строки

Как уже говорилось, при вызове программы `nasm` необходимо указать имя файла, содержащего исходный текст на языке ассемблера, а кроме этого, обычно требуется указать *ключи*, задающие режим работы. С некоторыми из них мы уже знакомы: это ключи `-f`, `-o` и `-d`.

Напомним, что ключ `-f` позволяет указать *формат* получаемого кода. В нашем случае всегда используется формат `elf`. Интересно, что, если не указать этот ключ, ассемблер создаст выходной файл в «сыром» формате, то есть, попросту говоря, переведёт наши команды в двоичное представление и в таком виде запишет в файл. Работая под управлением операционных систем, мы такой файл запустить на выполнение не сможем, однако если бы мы, к примеру, хотели написать программу для размещения в загрузочном секторе диска, то «сырой» формат оказался бы как раз тем, что нам нужно.

Ключ `-o` задаёт имя файла, в который следует записать результат трансляции. Если мы используем формат `elf`, то вполне можем доверить выбор имени файла самому NASM'у: он отбросит от имени исходного файла суффикс `.asm` и заменит его на `.o`, что нам в большинстве случаев и требуется. Если же по каким-то причинам нам удобнее другое имя, мы можем указать его явно с помощью `-o`.

Ключ `-d` нам понадобится после изучения макропроцессора; он используется для определения макросимвола в случае, если мы не хотим делать этого путём редактирования исходного текста. Например, `-dSYMBOL` даёт тот же эффект, что и вставленная в начало программы строка `%define SYMBOL`, а `-dSIZE=1024` не только определит символ `SIZE`, но и припишет ему значение `1024`, как это сделала бы директива `%define SIZE 1024`. Мы вернёмся к этому на стр. 117.

Очень интересны в познавательном плане возможности генерации так называемого ***листинга*** — подробного отчёта ассемблера о проделанной работе. Листинг включает в себя строки исходного кода, снабжённые информацией об используемых адресах и о том, какой итоговый код сгенерирован в результате обработки каждой исходной строки. Генерация листинга запускается ключом `-l`, после которого требуется указать имя файла. Для примера возьмите любую программу на языке ассемблера и оттранслируйте её с флагом `-l`; так, если ваша программа называется `prog.asm`, попробуйте применить команду

```
avst@host:~/work$ nasm -f elf -l prog.lst prog.asm
```

в результате которой текст листинга будет помещён в файл `prog.lst`. Обязательно просмотрите получившийся файл и разберитесь, что там к чему; если что-то окажется непонятно, найдите кого-нибудь, кто сможет вам помочь разобраться.

Весьма полезным может оказаться ключ `-g`, указывающий NASM'у на необходимость включения в результаты трансляции так называемой *отладочной информации*. При указании этого ключа NASM вставляет в объектный файл помимо объектного кода ещё и сведения об имени исходного файла, номерах строк в нём и т. п. Для работы программы вся эта информация совершенно бесполезна, тем более что по объёму она может в несколько раз превышать «полезный» объектный код. Однако если ваша программа работает не так, как вы ожидаете, компиляция с флажком `-g` позволит вам воспользоваться отладчиком (например, `gdb`) для пошагового выполнения программы, что, в свою очередь, даст возможность разобраться в происходящем.

Ещё один полезный ключ — `-e`; он предписывает NASM'у прогнать наш исходный код через макропроцессор, выдать результат в поток стандартного вывода (попросту говоря, на экран) и на этом успокоиться. Такой режим работы может оказаться полезен, если мы ошиблись при написании макроса и не можем найти свою ошибку; увидев результат макропроцессирования нашей программы, мы, скорее всего, поймём, что и почему пошло не так.

NASM поддерживает и другие ключи командной строки; желающие могут изучить их самостоятельно, обратившись к документации.

3.4.2. Основы синтаксиса

Основной синтаксической единицей практически любого языка ассемблера (и NASM тут не исключение) является *строка текста*. Этим языки ассемблера отличаются от большинства (хотя и далеко не всех) языков высокого уровня, в которых символ перевода строки приравнивается к обычному пробелу.

Если нам не хватило длины строки, чтобы уместить всё, что мы хотели в ней уместить, то можно воспользоваться средством «склеивания» строк. Поставив последним символом строки «обратный слэш» (символ «\»), мы прикажем ассемблеру считать следующую строку продолжением предыдущей. Отметим, что это гораздо лучше, чем допускать в тексте программы очень длинные строки; обычно строка программы (любой, не только на языке ассемблера) не должна превышать 75 символов, в самом крайнем случае — 79, хотя компиляторы этого от нас и не требуют.

Строка текста[30] на языке ассемблера NASM состоит (в общем случае) из четырёх полей: метки, имени команды, операндов и комментария, причём метка, имя команды и комментарий являются полями необязательными. Что касается операндов, то требования к ним налагаются

[30] Здесь и далее под «строкой» понимается в том числе и «логическая» строка, склеенная из нескольких строк с помощью обратных слэшей; в дальнейшем мы не будем уточнять, что имеем в виду именно такие строки.

командой; если имя команды отсутствует, то отсутствуют и операнды. Могут отсутствовать и все четыре поля, тогда строка оказывается пустой. Ассемблер пустые строки игнорирует, но мы можем использовать их, чтобы визуально разделять между собой части программы.

В качестве метки можно использовать слово, состоящее из латинских букв, цифр, а также символов `'_'`, `'$'`, `'#'`, `'@'`, `'~'`, `'.'` и `'?'`, а начинаться метка может только с буквы или символов `'_'`, `'?'` и `'.'`. Как мы помним из §3.3.8, метки, начинающиеся с точки, считаются *локальными*. Кроме того, в некоторых случаях имя метки можно предварить символом `'$'`; обычно это используется, если нужно создать метку, имя которой совпадает с именем регистра, команды или директивы[31]. Надо отметить, что ассемблер различает регистр букв в именах меток, то есть, например, `'label'`, `'LABEL'`, `'Label'` и `'LaBeL'` — это четыре разные метки. После метки, если она в строке присутствует, можно поставить символ двоеточия, но не обязательно. Как уже отмечалось, обычно программисты ставят двоеточия после меток, на которые можно передавать управление, и не ставят двоеточия после меток, обозначающих области памяти. Хотя ассемблер и не требует поступать именно так, программа при использовании этого соглашения становится яснее.

В поле имени команды, если оно присутствует, может быть обозначение машинной команды (возможно, с префиксом `rep`, см. стр. 77; существуют и другие префиксы), либо псевдокоманды — директивы специального вида (некоторые из них мы уже рассматривали и к этому вопросу ещё вернёмся), либо, наконец, имя макроса (с такими мы тоже встречались, к ним относится, например, использовавшийся в примерах `PRINT`; созданию макросов будет посвящён отдельный параграф). В отличие от меток, в именах машинных команд и псевдокоманд ассемблер регистры букв не различает, так что мы можем с равным успехом написать, например, `mov`, `MOV`, `Mov` и даже `mOv`, хотя так писать, конечно же, не стоит. В именах макросов, как и в именах меток, регистр различается.

Требования к содержимому поля операндов зависят от того, какая конкретно команда, псевдокоманда или макрос указаны в поле команды. Если операндов больше одного, то они разделяются запятой. В поле операндов часто приходится использовать названия регистров, и в этих названиях регистр букв не различается, как и в именах машинных команд.

Читателю, запутавшемуся в том, где же регистр важен, а где нет, стоит запомнить одно простое правило: **ассемблер `nasm` не различает заглавные и строчные буквы во всех словах, которые он**

[31] Такое может понадобиться только в случае, если ваша программа состоит из модулей, написанных на разных языках программирования; тогда в других модулях вполне могут встретиться метки, совпадающие по имени с ключевыми словами ассемблера, и может потребоваться возможность на них ссылаться.

ввёл сам: в именах команд, названиях регистров, директивах, псевдокомандах, обозначениях длины операндов и типа переходов (слова `byte`, `dword`, `near` и т. п.), но он считает заглавные и строчные разными буквами в тех именах, которые вводит пользователь (программист, пишущий на языке ассемблера) — в метках и именах макросов.

Отметим ещё одно свойство NASM, связанное с записью операндов. **Операнд типа «память» *всегда* записывается с использованием квадратных скобок**. Для некоторых других ассемблеров это не так, что порождает постоянную путаницу.

Комментарий обозначается символом «точка с запятой» («`;`»). Начиная с этого символа весь текст до конца строки ассемблер не принимает во внимание, что позволяет написать там всё что угодно. Обычно это используют для вставки в текст программы пояснений, предназначенных для тех, кому придётся этот текст читать.

3.4.3. Псевдокоманды

Под ***псевдокомандами*** понимается ряд вводимых ассемблером NASM слов, которые могут использоваться синтаксически так же, как и мнемоники машинных команд, хотя машинными командами на самом деле не являются. Некоторые такие псевдокоманды, а именно `db`, `dw`, `dd`, `resb`, `resw` и `resd` нам уже известны из §3.2.3. Отметим только, что кроме перечисленных, NASM поддерживает также псевдокоманды `resq`, `rest`, `dq` и `dt`. Буква `q` в их названиях означает «quadro» — «учетверённое слово» (8 байт), буква `t` — от слова «ten» и означает десятибайтные элементы. Эти псевдокоманды обычно используются в программах, обрабатывающих числа с плавающей точкой (попросту говоря, дробные числа); более того `dt` в качестве инициализаторов допускает исключительно числа с плавающей точкой (например, `71.361775`). Кроме псевдокоманды `dt`, числа с плавающей точкой можно применять также с `dd` и `dq`; это обусловлено тем, что стандарт IEEE-754[32] предусматривает три формата чисел с плавающей точкой — обычные, двойной точности и повышенной точности, занимающие, соответственно, 4 байта, 8 байт и 10 байт.

Отдельного разговора заслуживает псевдокоманда `equ`, предназначенная для *определения констант*. Эта псевдокоманда всегда применяется в сочетании с меткой, то есть не поставить перед ней метку считается ошибкой. Псевдокоманда `equ` связывает стоящую перед ней метку с *явно заданным числом*. Самый простой пример:

[32]IEEE-754 — это международный стандарт, описывающий способ представления в машинной памяти чисел с плавающей точкой и регламентирующий операции над ними; стандарт был создан под эгидой американской организации *Institute of Electrical and Electronics Engineers* (IEEE), представляющей собой профессиональную ассоциацию инженеров в области электротехники и электроники.

```
four    equ 4
```

Мы определили метку `four`, *задающую число 4*. Теперь, например,

```
        mov eax, four
```

есть то же самое, что и

```
        mov eax, 4
```

Уместно напомнить, что *любая метка представляет собой не более чем число*, но когда меткой снабжается строка программы, содержащая мнемонику машинной команды или директиву выделения памяти, с такой меткой связывается соответствующий *адрес в памяти* (который есть тоже не что иное, как просто число), тогда как директива `equ` позволяет указать число явно.

Директивы `equ` часто применяется, чтобы связать с некоторым именем (меткой) длину массива, только что заданного с помощью директивы `db`, `dw` или любой другой. Для этого используется ***псевдометка*** `$`, которая в каждой строчке, где она появляется, обозначает *текущий адрес*[33]. Например, можно написать так:

```
msg     db "Hello and welcome", 10, 0
msglen  equ $-msg
```

Выражение `$-msg`, представляющее собой разность двух чисел, известных ассемблеру во время его работы, будет вычислено прямо во время ассемблирования. Поскольку `$` означает адрес, ставший текущим уже *после* описания строки, а `msg` — адрес *начала* строки, то их разность в точности равна длине строки (в нашем примере — 19). К вычислению выражений во время ассемблирования мы вернёмся в §3.4.5.

Директива `times` позволяет повторить какую-нибудь команду (или псевдокоманду) заданное количество раз. Например,

```
stars  times 4096 db '*'
```

задаёт область памяти размером в 4096 байт, заполненную кодом символа `'*'`, точно так же, как это сделали бы 4096 одинаковых строк, содержащих директиву `db '*'`.

Иногда может оказаться полезной псевдокоманда `incbin`, позволяющая создать область памяти, заполненную данными из некоторого внешнего файла. Подробно мы её рассматривать не будем; заинтересованный читатель может изучить эту директиву самостоятельно, обратившись к документации.

[33] Точнее говоря, текущее смещение относительно начала секции.

3.4.4. Константы

Константы в языке ассемблера NASM делятся на четыре категории: целые числа, символьные константы, строковые константы и числа с плавающей точкой.

Как уже говорилось (см. стр. 45), ***целочисленные константы*** можно задавать в десятичной, двоичной, шестнадцатеричной и восьмеричной системах счисления. Если просто написать число, состоящее из цифр (и, возможно, знака «минус» в качестве первого символа), то это число будет воспринято ассемблером как десятичное. Шестнадцатеричное число можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$`, как в Паскале (например, `$2af3`), либо поставив перед числом символы `0x`, как в языке Си (`0x2af3`). При использовании символа `$` нужно следить, чтобы сразу после `$` стояла цифра, а не буква, так что если число начинается с буквы, следует добавить `0` (например, `$0f9` вместо просто `$f9`). Это требуется, чтобы ассемблер не путал запись числа с записью пользовательской метки, перед которыми, как мы уже говорили, иногда тоже ставится знак `$`. Восьмеричное число обозначается добавлением после числа буквы `o` или `q` (например, `634o`, `754q`). Наконец, двоичное число обозначается буквой `b` (`10011011b`).

Символьные константы и ***строковые константы*** очень похожи друг на друга; более того, в любом месте, где по смыслу должна быть строковая константа, можно употребить и символьную. Разница между строковыми и символьными константами заключается только в их длине: под *символьной* константой подразумевается такая константа, которая укладывается в длину «двойного слова» (то есть содержит не более 4 символов) и может в силу этого рассматриваться как альтернативная запись целого числа (либо битовой строки). И символьные, и строковые константы могут записываться как с помощью двойных кавычек, так и с помощью апострофов. Это позволяет использовать в строках и сами символы апострофов и кавычек: если строка содержит символ кавычек одного типа, то её заключают в кавычки другого типа (см. пример на стр. 46).

Символьные константы, содержащие меньше 4 символов, считаются синонимами целых чисел, младшие байты которых равны кодам символов из константы, а недостающие старшие байты заполнены нулями. При использовании символьных констант следует помнить, что целые числа в компьютерах с процессорами i386 записываются в обратном порядке байтов, то есть младший байт идёт первым. В то же время по смыслу строки (и символьной константы) код первой буквы должен в памяти размещаться первым. Поэтому, например, константа `'abcd'` эквивалентна числу `64636261h`: `64h` — это код буквы `d`, `61h` — код буквы `a`, и в обоих случаях байт со значением `61h` стоит первым, а `64h` — последним. В некоторых случаях ассемблер воспринимает в качестве строковых и такие константы, которые достаточно коротки и могли бы считаться символьными.

Это происходит, например, если ассемблер видит символьную константу длиной более 1 символа в параметрах директивы `db` или константу длиной более двух символов в параметрах директивы `dw`.

Константы с плавающей точкой, задающие дробные числа, синтаксически отличаются от целочисленных констант наличием десятичной точки. Учтите, что **целочисленная константа `1` и константа `1.0` не имеют между собой ничего общего!** Для наглядности отметим, что битовая запись числа с плавающей точкой `1.0` одиночной точности (то есть запись, занимающая 4 байта, так же, как и для целого числа) эквивалентна записи целого числа `3f800000h` (1065353216 в десятичной записи). Константу с плавающей точкой можно задать и в экспоненциальном виде, используя букву `e` или `E`. Например, `1.0e-5` есть то же самое, что и `0.00001`. Обратите внимание, что десятичная точка по-прежнему обязательна.

3.4.5. Вычисление выражений во время ассемблирования

Ассемблер NASM в некоторых случаях вычисляет встретившиеся ему арифметические выражения непосредственно во время ассемблирования. Важно понимать, что **в итоговый машинный код попадают только вычисленные результаты, а не сами действия по их вычислению**. Естественно, для вычисления выражения во время ассемблирования требуется, чтобы такое выражение не содержало никаких неизвестных: всё, что нужно для вычисления, должно быть известно ассемблеру во время его работы.

Выражение, вычисляемое ассемблером, должно быть **целочисленным**, то есть состоять из целочисленных констант и меток, и использовать операции из следующего списка:

- `+` и `-` — сложение и вычитание;
- `*` — умножение;
- `/` и `%` — целочисленное деление и остаток от деления (для беззнаковых целых чисел);
- `//` и `%%` — целочисленное деление и остаток от деления (для знаковых целых чисел);
- `&`, `|`, `^` — операции побитового «и», «или», «исключающего или»;
- `<<` и `>>` — операции побитового сдвига влево и вправо;
- унарные операции `-` и `+` используются в их обычной роли: `-` меняет знак числа на противоположный, `+` не делает ничего;
- унарная операция `~` обозначает побитовое отрицание.

При применении операций `%` и `%%` нужно обязательно оставлять пробельный символ после знака операции, чтобы ассемблер не перепутал их с *макродирективами* (макродирективы мы уже использовали в примерах, а подробнее рассмотрим позже).

Ещё одна унарная операция, `seg`, для нас неприменима ввиду отсутствия сегментов в «плоской» модели памяти.

Унарные операции имеют самый высокий приоритет, следом за ними идут операции умножения, деления и остатка от деления, ещё ниже приоритет у операций сложения и вычитания. Далее (в порядке убывания приоритета) идут операции сдвигов, операция `&`, затем операция `^`, и замыкает список операция `|`, имеющая самый низкий приоритет. Порядок выполнения операций можно изменить, применив круглые скобки.

3.4.6. Критические выражения

Ассемблер анализирует исходный текст в два прохода. На первом проходе вычисляется размер всех машинных команд и других данных, подлежащих размещению в памяти программы; в результате ассемблер устанавливает, какое *числовое значение* должно быть приписано каждой из встретившихся в тексте программы меток. На втором проходе генерируется собственно машинный код и прочее содержимое памяти. Второй проход нужен, чтобы, например, можно было ссылаться на метку, стоящую в тексте *позже*, чем ссылка на неё: когда ассемблер видит метку, скажем, в команде `jmp`, раньше, чем встретится собственно команда, помеченная этой меткой, на первом проходе он не может сгенерировать код, поскольку не знает численного значения метки. На втором проходе все значения уже известны, и никаких проблем с генерированием кода не возникает.

Всё это имеет прямое отношение к механизму вычисления выражений. Ясно, что выражение, содержащее метку, ассемблер может вычислить на первом проходе только если метка стояла в тексте раньше, чем вычисляемое выражение; в противном случае вычисление выражения приходится отложить до второго прохода. Ничего страшного в этом нет, *если только значение выражения не влияет на размер команды, выделяемой области памяти и т. п.*, то есть от значения этого выражения не зависят численные значения, которые нужно будет приписать дальнейшим встреченным меткам. Если же это условие не выполнено, то невозможность вычислить выражение на первом проходе приведёт к невозможности выполнить задачу первого прохода — определить численные значения всех меток. Более того, в некоторых случаях не помогло бы никакое количество проходов, даже если бы ассемблер это умел. В документации к ассемблеру NASM приведён такой пример:

```
        times (label-$) db 0
label:  db      'Where am I?'
```

Здесь строчка с директивой `times` должна создать столько нулевых байтов, на сколько ячеек метка `label` отстоит от самой этой строчки —

но ведь метка `label` как раз и отстоит от этой строчки на столько ячеек, сколько нулевых байтов будет создано. Так сколько же их должно быть?!

В связи с этим мы вводим понятие ***критического выражения***: это такое выражение, вычисляемое во время ассемблирования, которое ассемблеру необходимо вычислить во время первого прохода. Критическими ассемблер считает любые выражения, от которых тем или иным образом зависит размер чего бы то ни было, располагаемого в памяти (и которые, следовательно, могут повлиять на значения меток, вводимых позже). В критических выражениях можно использовать только числовые константы, а также метки, определённые *выше по тексту программы*, чем рассматриваемое выражение. Это гарантирует возможность вычисления выражения на первом проходе.

Кроме аргумента директивы `times` к категории критических относятся, например, выражения в аргументах псевдокоманд `resb`, `resw` и др., а также в некоторых случаях — выражения в составе исполнительных адресов, которые могут повлиять на итоговый размер ассемблируемой команды. Так, команды «`mov eax,[ebx]`», «`mov eax,[ebx+10]`» и «`mov eax,[ebx+10000]`» порождают соответственно 2 байта, 3 байта и 6 байтов кода, поскольку исполнительный адрес в первом случае занимает всего 1 байт, во втором из-за входящего в него однобайтового числа — 2 байта, а в последнем — 5, из которых четыре расходуются на представление числа `1000`; но сколько памяти займёт команда

```
mov eax, [ebx+label]
```

если значение `label` пока не определено? Впрочем, этих трудностей можно избежать, если внутри исполнительного адреса в явном виде указать разрядность словом `byte`, `word` или `dword`. Так, если написать

```
mov eax, [ebx + dword label]
```

то, даже если значение `label` ещё не известно, длина его (и, как следствие, длина всей машинной команды) уже указана.

3.4.7. Выражения в составе исполнительного адреса

На рис. 3.2 (см. стр. 52) мы приводили общий вид исполнительного адреса (операндов типа «память») с точки зрения машинных команд. Ассемблер NASM может воспринимать и более сложные выражения в квадратных скобках, лишь бы их было возможно привести к указанному виду. Так, например, в команде

```
mov eax, [5*ebx]
```

используется умножение на число 5, что вроде бы запрещено (умножать можно только на 1, 2, 4 и 8), но ассемблер справляется с этой

сложностью, приведя в команде операнд к виду `[ebx+4*ebx]`, который уже вполне корректен. Если же рассмотреть команду

```
mov eax, [ebx+4*ecx+5*x+y]
```

в которой `x` и `y` — некоторые метки, то и с этим ассемблер справится, попросту *вычислив* выражение `5*x+y` и получив в итоге одно число, что уже вполне соответствует общему виду исполнительного адреса.

Следует помнить, что, если в явном виде не указать нужную разрядность, такие выражения будут считаться *критическими*, то есть должны зависеть только от меток, уже введённых к моменту рассмотрения выражения (см. предыдущий параграф).

3.5. Макросредства и макропроцессор

3.5.1. Основные понятия

Под ***макропроцессором*** понимают программное средство, которое получает на вход некоторый текст и, пользуясь указаниями, данными в самом тексте, частично преобразует его, давая на выходе, в свою очередь, текст, но уже не имеющий указаний к преобразованию. В применении к языкам программирования макропроцессор — это преобразователь исходного текста программы, обычно совмещённый с компилятором; результатом работы макропроцессора является **текст на языке программирования**, который потом уже обрабатывается компилятором в соответствии с правилами языка (см. рис. 3.6).

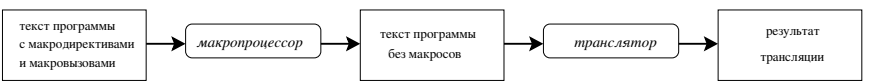

Рис. 3.6. Схема работы макропроцессора

Поскольку языки ассемблера обычно весьма бедны по своим изобразительным возможностям (если сравнивать с языками высокого уровня), то, чтобы хоть как-то компенсировать программистам неудобства, обычно ассемблеры снабжают очень мощными макропроцессорами. В частности, рассматриваемый нами ассемблер NASM содержит в себе алгоритмически полный макропроцессор, который мы можем при желании заставить написать за нас едва ли не всю программу.

С макросами мы уже встречались: часто использовавшиеся нами `PRINT` и `FINISH` представляют собой именно макросы, или, точнее, ***имена макросов***.

Вообще ***макросом*** называют некоторое правило, в соответствии с которым фрагмент программы, содержащий определённое слово, должен быть преобразован. Само это слово называют ***именем макроса***;

часто вместо термина «имя макроса» используют просто слово «макрос», хотя это и не совсем верно.

Прежде чем мы сможем воспользоваться макросом, его необходимо определить, то есть, во-первых, указать макропроцессору, что некий идентификатор отныне считается именем макроса (так что его появление в тексте программы требует вмешательства макропроцессора), и, во-вторых, задать то правило, по которому макропроцессор должен действовать, встретив это имя. Фрагмент программы, определяющий макрос, называют ***макроопределением***. Когда макропроцессор встречает в тексте программы имя макроса и параметры (так называемый ***вызов макроса***, или ***макровызов***), он *заменяет* имя макроса (и, возможно, параметры, относящиеся к нему) фрагментом текста, полученным в соответствии с определением макроса. Такая замена называется ***макроподстановкой***, а текст, полученный в результате — ***макрорасширением***[34].

Бывает и так, что макропроцессор производит преобразование текста программы, не видя ни одного имени макроса, но повинуясь ещё более прямым указаниям, выраженным в виде ***макродирективов***. Одну такую макродирективу мы уже знаем: это директива `%include`, которая приказывает макропроцессору заменить её саму на содержимое файла, указанного параметром директивы. Так, привычная нам строка

```
%include "stud_io.inc"
```

заменяется на всё, что есть в файле `stud_io.inc`.

3.5.2. Простейшие примеры макросов

Чтобы составить представление о том, как можно воспользоваться макропроцессором и для чего он нужен, приведём два простых примера. Как мы видели из §§3.3.6, 3.3.7 и 3.3.9, запись вызова подпрограммы на языке ассемблера занимает несколько строк ($2 + n$, где n — число параметров подпрограммы). Это не всегда удобно, особенно для людей, привыкших к языкам высокого уровня. Пользуясь механизмом макросов, мы можем изрядно сократить запись вызова подпрограммы. Для этого мы опишем макросы `pcall1`, `pcall2` и т. д., для вызова, соответственно, процедуры с одним параметром, с двумя параметрами и т. д. С помощью таких макросов запись вызова процедуры сократится до одной строчки; вместо

```
push edx
push dword mylabel
push dword 517
```

[34] Термин «макрорасширение» — это не слишком удачная калька английского термина «macro expansion».

```
        call myproc
        add esp, 12
```

можно будет написать

```
        pcall3 myproc, dword 517, dword mylabel, edx
```

что, конечно, гораздо удобнее и понятнее. Позже, разобравшись с макроопределениями глубже, мы перепишем эти макросы, вместо них введя один макрос `pcall`, работающий для любого количества аргументов, но пока для примера ограничимся частными случаями. Итак, пишем макроопределение:

```
%macro pcall1 2      ; 2 -- кол-во параметров макроса
        push %2
        call %1
        add esp, 4
%endmacro
```

Мы описали ***многострочный макрос*** с именем `pcall1`, имеющий два параметра: имя вызываемой процедуры для команды `call` и аргумент процедуры для занесения в стек. Строки, написанные между директивами `%macro` и `%endmacro`, составляют ***тело макроса*** — шаблон для текста, который должен получиться в результате макроподстановки. Сама макроподстановка в данном случае будет довольно простой: макропроцессор только заменит вхождения `%1` и `%2` соответственно на первый и второй параметры, заданные в макровызове. Если после такого определения в тексте нашей программы встретится строка вида

```
        pcall1 proc, eax
```

макропроцессор воспримет эту строку как макровызов и выполнит макроподстановку в соответствии с вышеприведённым макроопределением, считая первым параметром слово `proc`, вторым параметром слово `eax` и подставляя их вместо `%1` и `%2`. В результате получится следующий фрагмент:

```
        push eax
        call proc
        add esp, 4
```

Аналогичным образом опишем макросы `pcall2` и `pcall3`:

```
%macro pcall2 3
        push %3
        push %2
        call %1
```

```
        add esp, 8
%endmacro
%macro pcall3 4
        push %4
        push %3
        push %2
        call %1
        add esp, 12
%endmacro
```

Для полноты можно дописать также и макрос `pcall0`:

```
%macro pcall0 1
        call %1
%endmacro
```

Конечно, такой макрос, в отличие от предыдущих, ничуть не сокращает объём программы, но зато он позволит нам все вызовы подпрограмм оформить единообразно. Описание макросов `pcall4`, `pcall5` и т. д. до `pcall8` оставляем читателю в качестве упражнения; заодно для самопроверки ответьте на вопрос, почему мы предлагаем остановиться именно на `pcall8`, а не, например, на `pcall9` или `pcall12`.

Рассмотренный нами пример использовал *многострочный макрос*; как мы убедились, вызов многострочного макроса синтаксически выглядит точно так же, как использование машинных команд или псевдокоманд: вместо имени команды пишется имя макроса, затем через запятую перечисляются параметры. При этом многострочный макрос всегда преобразуется в одну или несколько *строк* на языке ассемблера. Но что если, к примеру, нам нужно сгенерировать с помощью макроса некоторую *часть строки*, а не фрагмент из нескольких строк? Такая потребность тоже возникает довольно регулярно. Так, в примере, приведённом в §3.3.9, видно, что внутри процедур очень часто приходится использовать конструкции вроде `[ebp+12]`, `[ebp-4]` и т. п. для обращения к параметрам процедуры и её локальным переменным. В принципе, к этим конструкциям несложно привыкнуть; но можно пойти и другим путём, применив ***однострочные макросы***. Для начала напишем следующие[35] макроопределения:

```
%define arg1 ebp+8
%define arg2 ebp+12
%define arg3 ebp+16
%define local1 ebp-4
%define local2 ebp-8
%define local3 ebp-12
```

[35] Здесь и далее в наших примерах мы предполагаем, что все параметры процедур и все локальные переменные всегда представляют собой «двойные слова», то есть имеют размер 4 байта; на самом деле, конечно, это не всегда так, но нам сейчас важнее иллюстративная ценность примера.

В дополнение к ним допишем ещё и такое:

```
%define arg(n) ebp+(4*n)+4
%define local(n) ebp-(4*n)
```

Теперь к параметру процедуры можно обратиться так:

```
mov eax, [arg1]
```

или так (если, например, не хватило описанных макросов)

```
mov [arg(7)], edx
```

В принципе мы могли и квадратные скобки включить внутрь макросов, чтобы не писать их каждый раз. Например, если изменить определение макроса `arg1` на следующее:

```
%define arg1 [ebp+8]
```

то соответствующий макровызов стал бы выглядеть так:

```
mov eax, arg1
```

Мы не сделали этого из соображений сохранения наглядности. Ассемблер NASM поддерживает, как мы уже знаем, соглашение о том, что любое обращение к памяти оформляется с помощью квадратных скобок, если же их нет, то мы имеем дело с непосредственным или регистровым операндом. Программист, привыкший к этому соглашению, при чтении программы будет вынужден прилагать лишние усилия, чтобы вспомнить, что `arg1` в данном случае не метка, а имя макроса, так что здесь происходит именно обращение к памяти, а не загрузка в регистр адреса метки. Понятности программы такие вещи отнюдь не способствуют. Учтите, что и вы сами, будучи даже автором программы, можете за несколько дней начисто забыть, что́ же имелось в виду, и тогда экономия двух символов (скобок) обернётся для вас потерей бесценного времени.

3.5.3. Однострочные макросы; макропеременные

Как видно из примеров предыдущего параграфа, однострочный макрос — это такой макрос, определение которого состоит из одной строки, а его вызов разворачивается во фрагмент строки текста (то есть может использоваться для генерации *части* строки). Отметим, что единожды определённый макрос можно при необходимости *переопределить*, просто вставив в текст программы ещё одно определение того же макроса. С того момента, как макропроцессор «увидит» новое определение, он будет использовать его вместо старого. Таким образом, одно и то же имя макроса в разных местах программы может означать разные вещи и раскрываться в разные фрагменты текста. Более того, макрос вообще можно убрать, воспользовавшись директивой `%undef`; встретив такую

директиву, макропроцессор немедленно «забудет» о существовании макроса. Представляет интерес вопрос о том, что будет, если в определении одного макроса использовать вызов другого макроса, а этот последний время от времени переопределять.

Если для описания однострочного макроса `A` использовать уже знакомую нам директиву `%define` и в её теле использовать макровызов макроса `B`, то этот макровызов в самой директиве не раскрывается; макропроцессор оставляет вхождение макроса `B` как оно есть до тех пор, пока не встретит вызов макроса `A`. Когда же будет выполнена макроподстановка для `A`, в её результате будет содержаться `B`, и для него макропроцессор, в свою очередь, выполнит макроподстановку. Очевидно, что при этом будет использовано то определение макроса `B`, которое было актуальным в момент *подстановки* (а не определения) `A`.

Поясним сказанное на примере. Пусть мы ввели два макроса:

```
%define        thenumber      25
%define        mkvar          dd thenumber
```

Если теперь написать в программе строчку

```
var1           mkvar
```

то макропроцессор сначала выполнит макроподстановку для `mkvar`, получив строку

```
var1           dd thenumber
```

а из неё, в свою очередь, макроподстановкой `thenumber` получит строку

```
var1           dd 25
```

Если теперь переопределить `thenumber` и снова вызвать `mkvar`:

```
%define        thenumber      36
var2           mkvar
```

то результатом работы макропроцессора будет строка, содержащая именно число `36`:

```
var2           dd 36
```

несмотря на то, что сам макрос `mkvar` мы не изменяли: на первом шаге будет получено, как и в прошлый раз, `dd thenumber`, но у `thenumber` теперь значение `36`, оно и будет подставлено. Такая стратегия макроподстановок называется «ленивой[36]». Однако ассембер NASM позволяет

[36] Такое название является калькой английского *lazy* и частично оправдано тем, что макропроцессор как бы «ленится» выполнять макроподстановку (в данном случае макроса `thenumber`), пока его к этому не вынудят.

применять и другую стратегию, называемую *энергичной*, для чего предусмотрена директива `%xdefine`. Эта директива полностью аналогична директиве `%define` с той только разницей, что, если в теле описания макроса встречаются макровызовы, макропроцессор производит их макроподстановки незамедлительно, то есть прямо в момент обработки макроопределения, не дожидаясь, пока пользователь вызовет описываемый макрос. Так, если в вышеприведённом примере заменить директиву `%define` в описании макроса `mkvar` на `%xdefine`:

```
%define        thenumber      25
%xdefine       mkvar          dd thenumber
var1           mkvar
%define        thenumber      36
var2           mkvar
```

то обе получившиеся строки будут содержать число 25:

```
var1           dd 25
var2           dd 25
```

Переопределение макроса `thenumber` теперь не в силах повлиять на работу макроса `mkvar`, поскольку тело макроса `mkvar` на этот раз не содержит слова `thenumber`: обрабатывая определение `mkvar`, макропроцессор подставил вместо слова `thenumber` его значение (25).

Иногда бывает нужно связать с именем макроса не просто строку, а число, являющееся результатом вычисления арифметического выражения. Ассемблер NASM позволяет это сделать, используя директиву `%assign`. В отличие от `%define` и `%xdefine`, эта директива не только выполняет все необходимые подстановки в теле макроопределения, но и пытается *вычислить* тело как обыкновенное целочисленное арифметическое выражение. Если это не получается, фиксируется ошибка. Так, если написать в программе сначала

```
%assign        var     25
```

а потом

```
%assign        var     var+1
```

то в результате с макроименем `var` будет связано значение 26, которое и будет подставлено, если макропроцессор встретит слово `var` в дальнейшем тексте программы.

Макроимена, вводимые директивой `%assign`, обычно называют ***макропеременными***. Как мы увидим далее, макропеременные являются важным средством, позволяющим задать макропроцессору целую программу, результатом которой может стать очень длинный текст на языке ассемблера.

3.5.4. Условная компиляция

Часто при разработке программ возникает потребность в создании различных версий исполняемого файла с использованием одного и того же исходного текста. Допустим, мы пишем программы на заказ и у нас есть два заказчика Петров и Сидоров; программы для них почти одинаковы, но у каждого из двоих имеются специфические требования, отсутствующие у другого. В такой ситуации хотелось бы, конечно, иметь и поддерживать один исходный текст: в противном случе у нас появятся две копии одного кода, и придётся, например, каждую найденную ошибку исправлять в двух местах. Однако при компиляции версии для Петрова нужно исключить из работы фрагменты, предназначенные для Сидорова, и наоборот.

Бывают и другие подобные ситуации; одну из них, *отладочную печать*, мы уже встречали при изучении Паскаля (см. т. 1, §2.16.3). Окончательная версия программы не должна содержать операций отладочной печати, поскольку вся отладочная информация предназначена для программиста, автора программы, а пользователю она только мешает. Проблема в том, что отладка программы — процесс бесконечный, и как только мы решим, что она завершена и удалим из текста всю отладочную печать, по закону подлости тут же обнаружится очередная ошибка, и нам вновь придётся редактировать исходник, чтобы вернуть отладочную печать на место.

Большинство профессиональных компилируемых языков программирования поддерживают для подобных случаев специальные конструкции, называемые ***директивами условной компиляции*** и позволяющие выбирать, какие фрагменты программы компилировать, а какие игнорировать. Обычно отработку директив условной компиляции возлагают на макропроцессор, если, конечно, он есть. Сказанное справедливо, кроме прочего, практически для всех языков ассемблера, включая и наш NASM.

Рассмотрим пример, связанный с отладкой. Допустим, мы написали программу, откомпилировали её и запустили, но она завершается аварийно, и мы не можем понять причину, но думаем, что авария происходит в некоем «подозрительном» фрагменте. Чтобы проверить своё предположение, мы хотим непосредственно перед входом в этот фрагмент и сразу после выхода из него вставить печать соответствующих сообщений. Чтобы нам не пришлось по несколько раз стирать эти сообщения и вставлять их снова, воспользуемся директивами условной компиляции. Выглядеть это будет примерно так:

```
%ifdef DEBUG_PRINT
        PRINT "Entering suspicious section"
        PUTCHAR 10
%endif
```

```
;
;     здесь идёт "подозрительная" часть программы
;
%ifdef DEBUG_PRINT
        PRINT "Leaving suspicious section"
        PUTCHAR 10
%endif
```

Здесь `%ifdef` — это одна из ***директив условной компиляции***, означающая «компилировать только в случае, если определён данный однострочный макрос» (в данном случае это макрос `DEBUG_PRINT`). Теперь в начало программы следует вставить строку, определяющую этот символ:

```
%define DEBUG_PRINT
```

Тогда при запуске NASM «увидит» и откомпилирует фрагменты нашего исходного текста, заключённые между соответствующими `%ifdef` и `%endif`; когда же мы найдём ошибку и отладочная печать будет нам больше не нужна, достаточно будет убрать этот `%define` из начала программы или даже поставить перед ним знак комментария:

```
;%define DEBUG_PRINT
```

и фрагменты, обрамлённые соответствующими директивами, макропроцессор будет попросту игнорировать, так что их можно оставить в тексте программы на случай, если они снова понадобятся.

Отметим, что для включения и отключения отладочной печати, оформленной таким образом, можно вообще обойтись без правки исходного текста. Как мы видели в §3.4.1, определить макросимвол можно ключом командной строки NASM; в частности, включить отладочную печать из нашего примера можно, вызвав NASM примерно так:

```
avst@host:~/work$ nasm -f elf -dDEBUG_PRINT prog.asm
```

Это избавляет нас от необходимости вставлять в исходный текст директиву `%define`, а потом её удалять.

Возвращаясь к ситуации с двумя заказчиками, мы можем предусмотреть в программе конструкции, подобные следующей:

```
%ifdef FOR_PETROV
;
; здесь код, предназначенный только для Петрова
;
%elifdef FOR_SIDOROV
;
; а здесь - только для Сидорова
;
```

```
%else
; если ни тот, ни другой символ не определён,
; прервём компиляцию и выдадим сообщение об ошибке
%error Please define either FOR_PETROV or FOR_SIDOROV
%endif
```

(директива `%elifdef` — это сокращённая форма записи для `else` и `ifdef`). При компиляции такой программы нужно будет обязательно указать ключ `-dFOR_PETROV` или `-dFOR_SIDOROV`, иначе NASM начнёт обрабатывать фрагмент, находящийся после `%else`, и, встретив директиву `%error`, выдаст сообщение об ошибке.

Кроме проверки *наличия* макросимвола, можно проверять также и факт *отсутствия* макросимвола (то есть прямо противоположное условие). Это делается директивой `%ifndef` (*if not defined*). Как и для `%ifdef`, для `%ifndef` существует сокращённая запись конструкции с `%else`, она называется `%elifndef`.

Для задания условия, при котором тот или иной фрагмент подлежит или не подлежит компиляции, можно пользоваться не только фактом наличия или отсутствия макроса; NASM поддерживает и другие директивы условной компиляции. Наиболее общей является директива `%if`, в которой условие задаётся арифметико-логическим выражением, вычисляемым во время компиляции. С такими выражениями мы уже встречались в §3.4.5; для формирования логических выражений набор допустимых операций расширяется операциями `=`, `<`, `>`, `>=`, `<=`, в их обычном смысле, операцию «не равно» можно задать символом `<>`, как в Паскале, или символом `!=`, как в Си; поддерживается и Си-подобная форма записи операции «равно» в виде двух знаков равенства `==`. Кроме того, доступны логические связки `&&` («и»), `||` («или») и `^^` («исключающее или»). Отметим, что все выражения, используемые в директиве `%if`, рассматриваются как *критические* (см. §3.4.6). Так же, как и для всех остальных `%if`-директив, для простого `%if` имеется форма сокращённой записи конструкции с `%else` — директива `%elif`.

Перечислим кратко остальные поддерживаемые NASM условные директивы. Директивы `%ifidn` и `%ifidni` принимают два аргумента, разделённые запятой, и сравнивают их как строки, предварительно произведя, если нужно, макроподстановки в тексте аргументов. Фрагмент кода, следующий за этими директивами, транслируется только в случае, если строки окажутся равными, причём `%ifidn` требует точного совпадения, тогда как `%ifidni` игнорирует регистр и считает, например, строки `foobar`, `FooBar` и `FOOBAR` одинаковыми. Для проверки противоположного условия можно использовать директивы `%ifnidn` и `%ifnidni`; все четыре директивы имеют `%elif`-формы, соответственно, `%elifidn`, `%elifidni`, `%elifnidn` и `%elifnidni`. Директива `%ifmacro` проверяет существование многострочного макроса; поддерживаются директивы `%ifnmacro`, `%elifmacro` и `%elifnmacro`. Директивы `%ifid`,

`%ifstr` и `%ifnum` проверяют, является ли их аргумент соответственно идентификатором, строкой или числовой константой. Как обычно, NASM поддерживает все дополнительные формы вида `%ifnXXX`, `%elifXXX` и `%elifnXXX` для всех трёх директив.

Кроме перечисленных, NASM поддерживает директиву `%ifctx` и соответствующие формы, но объяснение её работы достаточно сложно и обсуждать эту директиву мы не будем.

3.5.5. Макроповторения

Макропроцессор ассемблера NASM позволяет многократно (циклически) обрабатывать один и тот же фрагмент кода. Это достигается директивами `%rep` (от слова *repetition*) и `%endrep`. Директива `%rep` принимает один обязательный параметр, означающий количество повторений. Фрагмент кода, заключённый между директивами `%rep` и `%endrep`, будет обработан макропроцессором (и ассемблером) столько раз, сколько указано в параметре директивы `%rep`. Кроме того, между директивами `%rep` и `%endrep` может встретиться директива `%exitrep`, которая досрочно прекращает выполнение макроповторения. Рассмотрим простой пример. Пусть нам необходимо описать область памяти, состоящую из 100 последовательных байтов, причём в первом из них должно содержаться число `50`, во втором — число `51` и т. д., в последнем, соответственно, число `149`. Конечно, можно просто написать сто строк кода:

```
db 50
db 51
db 52
;....
db 148
db 149
```

но это, во-первых, утомительно, а во-вторых, занимает слишком много места в тексте программы. Гораздо правильнее будет поручить генерацию этого кода макропроцессору, воспользовавшись макроповторением и макропеременной:

```
%assign n 50
%rep 100
        db n
  %assign n n+1
%endrep
```

Встретив такой фрагмент, макропроцессор сначала свяжет с макропеременной `n` значение `50`, затем сто раз рассмотрит две строчки, заключённые между `%rep` и `%endrep`, причём каждое рассмотрение этих строк

приведёт к генерации очередной подлежащей ассемблированию строки `db 50`, `db 51`, `db 52` и т. д.; изменение числа происходит благодаря тому, что значение макропеременной `n` изменяется (увеличивается на единицу) на каждом проходе макроповторения. Иначе говоря, в результате обработки макропроцессором этого фрагмента как раз и получатся точно такие сто строк кода, как показано выше, и именно они и будут ассемблироваться.

Рассмотрим более сложный пример. Пусть имеется необходимость задать область памяти, содержащую последовательно в виде четырёхбайтных целых все числа Фибоначчи[37], не превосходящие 100 000. Сгенерировать соответствующую последовательность директив `dd` можно с помощью такого фрагмента кода:

```
fibonacci
%assign i 1
%assign j 1
%rep 100000
  %if j > 100000
     %exitrep
  %endif

        dd j

  %assign k j+i
  %assign i j
  %assign j k
%endrep
fib_count       equ ($-fibonacci)/4
```

Метка `fibonacci` будет связана с адресом начала сгенерированной области памяти, а метка `fib_count` — с общим количеством чисел, размещённых в этой области памяти (с этим приёмом мы уже сталкивались, см. стр. 104).

Использовать макроповторения можно не только для генерации областей памяти, заполненных числами, но и для других целей. Пусть, например, у нас есть массив из 128 двухбайтовых целых чисел:

```
array   resw 128
```

и мы хотим написать последовательность из 128 команд `inc`, увеличивающих на единицу каждый из элементов этого массива. Можно сделать это так:

```
%assign a 0
```

[37] Напомним, что числа Фибоначчи — это последовательность чисел, начинающаяся с двух единиц, каждое следующее число которой получается сложением двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 41, 54 и т. д.

```
%rep 128
        inc word [array + a]
%assign a a+2
%endrep
```

Читатель мог бы отметить, что использование в такой ситуации 128 команд нерационально и правильнее было бы воспользоваться циклом во время исполнения, например, так:

```
        mov ecx, 128
lp:     inc word [array + ecx*2 - 2]
        loop lp
```

В большинстве случаев такой вариант действительно предпочтительнее, поскольку эти три команды, естественно, будут занимать в несколько десятков раз меньше памяти, чем последовательность из 128 команд `inc`, но следует иметь в виду, что работать такой код будет примерно в полтора раза медленнее, так что в некоторых случаях применение макроцикла для генерации последовательности одинаковых команд (вместо цикла времени исполнения) может оказаться осмысленным.

3.5.6. Многострочные макросы и локальные метки

Вернёмся теперь к многострочным макросам; такие макросы генерируют не фрагмент строки, а фрагмент текста, состоящий из нескольких строк. Описание многострочного макроса также состоит из нескольких строк, заключённых между директивами `%macro` и `%endmacro`. В §3.5.2 мы уже рассматривали простейшие примеры многострочных макросов, однако в мало-мальски сложном случае рассмотренных средств нам не хватит. Пусть, например, мы хотим описать макрос `zeromem`, принимающий на вход два параметра — адрес и длину области памяти — и раскрывающийся в код, заполняющий эту память нулями. Не особенно задумываясь над происходящим, мы могли бы написать, например, следующий (**неправильный!**) код:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
        push esi
        mov ecx, %2
        mov esi, %1
lp:     mov byte [esi], 0
        inc esi
        loop lp
        pop esi
        pop ecx
%endmacro
```

NASM примет такое описание и даже позволит произвести один макровызов. Если же в нашей программе встретятся хотя бы два вызова макроса `zeromem`, то при попытке оттранслировать программу мы получим сообщение об ошибке — NASM пожалуется на то, что мы используем одну и ту же метку (`lp:`) дважды. Действительно, при каждом макровызове макропроцессор вставит вместо вызова всё тело нашего макроопределения, только заменив `%1` и `%2` на соответствующие параметры, а всё остальное сохранив без изменения. Значит, строка

```
lp:     mov byte [esi], 0
```

содержащая метку `lp`, встретится ассемблеру (уже после макропроцессирования) дважды — или, точнее, ровно столько раз, сколько раз будет вызван макрос `zeromem`.

Ясно, что нужен некий механизм, позволяющий локализовать метку, используемую внутри многострочного макроса, чтобы такие метки, полученные вызовом одного и того же макроса в разных местах программы, не конфликтовали друг с другом. В NASM такой механизм называется «локальные метки в макросах». Чтобы задействовать его, следует начать имя метки с двух символов `%` — так, в приведённом выше примере оба вхождения метки `lp` нужно заменить на `%%lp`. Такая метка будет в каждом следующем макровызове заменяться новым (не повторяющимся) идентификатором: при первом вызове макроса `zeromem` NASM заменит `%%lp` на `..@1.lp`, при втором — на `..@2.lp` и т. д.

Отметим ещё один недостаток вышеприведённого определения `zeromem`. Если при вызове этого макроса пользователь (программист, пользующийся нашим макросом, или, возможно, мы сами) укажет в качестве первого параметра (адреса начала области памяти) регистр `ECX` или в качестве второго (длины области памяти) — регистр `ESI`, макровызов будет успешно оттранслирован, но работать программа будет совсем не так, как от неё ожидается. Действительно, если написать что-то вроде

```
section .bss
array   resb 256
arr_len equ $-array

section .text
; ...
        mov ecx, array
        mov esi, arr_len
        zeromem ecx, esi
; ...
```

то начало макроса `zeromem` развернётся в следующий код:

```
        push ecx
        push esi
```

```
        mov ecx, esi
        mov esi, ecx
        ; ...
```

в результате чего, очевидно, в **обоих** регистрах ECX и ESI окажется длина массива, а адрес его начала будет потерян. Скорее всего, такая программа аварийно завершится, дойдя до этого фрагмента кода.

Чтобы избежать подобных проблем, можно воспользоваться директивами условной компиляции, проверяя, не является ли первый параметр регистром ECX, а второй — регистром ESI, но можно поступить и проще — загрузить значения параметров в регистры через временную запись их в стек, то есть вместо

```
        mov ecx, %2
        mov esi, %1
```

написать

```
        push dword %2
        push dword %1
        pop esi
        pop ecx
```

Окончательно наше макроопределение примет следующий вид:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
        push esi
        push dword %2
        push dword %1
        pop esi
        pop ecx
%%lp:   mov byte [esi], 0
        inc esi
        loop %%lp
        pop esi
        pop ecx
%endmacro
```

3.5.7. Макросы с переменным числом параметров

При описании многострочных макросов с помощью директивы %macro ассемблер NASM позволяет задать переменное число параметров. Это делается с помощью символа «-», который в данном случае символизирует тире. Например, директива

```
%macro mymacro 1-3
```

задаёт макрос, принимающий от одного до трёх параметров, а директива

```
%macro mysecondmacro 2-*
```

задаёт макрос, допускающий произвольное количество параметров, не меньшее двух. При работе с такими макросами может оказаться полезным обозначение `%0`, вместо которого макропроцессор во время макроподстановки подставляет число, равное фактическому количеству параметров.

Напомним, что сами аргументы многострочного макроса в его теле обозначаются как `%1`, `%2` и т. д., но средств индексирования (то есть способа извлечь n-ый параметр, где n вычисляется уже во время макроподстановки) NASM не предусматривает. Как же в таком случае использовать параметры, если даже их количество заранее не известно? Проблему решает директива `%rotate`, позволяющая переобозначить параметры. Рассмотрим самый простой вариант директивы:

```
%rotate 1
```

Числовой параметр обозначает, на сколько позиций следует сдвинуть номера параметров. В данном случае это число 1, так что параметр, ранее обозначавшийся `%2`, после этой директивы будет иметь обозначение `%1`, в свою очередь бывший `%3` превратится в `%2` и т. д., ну а параметр, стоявший самым первым и имевший обозначение `%1`, в силу «цикличности» нашего сдвига получит номер, равный общему количеству параметров. Обозначение `%0` в ротации не участвует и никак не изменяется.

Если директиве `%rotate` указать отрицательный параметр, она произведёт циклический сдвиг в обратном направлении (влево). Так, после

```
%rotate -1
```

`%1` будет обозначать параметр, ранее стоявший самым последним, `%2` станет обозначать параметр, ранее бывший первым (то есть имевший обозначение `%1`) и т. д.

Вспомним, что ранее (см. стр. 111) мы обещали написать макрос `pcall`, позволяющий в одну строчку сформировать вызов подпрограммы с любым количеством аргументов. Сейчас, имея в своём распоряжении макросы с переменным числом аргументов и директиву `%rotate`, мы готовы это сделать. Наш макрос, который мы назовём просто `pcall`, будет принимать на вход адрес процедуры (аргумент для команды `call`) и произвольное количество параметров, предназначенное для размещения в стеке. Мы будем, как и раньше, предполагать для простоты, что каждый параметр занимает ровно 4 байта. Напомним, что параметры должны быть помещены в стек в обратном порядке, начиная с последнего. Мы добьёмся этого с помощью макроцикла `%rep` и директивы `%rotate -1`, которая на каждом шаге будет делать последний (на

текущий момент) параметр параметром номер 1. Количество итераций цикла на единицу меньше, чем количество параметров, переданных в макрос, потому что первый из параметров является именем процедуры и его в стек заносить не надо. После этого цикла нам останется снова превратить последний параметр в первый (это как раз окажется самый первый из всех параметров, то есть адрес процедуры) и сделать `call`, а затем вставить команду `add` для очистки стека от параметров. Итак, пишем:

```
%macro pcall 1-*  ; от одного до скольки угодно
  %rep %0 - 1     ; цикл по всем параметрам, кроме первого
    %rotate -1    ; последний параметр становится %1
        push dword %1
  %endrep
  %rotate -1      ; адрес процедуры становится %1
        call %1
        add esp, (%0 - 1) * 4
%endmacro
```

Если теперь вызвать этот макрос, например, вот так:

```
        pcall myproc, eax, myvar, 27
```

то результатом подстановки станет следующий фрагмент:

```
        push dword 27
        push dword myvar
        push dword eax
        call myproc
        add esp, 12
```

что и требовалось.

3.5.8. Макродирективы для работы со строками

Ассемблер NASM поддерживает две директивы, предназначенные для преобразования строк (строковых констант) во время макропроцессирования. Они могут оказаться полезными, например, внутри многострочного макроса, одним из параметров которого является (должна быть) строка и эту строку нужно как-то преобразовать.

Первая из директив, `%strlen`, позволяет определить длину строки. Директива имеет два параметра. Первый из них — имя макропеременной, которой следует присвоить число, соответствующее длине строки, а второй — собственно строка. Так, в результате выполнения

```
%strlen sl 'my string'
```

макропеременная `s1` получит значение 9.

Вторая директива, `%substr`, позволяет выделить из строки символ с заданным номером. Например, после выполнения

```
%substr var1 'abcd' 1
%substr var2 'abcd' 2
%substr var3 'abcd' 3
```

макропеременные `var1`, `var2` и `var3` получат значения `'a'`, `'b'` и `'c'` соответственно, то есть эффект будет такой же, как если бы мы написали

```
%define var1 'a'
%define var2 'b'
%define var3 'c'
```

Всё это имеет смысл, как правило, только в случае, если в качестве аргумента директивы получают либо имя макропеременной, либо обозначение позиционного параметра в многострочном макросе.

Напомним, что все макродирективы отрабатывают во время макропроцессирования (*перед* компиляцией, то есть задолго до выполнения нашей программы), так что, разумеется, на момент соответствующих макроподстановок все используемые строки должны быть уже известны.

3.6. Взаимодействие с операционной системой

В этой главе мы рассмотрим средства взаимодействия пользовательской программы с операционной системой, что позволит в дальнейшем отказаться от использования макросов из файла `stud_io.inc`, а при желании и самостоятельно создавать их аналоги.

Пользовательские задачи обращаются к ядру операционной системы, используя так называемые ***системные вызовы***, которые, в свою очередь, реализованы через так называемые ***программные прерывания***. Чтобы понять, что это такое, нам придётся подробно обсудить, что такое, собственно говоря, *прерывания*, для чего они служат и откуда взялся странный термин «программное прерывание» (которое, заметим, ничего ни в каком виде не прерывает). Поэтому первые четыре параграфа этой главы мы посвятим изложению необходимых теоретических сведений, и лишь затем, имея готовую базу, рассмотрим механизм системных вызовов операционных систем Linux и FreeBSD на уровне машинных команд.

3.6.1. Мультизадачность и её основные виды

Как уже говорилось во введении, ***мультизадачность*** (режим мультипрограммирования) — это такой режим работы вычислительной системы, при котором несколько программ могут выполняться в

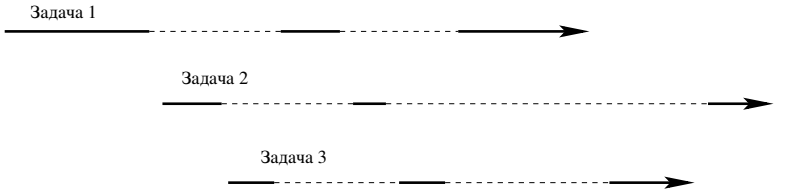

Рис. 3.7. Одновременное выполнение задач на одном процессоре

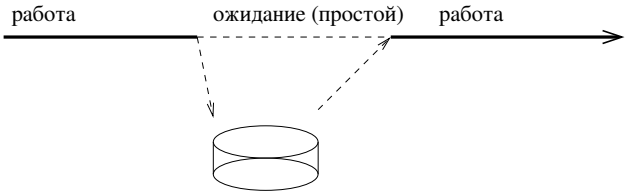

Рис. 3.8. Простой процессора в однозадачной системе

системе одновременно. Для этого, вообще говоря, не нужно несколько физических *процессоров*. Вычислительная система может иметь всего один процессор, что не мешает само по себе реализации режима мультипрограммирования. Так или иначе, количество процессоров в системе в общем случае меньше количества одновременно выполняемых программ. Ясно, что процессор в каждый момент времени может выполнять только одну программу. Что же, в таком случае, понимается под мультипрограммированием?

Кажущийся парадокс разрешается введением следующего определения ***одновременности*** для случая выполняющихся программ (***процессов***, или ***задач***): **две задачи, запущенные на одной вычислительной системе, называются выполняемыми *одновременно*, если периоды их выполнения (временно́й отрезок с момента запуска до момента завершения каждой из задач) полностью или частично перекрываются.**

Иными словами, если процессор, работая в каждый момент времени с одной задачей, при этом переключается между несколькими задачами, уделяя внимание то одной, то другой, эти задачи в соответствии с нашим определением будут считаться выполняемыми одновременно (см. рис. 3.7).

В простейшем случае мультизадачность позволяет решить проблему простоя центрального процессора во время операций ввода-вывода. Представим себе вычислительную систему, в которой выполняется одна задача (например, обсчет сложной математической модели). В некоторый момент времени задаче может потребоваться операция обмена данными с каким-либо внешним устройством (например, чтение оче-

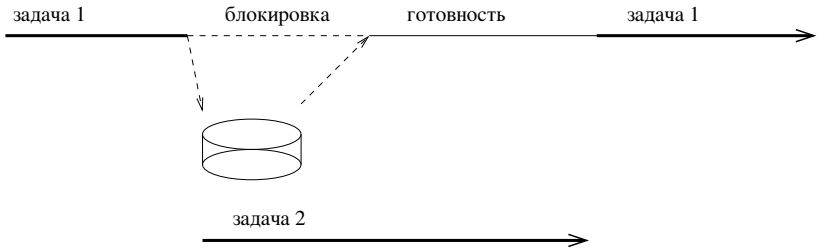

Рис. 3.9. Пакетная ОС

редного блока входных данных либо, наоборот, запись конечных или промежуточных результатов). Скорость работы внешних устройств (дисков и т. п.) обычно на порядки ниже, чем скорость работы центрального процессора, и в любом случае никоим образом не бесконечна. Так, для чтения заданного блока данных с диска необходимо включить привод головки, чтобы переместить её в нужное положение (на нужную дорожку) и дождаться, пока сам диск повернётся на нужный угол (для работы с заданным сектором); затем, пока сектор проходит под головкой, прочитать записанные в этом секторе данные во внутренний буфер контроллера диска[38]; наконец, следует разместить прочитанные данные в той области памяти, где их появления ожидает пользовательская программа, и лишь после этого вернуть ей управление. Всё это время (как минимум, время, затрачиваемое на перемещение головки и ожидание нужной фазы поворота диска) центральный процессор будет простаивать (рис. 3.8).

Если задача у нас всего одна и больше делать нечего, такой простой не создаёт проблем, но если кроме той задачи, которая уже работает, у нас есть и другие задачи, дожидающиеся своего часа, то лучше употребить время центрального процессора, впустую пропадающее в ожидании окончания операций ввода-вывода, на решение других задач. Именно так поступают мультизадачные операционные системы. В такой системе из задач, которые нужно решать, формируется *очередь заданий*. Как только активная задача затребует проведение операции ввода-вывода, операционная система выполняет необходимые действия по запуску контроллеров устройств на исполнение запрошенной операции либо ставит запрошенную операцию в очередь, если начать её немедленно по каким-то причинам нельзя, после чего активная задача заменяется на другую — новую (взятую из очереди) или уже выполнявшуюся раньше, но не успевшую завершиться. Замененная задача в этом случае считается перешедшей в состояние ожидания результата ввода-вывода, или ***состояние блокировки***.

[38] Чтение непосредственно в оперативную память теоретически возможно, но технически сопряжено с определенными трудностями и применяется редко.

В простейшем случае новая активная задача остается в режиме выполнения до тех пор, пока она не завершится либо не затребует в свою очередь проведение операции ввода-вывода. При этом блокированная задача по окончании операции ввода-вывода переходит из состояния блокировки в ***состояние готовности к выполнению***, но переключения на нее не происходит (см. рис. 3.9); это обусловлено тем, что операция смены активной задачи, вообще говоря, отнимает много процессорного времени. Такой способ построения мультизадачности, при котором смена активной задачи происходит только в случае ее окончания или запроса на операцию ввода-вывода, называется ***пакетным режимом***[39], а операционные системы, реализующие этот режим,— ***пакетными операционными системами***. Режим пакетной мультизадачности является самым эффективным с точки зрения использования вычислительной мощности центрального процессора, поэтому именно пакетный режим используется для управления суперкомпьютерами и другими машинами, основное назначение которых — большие объёмы численных расчетов.

С появлением первых терминалов и диалогового (иначе говоря, интерактивного) режима работы с компьютерами возникла потребность в других стратегиях смены активных задач, или, как принято говорить, ***планирования времени центрального процессора***. Действительно, пользователю, ведущему диалог с той или иной программой, вряд ли захочется ждать, пока некая активная задача, вычисляющая, скажем, обратную матрицу порядка 1000x1000, завершит свою работу. При этом много процессорного времени на обслуживание диалога с пользователем не требуется: в ответ на каждое действие пользователя (например, нажатие на клавишу) обычно необходимо выполнить набор действий, укладывающийся в несколько миллисекунд, тогда как самих таких событий пользователь даже в режиме активного набора текста может создать никак не больше трёх-четырёх в секунду (скорость компьютерного набора 200 символов в минуту считается довольно высокой). Было бы нелогично ждать, пока пользователь полностью завершит свой диалоговый сеанс: бóльшую часть времени процессор мог бы производить арифметические действия, необходимые для задачи, вычисляющей матрицу.

Решить проблему позволяет ***режим разделения времени***. В этом режиме каждой задаче отводится определенное время работы, называемое ***квантом времени***. По окончании этого кванта, если в системе имеются другие готовые к исполнению задачи, активная зада-

[39]Русскоязычный термин «пакетный режим» является устоявшимся, хотя и не слишком удачным переводом английского термина «batch mode»; слово *batch* можно также перевести как «колода» (собственно, изначально имелись в виду колоды перфокарт, олицетворявшие задания). Не следует путать этот термин со словами, происходящими от английского слова packet, которое тоже обычно переводится на русский как «пакет».

ча принудительно приостанавливается и заменяется другой задачей. Приостановленная задача помещается в ***очередь задач, готовых к выполнению*** и находится там, пока остальные задачи отработают свои кванты; затем она снова получает очередной квант времени для работы, и т. д. Естественно, если активная задача затребовала операцию ввода-вывода, она переводится в состояние блокировки (точно так же, как и в пакетном режиме). Задачи, находящиеся в состоянии блокировки, не ставятся в очередь на выполнение и не получают квантов времени до тех пор, пока операция ввода-вывода не будет завершена (либо не исчезнет другая причина блокировки), и задача не перейдет в состояние готовности к выполнению.

Существуют различные алгоритмы поддержки очереди на выполнение, в том числе и такие, в которых задачам приписывается некоторый приоритет, выраженный числом. Например, в ОС Unix обычно задача имеет две составляющие приоритета — статическую и динамическую; статическая составляющая представляет собой назначенный администратором уровень «важности» выполнения данной конкретной задачи, динамическая же изменяется планировщиком: пока задача выполняется, её динамический приоритет падает, когда же она находится в очереди на исполнение, динамическая составляющая приоритета, напротив, растёт. Из нескольких готовых к исполнению задач выбирается имеющая наибольшую сумму приоритетов, так что рано или поздно задача даже с самым низким статическим приоритетом получит управление за счёт возросшего динамического приоритета.

Некоторые операционные системы, включая ранние версии Windows, применяли стратегию, занимающую промежуточное положение между пакетным режимом и режимом разделения времени. В этих системах задачам выделялся квант времени, как и в системах разделения времени, но принудительной смены текущей задачи по истечении кванта времени не производилось; система проверяла, не истёк ли квант времени у текущей задачи, только когда задача обращалась к операционной системе (не обязательно за вводом-выводом). Задача, не нуждающаяся в услугах операционной системы, могла оставаться на процессоре сколь угодно долго, как и в пакетных операционных системах. Такой режим работы называется ***невытесняющим***. В современных системах он не применяется, поскольку налагает слишком жесткие требования на исполняемые в системе программы; так, в ранних версиях Windows любая программа, занятая длительными вычислениями, блокировала работу всей системы, а зациклившаяся задача приводила к необходимости перезагрузки компьютера.

Иногда режим разделения времени также оказывается непригоден. В некоторых ситуациях, таких как управление полётом самолёта, ядерным реактором, автоматической линией производства и т. п., некоторые задачи должны быть завершены строго до определённого момента времени; так, если автопилот самолёта, получив сигнал от датчиков тангажа и крена, потратит на вычисление необходимого корректирую-

щего воздействия больше времени, чем допустимо, самолёт может вовсе потерять управление.

В случае, когда выполняемые задачи (как минимум некоторые из них) имеют жёсткие рамки по времени завершения, применяются ***операционные системы реального времени***. В отличие от систем разделения времени, задача планировщика реального времени не в том, чтобы дать всем программам отработать некоторое время, а в том, чтобы *обеспечить завершение каждой задачи за отведённое ей время*, если же это невозможно — снять задачу, освободив процессор для тех задач, которые ещё можно успеть завершить к сроку. В системах реального времени более важным считается не *общее количество задач*, решённых в системе за фиксированное время (так называемая *производительность* системы), а *предсказуемость* времени выполнения для каждой отдельно взятой задачи. Планирование в системах реального времени — это довольно сложный раздел наук о вычислениях, достойный отдельной книги и заведомо выходящий за рамки нашего учебника. На практике вы вряд ли когда-нибудь столкнётесь с системами реального времени, во всяком случае, в роли программиста; если такое всё же произойдёт, вам потребуется потратить время на изучение специальной литературы, но так происходит в любой специфической области инженерной деятельности.

3.6.2. Аппаратная поддержка мультизадачности

Ясно, что для построения мультизадачного режима работы вычислительной системы аппаратура (прежде всего сам центральный процессор) должна обладать определёнными свойствами. О некоторых из них мы уже говорили в §3.1.2 — это, во-первых, защита памяти, а во-вторых, разделение машинных команд на обычные и привилегированные, с отключением возможности выполнения привилегированных команд в ограниченном режиме работы центрального процессора.

Действительно, при одновременном нахождении в памяти машины нескольких программ, если не предпринять специальных мер, одна из программ может модифицировать данные или код других программ или самой операционной системы. Даже если допустить отсутствие злого умысла у разработчиков запускаемых программ, от случайных ошибок в программах нас это не спасёт, причём такая ошибка может, с одной стороны, привести к тяжёлым авариям всей системы, а с другой стороны — оказаться совершенно неуловимой, вплоть до абсолютной невозможности установить, какая из задач «виновата» в происходящем. Дело в том, что для обнаружения и устранения ошибки необходима возможность воссоздания обстоятельств, при которых она проявляется, а точно воссоздать состояние всей системы со всеми запущенными в ней задачами практически невозможно. Очевидно, нужны средства огра-

ничения возможностей работающей программы по доступу к областям памяти, занятым другими программами. Программно такую защиту можно реализовать разве что путем интерпретации всего машинного кода исполняющейся программы, что, как правило, недопустимо из соображений эффективности. Следовательно, необходима *аппаратная* поддержка защиты памяти, позволяющая ограничить возможности текущей задачи по доступу к оперативной памяти.

Коль скоро существует защита памяти, процессор должен иметь набор команд для управления этой защитой. Если, опять-таки, не предпринять специальных мер, то такие команды сможет исполнить любая из выполняющихся программ, сняв защиту памяти или модифицировав ее конфигурацию, что сделало бы саму защиту памяти практически бессмысленной. Рассматриваемая проблема касается не только защиты памяти, но и работы с внешними устройствами. Как уже говорилось, чтобы обеспечить нормальное взаимодействие всех программ с устройствами ввода-вывода, операционная система должна взять непосредственную работу с устройствами на себя, а пользовательским программам предоставлять интерфейс для обращения к операционной системе за услугами по работе с устройствами, причём пользовательские программы должны иметь возможность работы с внешними устройствами только через операционную систему. Следовательно, необходимо запретить пользовательским программам выполнение команд процессора, осуществляющих чтение/запись портов ввода-вывода. Вообще, передавая управление пользовательской программе, операционная система должна быть уверена, что задача не сможет (иначе как обратившись к самой операционной системе) выполнить никакие действия, влияющие на систему в целом.

Проблема решается введением двух режимов работы центрального процессора: ***привилегированного*** и ***ограниченного***. В литературе привилегированный режим часто называют «режимом ядра» или «режимом супервизора» (англ. ***kernel mode***, ***supervisor mode***). Ограниченный режим называют также «пользовательским режимом» (англ. ***user mode***) или просто ***непривилегированным*** (англ. ***nonprivileged***). Термин ***ограниченный режим*** избран нами как наиболее точно описывающий сущность этого режима работы центрального процессора без привязки к его использованию операционными системами. В привилегированном режиме процессор может выполнять любые существующие команды. В ограниченном режиме выполнение команд, влияющих на систему в целом, запрещено; разрешаются только команды, эффект которых ограничен модификацией данных в областях памяти, не закрытых защитой памяти. Сама операционная система выполняется в привилегированном режиме, пользовательские программы — в ограниченном.

Как мы уже отмечали в §3.1.2, пользовательская программа может только модифицировать данные в отведённой ей памяти; любые другие действия требуют обращения к операционной системе. Это обеспечивается поддержкой в центральном процессоре механизма защиты памяти и наличием ограниченного режима работы. Соблюдения этих двух аппаратных требований, однако, ещё не достаточно для реализации мультизадачного режима работы системы.

Вернемся к ситуации с операцией ввода-вывода. В однозадачной системе (рис. 3.8 на стр. 127) во время исполнения операции ввода-вывода центральный процессор мог непрерывно опрашивать контроллер устройства на предмет его готовности (завершена ли требуемая операция), после подготовить всё к возобновлению работы активной задачи — в частности, скопировать прочитанные данные из буфера контроллера в память, принадлежащую задаче. Следует отметить, что в этом случае процессор был бы непрерывно занят во время операции ввода-вывода, несмотря на то, что никаких полезных вычислений он бы при этом не производил. Такой способ взаимодействия называется ***активным ожиданием***. Ясно, что процессорное время можно было бы расходовать с бóльшей пользой.

При переходе к мультизадачной обработке, показанной на рис. 3.9 на стр. 128, возникает другая проблема. В момент завершения операции ввода-вывода процессор занят исполнением второй задачи. Между тем в момент завершения операции требуется как минимум перевести первую задачу из состояния блокировки в состояние готовности; могут потребоваться и другие действия, такие как копирование данных из буфера контроллера, сброс контроллера (например, выключение мотора диска), а в более сложных ситуациях — инициирование другой операции ввода-вывода, ранее отложенной (это может быть операция чтения с того же диска, которую затребовала другая задача в то время, когда первая операция ещё выполнялась). Всё это должна сделать операционная система. Но каким образом она узнает о завершении операции ввода-вывода, если процессор при этом занят выполнением другой задачи и непрерывного опроса контроллера не производит?

Решить проблему позволяет аппарат ***прерываний***. В случае операции с диском в момент её завершения контроллер диска подаёт центральному процессору определённый сигнал (электрический импульс), называемый ***запросом прерывания***. Центральный процессор, получив этот сигнал, прерывает выполнение активной задачи и передаёт управление процедуре операционной системы, которая выполняет все действия, необходимые по окончании операции ввода-вывода. Такая процедура называется ***обработчиком прерывания***. После завершения процедуры-обработчика управление возвращается активной задаче.

Для реализации пакетного мультизадачного режима достаточно, чтобы на уровне аппаратуры были реализованы прерывания, защита

памяти и два режима работы процессора. Если же нужно реализовать систему разделения времени или тем более систему реального времени, требуется наличие в аппаратуре ещё одного компонента — ***таймера***. Действительно, планировщику операционной системы разделения времени нужна возможность отслеживать истечение квантов времени, выделенных пользовательским программам; в системе реального времени такая возможность также необходима, причём требования к ней даже более жёсткие: не сняв вовремя с процессора активное на тот момент приложение, планировщик рискует не успеть выделить более важным программам требуемое процессорное время, что чревато неприятными последствиями (вспомните пример с автопилотом самолёта). Таймер представляет собой сравнительно простое устройство, вся функциональность которого сводится к генерации прерываний через равные промежутки времени. Эти прерывания дают возможность операционной системе получить управление, проанализировать текущее состояние имеющихся задач и при необходимости сменить активную задачу.

Итак, для реализации мультизадачной операционной системы аппаратное обеспечение компьютера обязано поддерживать:

- аппарат прерываний;
- защиту памяти;
- привилегированный и ограниченный режимы работы центрального процессора;
- таймер.

Первые три свойства необходимы в любой мультизадачной системе, последнее может отсутствовать в случае пакетной планировки (хотя в реально существующих системах таймер присутствует всегда). Следует обратить внимание, что из перечисленого только таймер представляет собой отдельное устройство, остальное — особенности центрального процессора.

Теоретически при наличии таймера можно сделать прерывание по таймеру *единственным* прерыванием в системе. Операционная система, получив управление в результате такого прерывания, должна будет уже сама опросить все активные контроллеры внешних устройств на предмет завершения выполнявшихся операций. Реально такая схема порождает множество проблем, прежде всего с эффективностью, а выигрыш от её применения неочевиден.

3.6.3. Прерывания и исключения

Современный термин «***прерывание***» довольно далеко ушел в своем развитии от изначального значения; начинающие программисты часто с удивлением обнаруживают, что некоторые прерывания вовсе ничего не прерывают. Дать строгое определение прерывания было бы несколько затруднительно. Вместо этого попытаемся объяснить сущ-

ность различных видов прерываний и найти между ними то общее, что и оправдывает существование самого термина.

Прерывания в изначальном смысле уже знакомы нам из предыдущего параграфа. Те или иные устройства вычислительной системы могут осуществлять свои функции независимо от центрального процессора; время от времени им может требоваться внимание операционной системы, но единственный центральный процессор (или, что ничуть не лучше, все имеющиеся в системе центральные процессоры) может быть именно в такой момент занят обработкой пользовательской программы. Аппаратные (или ***внешние***) прерывания были призваны решить эту проблему. Для поддержки аппаратных прерываний процессор имеет специально предназначенные для этого контакты; электрический импульс, поданный на такой контакт, воспринимается процессором как сигнал, что некоему устройству требуется внимание операционной системы. В современных архитектурах, основанных на общей шине, для запроса прерывания используется одна из дорожек шины.

Последовательность событий при возникновении и обработке прерывания выглядит приблизительно так[40].

- Устройство, которому требуется внимание процессора, устанавливает на шине сигнал «запрос прерывания».
- Процессор доводит выполнение текущей программы до такой точки, в которой выполнение можно прервать так, чтобы потом восстановить его с того же места; после этого процессор выставляет на шине сигнал «подтверждение прерывания». При этом другие прерывания блокируются.
- Получив подтверждение прерывания, устройство передаёт по шине некоторое число, идентифицирующее данное устройство; это число называют ***номером прерывания***.
- Процессор сохраняет где-то (обычно в стеке активной задачи) текущие значения счётчика команд и регистра флагов; это называется ***малым упрятыванием***. Счётчик команд и регистр флагов должны быть сохранены по той причине, что выполнение первой же инструкции обработчика прерывания изменит (испортит) и то, и другое, сделав невозможным прозрачный (т. е. незаметный для пользовательской задачи) возврат из обработчика; остальные регистры обработчик прерывания может при необходимости сохранить самостоятельно.
- Устанавливается привилегированный режим работы центрального процессора, после чего управление передаётся на точку входа процедуры в операционной системе, называемой, как мы уже говорили, ***обработчиком прерывания***. Адрес обработчика может быть предварительно считан из специальных областей памяти либо вычислен иным способом.

[40] Здесь приводится общая схема; в действительности всё намного сложнее.

Напомним, что переключение из привилегированного режима работы центрального процессора в ограниченный можно осуществить простой командой, поскольку в привилегированном режиме доступны все возможности процессора; в то же время переход из ограниченного (пользовательского) режима обратно в привилегированный произвести с помощью обычной команды нельзя, поскольку это лишило бы смысла само существование привилегированного и ограниченного режимов. В этом плане **прерывание интересно ещё и тем, что перед его обработкой режим работы центрального процессора становится привилегированным**.

Уже упоминавшееся выше устройство «таймер» является, пожалуй, самым простым из всех внешних устройств: всё, что он делает — подаёт запросы на прерывание через равные промежутки времени (например, 1000 раз в секунду на рассматриваемых нами процессорах).

Рассмотрим теперь следующий вопрос: что следует делать центральному процессору, если активная задача выполнила целочисленное деление на ноль? Ясно, что дальнейшее выполнение программы лишено смысла: результат деления на ноль невозможно представить каким-либо целым числом, так что в переменной, которая должна была содержать результат произведённого деления, в лучшем случае будет содержаться мусор; конечные результаты, скорее всего, окажутся иррелевантными. Пытаться оповестить программу о происшедшем, выставляя какой-нибудь флаг, очевидно, также бессмысленно: если программист не произвёл *перед* выполнением деления проверку делителя на равенство нулю, представляется и вовсе ничтожной вероятность того, что он станет проверять *после* деления значение какого-то флага.

Завершить текущую задачу процессор самостоятельно не может, это слишком сложное действие, зависящее от реализации операционной системы. Всё, что ему остаётся — передать управление операционной системе, известив её о происшедшем. Что делать с аварийной задачей, операционная система решит самостоятельно. Для этого требуется, очевидно, переключиться в привилегированный режим и передать управление коду операционной системы; перед этим желательно сохранить регистры (хотя бы счётчик команд и регистр флагов); даже если задача ни при каких условиях не будет продолжена с того же места (а предполагать это процессор, вообще говоря, не вправе), значения регистров в любом случае могут пригодиться операционной системе для анализа происшествия. Более того, каким-то образом следует сообщить операционной системе о причине того, что управление передано ей; кроме деления на ноль, такими причинами могут быть нарушение защиты памяти, попытка выполнить запрещённую или несуществующую инструкцию и т. п. Все такие ситуации называются ***исключениями***.

Легко заметить, что действия, которые должен выполнить процессор при возникновении исключения, оказываются очень похожи на рассмотренный ранее случай аппаратного прерывания. Основное отличие состоит в отсутствии обмена по шине (запроса и подтверждения прерывания): действительно, информация о перечисленных событиях возникает внутри процессора, а не вне его[41]. Остальные шаги по обработке исключений повторяют шаги по обработке аппаратного прерывания практически дословно.

Ещё одно кардинальное отличие ситуации недопустимых действий задачи от аппаратного прерывания состоит, собственно говоря, в заведомом *наличии* задачи, ответственной за происходящее, а задача является *единицей планирования*, то есть выполнение задачи можно приостановить, а затем продолжить с того же места. Это позволяет операционной системе проводить обработку исключений совершенно иначе, чем аппаратных прерываний. Говорят, что обработка исключения происходит *в контексте пользовательской задачи*.

Несмотря на различия, обработка ситуаций, в которых процессор не может по тем или иным причинам выполнить очередную команду, схожа с аппаратными прерываниями хотя бы в том, что где-то в памяти должен располагаться *обработчик*, на который следует при наступлении соответствующей ситуации передать управление, причём адрес, по которому находится обработчик, должен быть настраиваемым, но, конечно же, такая настройка должна быть действием привилегированным, чтобы только операционная система могла указать процессору, куда передавать управление, когда это потребуется. Разработчики процессоров x86 пошли здесь довольно простым путём. Все обработчики, предназначенные как для аппаратных прерываний, так и для исключений, снабжены номерами от `0` до `255`; в оперативной памяти выделяется специальная область для хранения так называемой ***таблицы дескрипторов прерываний*** (*interrupt descriptor table*). Эта таблица содержит некоторое количество (например, 256) записей по восемь байт; каждая такая запись соответствует своему обработчику и содержит информацию о том, как именно следует передавать ему управление — по какому адресу, в какой сегмент[42], следует ли при этом временно заблокировать аппаратные прерывания и т. п.

Поскольку нумерация обработчиков сквозная и включает как аппаратные прерывания, так и исключения, нам не покажется странной терминология, фактически введённая создателями x86: исключения они называют ***внутренними прерываниями*** (*internal interrupts*).

[41] С точки зрения аппаратной реализации исключения могут оказаться много проще, чем аппаратные прерывания, так как они всегда происходят на определённой фазе выполнения инструкции; подробности читатель найдет в книге [1].

[42] Конечно, мы помним, что операционные системы обычно не используют сегментную составляющую виртуальной памяти на i386, но сама эта составляющая никуда не исчезает.

Такая терминология оправдывается тем, что причина внешнего прерывания находится вне центрального процессора, тогда как причина внутреннего — у ЦП внутри. Нужно отметить, что при описании других процессоров обычно такую терминологию не используют: аппаратные (они же внешние) прерывания называют просто прерываниями, а исключения так и называют исключениями (*exceptions*), ***ловушками*** (*traps*) или как-то ещё.

Подчеркнём, что **внутренние прерывания никого и ничего не прерывают!** Их название (даже если называть их именно так, а не исключениями) оправдано лишь тем, что они используют общую с аппаратными прерываниями нумерацию и систему организации обработчиков. В действительности при возникновении внутреннего прерывания обработчик, хотя и находится в ядре операционной системы, являясь её частью, выполняется *в рамках задачи как единицы планирования*, так что задачу вообще было бы некорректно считать прерванной в каком бы то ни было смысле; но даже если бы это было не так, странно было бы считать, что выполнение задачи *кто-то прервал*, когда на самом деле она сама своими действиями устроила аварию. Заметим, *аппаратные* прерывания совершенно очевидным образом *действительно прерывают выполнение текущей задачи.*

3.6.4. Системные вызовы и «программные прерывания»

Как уже говорилось, пользовательской задаче не позволяется делать ничего, кроме преобразования данных в отведённой ей памяти. Все действия, затрагивающие внешний по отношению к задаче мир, выполняются через операционную систему. Следовательно, нужен механизм, позволяющий пользовательской задаче обратиться к ядру операционной системы за теми или иными услугами. Напомним, что **обращение пользовательской задачи к ядру операционной системы за услугами называется *системным вызовом***. Ясно, что по своей сути системный вызов — это передача управления от пользовательской задачи ядру операционной системы. Однако здесь есть две проблемы. Во-первых, ядро работает в привилегированном режиме, а пользовательская задача — в ограниченном. Во-вторых, пространство адресов ядра для пользовательской задачи обычно недоступно (более того, в адресном пространстве задачи этих адресов может вообще не быть). Впрочем, даже если бы оно было доступно, позволить пользовательской задаче передавать управление в произвольную точку ядра было бы несколько странно.

Итак, для осуществления системного вызова необходимо сменить режим выполнения с пользовательского на привилегированный и передать управление в некоторую точку входа в операционной системе. Всё это

должно происходить по инициативе пользовательской задачи, то есть для этого нужна какая-нибудь специальная команда центрального процессора. На разных архитектурах соответствующая инструкция может называется `trap` (*ловушка*), `svc` (*supervisor call*, то есть «обращение к супервизору») и т. д. На некоторых архитектурах, включая современную 64-битную «наследницу» рассматриваемой i386, эта инструкция называется просто `syscall`, то есть «системный вызов». Естественно, то, *куда* (и как) при этом будет передано управление, должна определять операционная система, в которой, разумеется, должен присутствовать некий фрагмент кода, специально предназначенный для обработки системных вызовов (т. е. *обработчик*).

Что-то похожее мы уже видели, рассматривая аппаратные и внутренние прерывания. Создатели процессоров x86 решили не изобретать отдельного механизма для системных вызовов; вместо этого в систему команд ввели команду `int` (от слова *interrupt* — «прерывание»), которая изначально предназначалась буквально для принудительного вызова обработчика прерывания; в те времена, впрочем, ещё не было ни привилегированного режима, ни виртуальной памяти. Команду (или, точнее, её эффект) назвали ***программным прерыванием*** (*software interrupt*). На i386 командой `int` может быть вызван уже не любой обработчик, а только такой, который специально для этого предназначен.

Итак, если принять терминологию, характерную для процессоров x86, мы можем различать *три вида прерываний*: внешние (они же аппаратные), внутренние и программные. Отличие программного прерывания от остальных состоит в том, что оно происходит по инициативе пользовательской задачи, тогда как другие прерывания случаются без её ведома: внешние — по требованию внешних устройств, внутренние — в случае непредвиденных обстоятельств, которые вряд ли были выполняемой программой предусмотрены. Если мы берём на вооружение термин «программное прерывание», то можно сказать, что *системные вызовы реализуются через программные прерывания.*

То, что **программное прерывание уж точно ничего не прерывает**, представляется очевидным, а сам термин «программное прерывание» при внимательном рассмотрении кажется оксюмороном; неудивительно, что при описании архитектур, отличных от x86, чаще всего прерываниями называют только «настоящие» (читай — аппаратные) прерывания; как уже упоминалось, вместо термина «внутренние прерывания» в этом случае используют термин «исключения» или «ловушки», а вместо «программного прерывания» говорят просто о *системном вызове*, не делая различия между самим вызовом и механизмом его реализации.

Так или иначе, повышение уровня привилегий (переход из ограниченного режима в привилегированный) возможно только при условии одновременной передачи управления на заранее заданную точку входа,

причём адреса возможных точек входа могут настраиваться только в привилегированном режиме. Таким образом операционная система получает возможность гарантировать, что при смене режима работы на привилегированный управление получит только код самой операционной системы, причем только такой её код, который для этого специально предназначен. Исполнение в привилегированном режиме пользовательского кода полностью исключается. Применяя терминологию с «тремя видами прерываний», мы можем сказать, что переключение режима с ограниченного на привилегированный происходит только при прерывании (любого из трёх видов), тогда как если мы прерываниями называем только «настоящие» прерывания, придётся сказать, что режим ЦП меняется на привилегированный в трёх случаях: при прерывании, исключении и системном вызове.

Следует отметить, что соглашения о том, как конкретно должен происходить системный вызов, как передать ему необходимые параметры, какое использовать прерывание, как получить результат выполнения и т. п., варьируются от системы к системе. Даже если речь идёт о двух представителях семейства Unix (ОС Linux и ОС FreeBSD), работающих на одной и той же аппаратной платформе i386, низкоуровневая реализация системных вызовов оказывается в них совершенно различна. Следующие два параграфа посвящены описанию соглашений об организации системных вызовов этих двух систем[43]; при желании вы можете прочитать только один из этих двух параграфов, относящийся к той системе, которую вы используете.

Следует иметь в виду, что системы семейства Unix рассчитаны в основном на программирование на языке Си. Для этого языка вместе с системой поставляются библиотеки, облегчающие работу с системными вызовами — в частности, для каждого системного вызова предоставляется библиотечная функция («обёртка»), позволяющая обратиться к услугам ядра как к обычной подпрограмме. Системные вызовы в ОС Unix имеют названия, совпадающие с именами соответствующих функций-обёрток из библиотеки языка Си. К сожалению, такая ориентированность на Си приводит к некоторым неудобствам при работе на уровне языка ассемблера. Так, системные вызовы при переходе от системы к системе могут менять свои номера: например, `getppid` в ОС Linux имеет номер 64, а в ОС FreeBSD — номер 39. Программисты, работающие на языке Си, об этом могут не задумываться, поскольку в любой системе семейства Unix им достаточно вызвать обычную функцию с именем `getppid`, а конкретное исполнение системного вызова возлагается на библиотеку, которая прилагается к системе, так что программа, написанная программистом на Си с использованием `getppid`, будет успешно компилироваться в любой системе и работать одинаково.

[43] Естественно, в варианте для i386; версии, предназначенные для других аппаратных архитектур, устроены иначе.

Когда мы пишем на языке ассемблера, никакой библиотеки системных вызовов у нас нет, номер вызова мы должны указать явно, так что в тексте, предназначенном для Linux, придётся использовать число `64`, а для FreeBSD — `39`. Получается, что исходный текст будет пригоден для одной системы и ошибочен для другой. Аналогично обстоят дела и с некоторыми числовыми константами, которые вызовы получают на вход. Частично нас может выручить макропроцессор с его директивами условной компиляции, либо мы можем ограничиться только одной системой (что на самом деле не совсем правильно). К счастью, системы FreeBSD и Linux всё же во многом похожи; числовые значения, связанные с системными вызовами, частично совпадают (с другими системами семейства Unix было бы хуже). Так или иначе, кто предупреждён, тот вооружён.

3.6.5. Конвенция системных вызовов ОС Linux

Ядро Linux на платформе i386 использует для осуществления системного вызова программное прерывание с номером `80h`. Номер системного вызова передаётся ядру через регистр `EAX`; если системный вызов принимает параметры, то они располагаются в регистрах `EBX`, `ECX`, `EDX`, `ESI` и `EDI`; отметим, что все параметры системных вызовов являются четырёхбайтными значениями — либо целочисленными, либо адресными. Результат выполнения вызова возвращается через регистр `EAX`, причём значение, заключённое между `fffff000h` и `ffffffffh`, свидетельствует об ошибке и представляет собой условный код этой ошибки.

Рассмотрим для примера системный вызов `write`, позволяющий произвести вывод данных через один из открытых потоков ввода-вывода, в том числе запись в открытый файл, а также в стандартный поток вывода (в просторечии «на экран»). Этот системный вызов имеет номер `4` и принимает три параметра: дескриптор (номер) потока ввода-вывода, адрес памяти, где расположены данные, подлежащие выводу, и количество этих данных в байтах. Отметим, что поток стандартного вывода в ОС Unix имеет дескриптор `1` (точнее, поток вывода под номером `1` считается стандартным выводом). Таким образом, если мы хотим вывести строку «на экран», то есть сделать то, что делает макрос `PRINT`, нам нужно будет занести число `4` в `EAX`, занести число `1` в `EBX`, занести адрес строки в `ECX` и длину строки — в `EDX`, а затем дать команду `int 80h`, чтобы инициировать программное прерывание.

Другой важный системный вызов — это вызов `_exit`, используемый для завершения программы. Он имеет номер `1` и принимает один параметр, представляющий собой ***код завершения***. Программы используют код завершения, чтобы сообщить операционной системе, успешно ли они справились с возложенной на них задачей: если всё прошло как

ожидалось, используется код 0, если же в ходе работы возникли ошибки, используются коды 1, 2 и т. д.

Зная всё это, мы можем написать программу, печатающую строку и сразу после этого завершающуюся; файл `stud_io.inc` и его макросы нам для этого больше не нужны:

```
global  _start

section .data
msg     db "Hello world", 10
msg_len equ $-msg

section .text
_start: mov     eax, 4          ; вызов write
        mov     ebx, 1          ; стандартный вывод
        mov     ecx, msg
        mov     edx, msg_len
        int     80h

        mov     eax, 1          ; вызов _exit
        mov     ebx, 0          ; код "успех"
        int     80h
```

Некоторые системные вызовы не укладываются в приведённую конвенцию; например, вызов `mmap` предполагает шесть параметров, а у вызова `lseek64` один из параметров — 64-битный. Как поступают ядро и библиотека в таких нестандартных случаях — вопрос, который мы оставим за рамками нашей книги.

3.6.6. Конвенция системных вызовов ОС FreeBSD

Описание конвенции ОС FreeBSD несколько сложнее. Эта система также использует прерывание `80h` и принимает номер системного вызова через регистр `EAX`, но все параметры вызова передаются не через регистры, а через стек, подобно тому, как передаются параметры в подпрограммы в соответствии с соглашениями языка Си, то есть в обратном порядке (см. стр. 91). Как и в ОС Linux, все параметры вызовов представляют собой четырёхбайтные значения. Результат выполнения системного вызова возвращается через регистр `EAX`, но при этом об ошибке свидетельствует не попадание значения в специальный промежуток (как это сделано в Linux), а установленное значение флага `CF`. Если `CF` сброшен, то вызов завершился успешно и его результат находится в `EAX`, если же флаг установлен, то произошла ошибка и в `EAX` записан код этой ошибки.

Необходимо отметить ещё одну особенность. Ядро FreeBSD предполагает, что управление ему передано путём обращения к процедуре следующего вида:

```
kernel:
        int 80h
        ret
```

Если у нас есть такая процедура, нам для обращения к ядру достаточно поместить в стек параметры точно так же, как для обычной процедуры, занести номер вызова в `EAX` и сделать `call kernel`; при этом команда `call` занесёт в стек адрес возврата, который и будет лежать на вершине стека в момент выполнения программного прерывания, а параметры будут располагаться в стеке ниже вершины. Ядро FreeBSD учитывает это и ничего не делает с числом на вершине стека (ведь это число — адрес возврата из процедуры `kernel` — никакого отношения к параметрам вызова не имеет), а настоящие параметры извлекает из стека ниже вершины (из позиций `[esp+4]`, `[esp+8]` и т. д.)

При работе на языке ассемблера выделять вызов прерывания в отдельную подпрограмму не обязательно, достаточно перед командой `int` занести в стек дополнительное «двойное слово», например, выполнив лишний раз команду `push eax` (или любой другой 32-битный регистр). После выполнения системного вызова и возврата из него следует убрать из стека всё, что туда было занесено; делается это, как и при вызове обычных подпрограмм, путём увеличения регистра `ESP` на нужную величину простой командой `add`.

Описывая в предыдущем параграфе конвенцию ОС Linux, мы для иллюстрации использовали вызовы `write` и `_exit` (см. стр. 141). Аналогичная программа для FreeBSD будет выглядеть следующим образом:

```
global  _start

section .data
msg     db "Hello world", 10
msg_len equ $-msg

section .text
_start:
        push    dword msg_len
        push    dword msg
        push    dword 1         ; стандартный вывод
        mov     eax, 4          ; write
        push    eax             ; что угодно
        int     80h
        add     esp, 16         ; 4 двойных слова

        push    dword 0         ; код "успех"
        mov     eax, 1          ; вызов _exit
        push    eax             ; что угодно
        int     80h
```

Мы не стали очищать стек после системного вызова `_exit`, поскольку он всё равно не возвращает управление.

В этом примере мы не обрабатываем ошибки, предполагая, что запись в стандартный поток ввода всегда успешна (это в общем случае не так, но достаточно часто программисты на это не обращают внимания). Если бы мы хотели обрабатывать ошибки «честно», первой же командой после `int 80h` должна была бы быть команда `jc` или `jnc`, делающая условный переход в зависимости от состояния флага `CF`, в противном случае мы рискуем, что очередная команда выставит этот флаг сообразно своим результатам и признак произошедшей ошибки будет потерян. В ОС Linux с этим было несколько проще, достаточно не трогать регистр `EAX`, и ничего не потеряется.

3.6.7. Примеры системных вызовов

В вышеприведённых примерах мы рассмотрели системные вызовы `_exit` и `write`; напомним, что `_exit` имеет[44] номер `1` и принимает один параметр — код завершения, а вызов `write` имеет номер `4` и принимает три параметра, а именно номер («дескриптор») потока вывода (`1` для потока стандартного вывода), адрес области памяти, где расположены выводимые данные, и количество этих данных.

Для ввода данных (как из файлов, так и из стандартного потока ввода, т. е. «с клавиатуры») используется вызов `read`, имеющий номер `3`. Его параметры аналогичны вызову `write`: первый параметр — номер дескриптора потока ввода (для стандартного ввода используется дескриптор `0`), второй параметр — адрес области памяти, в которой необходимо разместить прочитанные данные, а третий — количество байтов, которое надлежит попытаться прочитать. Естественно, область памяти, адрес которой мы передаём вторым параметром, должна иметь размер не менее числа, передаваемого третьим параметром. **Очень важно проанализировать значение, возвращаемое вызовом `read`** (напомним, что это значение сразу после вызова содержится в регистре `EAX`). Если чтение прошло успешно, вызов вернёт строго положительное число — количество прочитанных байтов, которое, естественно, не может превышать «заказанное» через третий параметр количество, но вполне может оказаться меньше (например, мы потребовали прочитать 200 байтов, а реально было прочитано только 15). Очень важен случай, когда `read` возвращает число `0` — это свидетельствует о том, что в используемом потоке ввода возникла ситуация «конец файла». При чтении из файлов это значит, что весь файл прочитан и больше в нём данных нет. Однако «конец файла» может произойти не только при чтении из настоящего файла; так, при вводе с клавиатуры в ОС Unix

[44] Во всяком случае, в системах Linux и FreeBSD; в дальнейшем, если нет явных указаний, подразумевается, что сказанное верно как минимум для этих двух систем.

можно сымитировать ситуацию «конец файла», нажав комбинацию клавиш Ctrl-D.

Помните, что программа, в которой используется вызов `read` и не производится анализ его результата, заведомо неправильна. Действительно, мы в этом случае не можем знать, сколько первых байтов нашей области памяти содержат реально прочитанные данные, а сколько оставшихся продолжают содержать произвольный «мусор» — а значит, какая-либо осмысленная работа с этими данными невозможна.

При чтении, как и при использовании других системных вызовов, может произойти ошибка. Как мы видели, в ОС Linux это обнаруживается по «отрицательному» значению регистра `EAX` после возврата из вызова, или, если говорить точнее, по значению, которое заключено между `fffff000h` и `ffffffffh`; в ОС FreeBSD используется флаг `CF` (флаг переноса): если вызов завершился успешно, на выходе из него этот флаг будет сброшен, если же произошла ошибка, то флаг будет установлен. Это касается и вызова `read`, и рассмотренного ранее вызова `write` (мы не обрабатывали ошибочные ситуации, чтобы не усложнять наши примеры, но это не значит, что ошибки не могут произойти), и всех остальных системных вызовов.

На момент запуска программы для неё, как правило, открыты потоки ввода-вывода с номерами `0` (стандартный ввод), `1` (стандартный вывод) и `2` (поток для выдачи сообщений об ошибках), так что мы можем применять вызов `read` к дескриптору `0`, а к дескрипторам `1` и `2` — вызов `write`. Часто, однако, задача требует создания иных потоков ввода-вывода, например, для чтения и записи файлов на диске. Прежде чем мы сможем работать с файлом, его необходимо *открыть*, в результате чего у нас появится ещё один поток ввода-вывода со своим номером (дескриптором). Делается это с помощью системного вызова `open`, имеющего номер `5`. Вызов принимает три параметра. Первый параметр — адрес строки текста, задающей имя файла; имя должно заканчиваться нулевым байтом, который служит в качестве ограничителя. Второй параметр — число, задающее режим использования файла (чтение, запись и пр.); значение этого параметра формируется как битовая строка, в которой каждый бит означает определённую особенность режима, например, доступность только на запись, разрешение создать новый файл, если его нет, и т. п. К сожалению, расположение этих битов различно для ОС Linux и ОС FreeBSD; некоторые из флагов вместе с их описаниями и численными значениями приведены в таблице 3.4. Отметим, что наиболее часто встречаются два варианта для этого параметра. Первый — открытие файла только для чтения, в обеих рассматриваемых системах этот случай задаётся числом `0`. Второй случай — открытие файла на запись, при котором файл создаётся, если его не было, а если он был, то его старое содержимое теряется (в програм-

Таблица 3.4. Некоторые флаги для второго параметра вызова `open`

название	описание	значение для	
		Linux	FreeBSD
`O_RDONLY`	только чтение	000h	000h
`O_WRONLY`	только запись	001h	001h
`O_RDWR`	чтение и запись	002h	002h
`O_CREAT`	разрешить создание файла	040h	200h
`O_EXCL`	потребовать создания файла	080h	800h
`O_TRUNC`	если файл существует, уничтожить его содержимое	200h	400h
`O_APPEND`	если файл существует, дописывать в конец	400h	008h

мах на Си это задаётся комбинацией `O_WRONLY|O_CREAT|O_TRUNC`). Для Linux соответствующее числовое значение — `241h`, для FreeBSD — `601h`. Третий параметр вызова `open` используется только в случае создания файла и задаёт *права доступа* для него. Подробное описание этого параметра мы опускаем, отметим только, что в большинстве случаев его следует задать равным восьмеричному числу `0666q`.

Для вызова `open` особенно важен анализ его возвращаемого значения и проверка, не произошла ли ошибка. Вызов `open` может завершиться с ошибкой в силу массы причин, большинство из которых программист никак не может ни предотвратить, ни предсказать: например, кто-то может неожиданно стереть файл, который мы собирались открыть на чтение, или запретить нам доступ к директории, где мы намеревались создать новый файл. Итак, после выполнения вызова `open` нам необходимо проверить, не содержит ли регистр `EAX` отрицательное значение (в ОС Linux) или не взведён ли флаг `CF` (в ОС FreeBSD). Если вызов закончился успешно, то регистр `EAX` содержит ***дескриптор открытого файла*** (потока ввода или вывода). Именно этот дескриптор теперь следует использовать в качестве первого параметра в вызовах `read` и `write` для работы с файлом. Как правило, это значение следует сразу же после вызова скопировать в специально отведённую для него область памяти.

Когда все действия с файлом завершены, его следует закрыть. Это делается с помощью вызова `close`, имеющего номер `6`. Вызов принимает один параметр, равный дескриптору закрываемого файла. После этого поток ввода-вывода с таким дескриптором перестаёт существовать; последующие вызовы `open` могут снова использовать тот же номер дескриптора.

Задача в ОС Unix может узнать свой номер (так называемый идентификатор процесса) с помощью вызова `getpid`, а также номер своего «родительского процесса» (того, который создал данный процесс) с помо-

щью вызова `getppid`. Вызов `getpid` в обеих рассматриваемых системах имеет номер `20`, тогда как вызов `getppid` имеет номер `64` в ОС Linux и номер `39` в ОС FreeBSD. Оба вызова не принимают параметров; запрашиваемый номер возвращается в качестве результата работы вызова через регистр `EAX`. Отметим, что эти два вызова всегда завершаются успешно, ошибкам тут просто неоткуда взяться.

Системный вызов `kill` (номер `37`) позволяет отправить сигнал процессу с заданным номером. Вызов принимает два параметра, первый задаёт номер процесса[45], второй задаёт номер сигнала; в частности, сигнал №15 (`SIGTERM`) предписывает процессу завершиться (но процесс может этот сигнал перехватить и завершиться не сразу, либо вообще не завершаться), а сигнал №9 (`SIGKILL`) уничтожает процесс, причём этот сигнал нельзя ни перехватить, ни игнорировать.

Ядра операционных систем семейства Unix поддерживают сотни разнообразных системных вызовов; заинтересованные читатели могут найти информацию об этих вызовах в сети Интернет или в специальной литературе. Отметим, что для ознакомления с информацией о системных вызовах желательно знать язык программирования Си, да и работа на уровне системных вызовов с помощью языка Си строится гораздо проще. Более того, некоторые системные вызовы в отдельных системах могут не поддерживаться ядром, а вместо этого эмулироваться библиотечными функциями Си, что делает их использование в программах на языке ассемблера практически невозможным. В этой связи уместно будет напомнить, что язык ассемблера мы рассматриваем с учебной, а не практической целью. Программы, предназначенные для практического применения, лучше писать на Си или на других подходящих языках высокого уровня.

3.6.8. Доступ к параметрам командной строки

При работе в операционной среде ОС Unix мы, как правило, запускаем программы, указывая кроме их имён ещё и параметры — имена файлов, опции и т. п. Так, при запуске ассемблера NASM мы можем написать что-то вроде

```
nasm -f elf prog.asm
```

Слова, указанные после имени программы, называются ***параметрами командной строки***. В данном случае этих аргументов три: ключ «`-f`», слово «`elf`», обозначающее нужный нам формат результата трансляции, и имя файла «`prog.asm`». Отметим, что и само имя программы, в данном случае «`nasm`», считается элементом командной строки. Иначе говоря,

[45] На самом деле можно отправить сигнал сразу группе процессов или даже всем процессам в системе, но подробное описание этого выходит за рамки нашего курса.

командная строка представляет собой массив строк, состоящий в данном случае из четырёх элементов: «nasm», «-f», «elf» и «prog.asm».

Естественно, мы и сами можем написать программу, получающую те или иные сведения через командную строку; больше того, на Паскале мы так уже делали, пользуясь специально предназначенными для этого встроенными функциями ParamStr и ParamCount (см. т. 1, §2.9.4). Посмотрим, как то же самое выглядит на уровне машинных команд.

При запуске программы операционная система отводит в её адресном пространстве специальную область памяти, в которой располагает строки, составляющие командную строку. Информация об адресах этих строк вместе с их общим количеством для удобства помещается в стек запускаемой задачи (во всяком случае, Linux и FreeBSD поступают именно так, хотя теоретически возможны другие соглашения о передаче параметров), после чего управление передаётся нашей программе. В тот момент, когда программа начинает выполняться с метки _start, на вершине стека (то есть по адресу [esp]) располагается четырёхбайтное целое число, равное количеству элементов командной строки (включая имя программы), в следующей позиции стека (по адресу [esp+4]) находится адрес области памяти, содержащей имя, по которому программу вызвали, далее (по адресу [esp+8]) — адрес первого параметра, потом второго параметра и т. д. Каждый элемент командной строки хранится в памяти в виде строки (массива символов), ограниченной справа нулевым байтом.

Для примера рассмотрим программу, печатающую параметры своей командной строки (включая нулевой). Пользоваться средствами stud_io.inc мы уже не станем, поскольку знаем, как без них обойтись. Наша программа будет расчитана как для работы с ОС Linux, так и для работы с ОС FreeBSD. Поскольку системные вызовы в этих ОС выполняются по-разному, мы воспользуемся директивами условной компиляции для выбора того или иного текста. Эти директивы будут предполагать, что при компиляции под ОС Linux мы определяем (в командной строке NASM) макросимвол OS_LINUX, а при работе под FreeBSD — символ OS_FREEBSD. При работе под ОС Linux наш пример (назовём его cmdl.asm) нужно будет компилировать с помощью команды

```
nasm -f elf -dOS_LINUX cmdl.asm
```

а при работе под ОС FreeBSD — командой

```
nasm -f elf -dOS_FREEBSD cmdl.asm
```

Для использования вызова write нам понадобится знать длину каждой печатаемой строки, поэтому для удобства мы опишем подпрограмму strlen, получающую в качестве параметра через стек адрес строки и возвращающую через регистр EAX длину этой строки (предполагается,

что конец строки обозначен нулевым байтом). Подпрограмма будет соответствовать конвенции CDECL: для своих внутренних нужд она воспользуется регистрами `EAX` и `ECX`, которые в соответствии с CDECL имеет право испортить, регистр `EBP` она использует в качестве реперной точки стекового фрейма, как это обычно делается, и восстановит его при выходе, а остальные регистры трогать не будет.

Используя `strlen`, мы напишем подпрограмму `print_str`, которая будет получать первым и единственным параметром адрес строки, определять её длину, вызвав для этого `strlen`, и выдавать полученную строку в поток стандартного вывода с помощью системного вызова `write`. В этой подпрограмме нам дважды потребуется адрес строки — в первый раз мы его будем передавать подпрограмме `strlen`, во второй раз — системному вызову. Из стека его для этого в любом случае придётся извлечь в регистр, так что мы оставим его в регистре и второй раз к стеку обращаться не станем; но поскольку мы используем CDECL, нам необходимо предполагать, что вызываемая подпрограмма испортит `EAX`, `ECX` и `EDX`. На самом деле мы знаем, что `strlen` не портит `EDX`, но использовать это знание не станем, в противном случае возникает риск, что когда-нибудь в будущем мы изменим код `strlen`, вроде бы оставшись в рамках CDECL, но при этом `print_str` работать перестанет; поэтому при вызове подпрограмм не следует пользоваться знанием об их внутреннем устройстве, вместо этого нужно использовать общие правила. С учётом этого мы используем для хранения адреса строки регистр `EBX`, который придётся в начале подпрограммы сохранить, а в конце — восстановить; кстати, если выполнять системный вызов по правилам Linux, `EBX` всё равно придётся испортить (для FreeBSD это не так).

Кроме параметров командной строки нам придётся напечатать ещё и символы перевода строки. Здесь мы пойдём не совсем оптимальным путём с точки зрения производительности, но зато сэкономим десяток строк кода: опишем в памяти *строку*, состоящую из символа перевода строки (то есть область памяти из двух байтов, первый — символ перевода строки, имеющий код 10, второй — ограничительный ноль) и будем эту строку печатать с помощью уже имеющейся у нас подпрограммы `print_str`.

Конечно, специальная подпрограмма, вызывающая `write` для одного заранее известного байта, работала бы быстрее, ведь ей не надо было бы подсчитывать длину строки. Если же говорить об общей производительности, то нам вообще не стоило бы ради каждой строки дважды обращаться к ядру ОС, да и одного такого обращения, в принципе, слишком много; правильнее было бы сформировать в памяти один большой массив, скопировав в него содержимое всех параметров командной строки и расставив где надо символы перевода, и напечатать всё это за один системный вызов. Системные вызовы — удовольствие дорогое, ведь они требуют переключения контекста и влекут целый ряд сложных действий, выполняемых в ядре. Проблема в том, что текст программы

при такой оптимизации вырастет раз в пять и довольно серьёзно проиграет в иллюстративности.

Строку, состоящую из одного перевода строки, мы назовём `nlstr` и отведём прямо в секции `.text` — в самом её начале. Мы можем так поступить, поскольку эту область памяти наша программа не меняет; если бы это было не так, пришлось бы располагать её в секции `.data`.

Главная программа, начинающаяся с метки `_start`, расположит количество параметров командной строки в регистре `EBX`, а в регистр `ESI` поместит указатель на то место в стеке, где находится адрес очередного печатаемого параметра командной строки. Для счётчика логичнее было бы использовать `ECX`, но его могут испортить — и испортят — вызываемые подпрограммы, тогда как `EBX` согласно CDECL все обязаны восстанавливать. На каждой итерации цикла `ESI` будет увеличиваться на 4, чтобы указывать на следующую позицию в стеке, а `EBX` будет уменьшаться, чтобы показать, что печатать осталось на одну строку меньше. Полностью текст получится таким:

```
;; cmdl.asm ;;
global        _start

section        .text

nlstr        db        10, 0

strlen:          ; arg1 == address of the string
        push ebp
        mov ebp, esp
        xor eax, eax
        mov ecx, [ebp+8]  ; arg1
.lp:    cmp byte [eax+ecx], 0
        jz .quit
        inc eax
        jmp short .lp
.quit:  pop ebp
        ret

print_str:      ; arg1 == address of the string
        push ebp
        mov ebp, esp
        push ebx          ; will be spoiled
        mov ebx, [ebp+8] ; arg1
        push ebx          ;   (and in ebx, as well)
        call strlen
        add esp, 4        ; the length is now in eax
%ifdef OS_FREEBSD
        push eax          ; length
        push ebx          ; arg1
```

```
        push dword 1     ; stdout
        mov eax, 4       ; write
        push eax         ; extra dword
        int 80h
        add esp, 16
%elifdef OS_LINUX
        mov edx, eax     ; edx now contains the length
        mov ecx, ebx     ; arg1; was stored in ebx
        mov ebx, 1       ; stdout
        mov eax, 4       ; write
        int 80h
%else
%error please define either OS_FREEBSD or OS_LINUX
%endif
        pop ebx
        mov esp, ebp
        pop ebp
        ret

_start:
        mov ebx, [esp]   ; argc
        mov esi, esp
        add esi, 4       ; argv
again:  push dword [esi] ; argv[i]
        call print_str
        add esp, 4
        push dword nlstr
        call print_str
        add esp, 4
        add esi, 4
        dec ebx
        jnz again

%ifdef OS_FREEBSD
        push dword 0     ; success
        mov eax, 1       ; _exit
        push eax         ; extra dword
        int 80h
%else
        mov ebx, 0       ; success
        mov eax, 1       ; _exit
        int 80h
%endif
```

3.6.9. Пример: копирование файла

Рассмотрим ещё один пример программы, активно взаимодействующей с операционной системой. Эта программа будет получать через параметры командной строки имена двух файлов — оригинала и копии и создавать копию под заданным именем с заданного оригинала. Наша программа будет работать достаточно просто: проверив, что ей действительно передано два параметра, она попытается открыть первый файл на чтение, второй файл — на запись, и если ей это удалось, то циклически читать из первого файла данные порциями по 4096 байт, пока не возникнет ситуация «конец файла». Сразу после чтения каждой порции программа будет записывать прочитанное во второй файл. Настоящая команда `cp`, предназначенная для копирования файлов, устроена гораздо сложнее, но для учебного примера лишняя сложность не нужна.

Ясно, что нашей программе предстоит активно пользоваться системными вызовами. Дело осложняется тем, что нам хотелось бы, конечно, написать программу, которая будет успешно компилироваться и работать как под ОС Linux, так и под ОС FreeBSD. Как мы видели на примере программы из предыдущего параграфа, это требует довольно громоздкого оформления каждого системного вызова директивами условной компиляции. Предыдущий пример, содержащий всего два системных вызова, можно было написать, не особенно задумываясь над этой проблемой, что мы и сделали; иное дело — программа, в которой предполагается больше десятка обращений к операционной системе. Чтобы не загромождать исходный код однообразными, но при этом объёмными (и, значит, отвлекающими внимание) конструкциями, мы напишем один многострочный макрос, который и будет осуществлять системный вызов (точнее, *генерировать ассемблерный код* для выполнения системного вызова). В тексте этого макроса и будут заключены все различия в организации системных вызовов для Linux и FreeBSD. Макрос будет принимать на вход произвольное количество параметров, не меньшее одного; первый параметр будет задавать номер системного вызова, остальные — значения параметров системного вызова. Отметим, что под ОС Linux наш макрос откажется работать с более чем пятью параметрами, поскольку они уже не уместятся в регистры; для FreeBSD такого ограничения мы вводить не будем.

Наш макрос мы сделаем соответствующим конвенции CDECL: при его использовании нужно будет предполагать, что регистры `EAX`, `ECX` и `EDX` будут испорчены, а все остальные сохранят своё значение. Для FreeBSD соблюдение CDECL у нас получится само собой, поскольку её системные вызовы задействуют только регистр `EAX`, тогда как для Linux нам придётся принять некоторые меры. Если общее число параметров макроса больше одного (то есть системному вызову передаётся минимум

один параметр), нам нужно будет сохранить в стеке, а в конце макроса восстановить регистр `EBX`, если же общее количество параметров окажется превышающим четыре, то дополнительно мы также сохраним и восстановим `ESI` и `EDI`.

Отдельного рассмотрения заслуживает вопрос со значениями, которые *возвращает* системный вызов. Как мы уже знаем, в ОС Linux для этого задействован только регистр `EAX`, среди возможных значений которого выделен специальный диапазон для кодов ошибок, тогда как в ОС FreeBSD задействуется ещё и флаг `CF`, и если он взведён, то, следовательно, произошла ошибка и `EAX` содержит её код. Оба варианта предполагают некие трудности при обработке: под FreeBSD флаг `CF` немедленно портится, так что его нужно проверить сразу после возврата из вызова (в нашем случае это означает, что проверять его придётся в теле макроса), а для Linux приходится писать несколько громоздкую проверку на попадание числа в заданный диапазон.

Воспользовавшись тем, что регистр `ECX` всё равно (согласно конвенции CDECL) может быть испорчен, мы примем определённое соглашение, которому наш макрос будет следовать в обеих системах. Если системный вызов завершится успешно, его результат будет находиться в регистре `EAX`, а `ECX` при этом будет равен нулю; если же произойдёт ошибка, то регистр `ECX` будет содержать её код (к счастью, коды ошибок в обеих системах никогда нулю не равны), а в регистр `EAX` тогда будет занесено число `-1`. Это несколько упростит проверку успешности системного вызова в тексте программы (после вызова нашего макроса).

При передаче параметров в макрос и раскладывании их по соответствующим регистрам (в варианте для Linux) мы применим приём, который уже встречали (см. комментарий на стр. 123) — занесение всех параметров в стек с последующим их извлечением в нужные регистры. В варианте для FreeBSD никакого раскладывания по регистрам нам не требуется, зато требуется занести параметры в стек уже для использования их самим системным вызовом. Таким образом, в обоих случаях тело макроса можно начать с занесения в стек всех его параметров (в обратном порядке, чтобы не пришлось их как-либо переупорядочивать в варианте для FreeBSD). Для этого мы воспользуемся директивой `%rotate` точно так же, как при написании макроса `pcall` (см. стр. 125).

После этого в варианте для FreeBSD достаточно занести номер вызова в `EAX`, и можно отдавать управление ядру; в варианте для Linux всё не так просто, нужно ещё извлечь из стека параметры и расположить их в регистрах, причём для различного количества параметров будут задействоваться различные наборы регистров; чтобы корректно обработать всё это, нам придётся написать целый ряд вложенных друг в друга директив условной компиляции, срабатывающих в зависимости от количества переданных макросу параметров.

После возврата из системного вызова наши действия также различаются в зависимости от используемой операционной системы. В целом между двумя реализациями макроса оказывается больше различий, чем общего, так что мы их полностью разделим директивами условной компиляции, так получится понятнее. Сам макрос мы назовём `kernel` (англ. *ядро*). Возможно, логичнее было бы назвать его как-то иначе, но, например, самое естественное для этого слово — `syscall` — обозначает мнемонику машинной команды, хотя и присутствующей не на всех процессорах, совместимых с i386, но известной ассемблеру NASM, так что использовать это слово нам не следует.

Поскольку нас ожидает достаточно запутанная структура макродиректив, мы воспользуемся для них структурными отступами, а мнемоники машинных команд расположим в двух табуляциях от левого края экрана, чтобы они не смешивались с директивами. Окончательно наш макрос будет выглядеть так:

```
%macro        kernel 1-*
%ifdef OS_FREEBSD
  %rep %0
    %rotate -1
                push dword %1
  %endrep
                mov eax, [esp]
                int 80h
                jnc %%ok
                mov ecx, eax
                mov eax, -1
                jmp short %%q
  %%ok:         xor ecx, ecx
  %%q:          add esp, (%0-1)*4
%elifdef OS_LINUX
  %if %0 > 1
                push ebx
    %if %0 > 4
                push esi
                push edi
    %endif
  %endif
  %rep %0
    %rotate -1
                  push dword %1
  %endrep
                pop eax
  %if %0 > 1
                pop ebx
    %if %0 > 2
                pop ecx
```

```
        %if %0 > 3
                  pop edx
          %if %0 > 4
                  pop esi
            %if %0 > 5
                  pop edi
              %if %0 > 6
                %error "Can't do Linux syscall with 6+ params"
              %endif
            %endif
          %endif
        %endif
      %endif
    %endif
                  int 80h
                  mov ecx, eax
                  and ecx, 0fffff000h
                  cmp ecx, 0fffff000h
                  jne %%ok
                  mov ecx, eax
                  neg ecx
                  mov eax, -1
                  jmp short %%q
    %%ok:         xor ecx, ecx
    %%q:
    %if %0 > 1
      %if %0 > 4
                  pop edi
                  pop esi
      %endif
                  pop ebx
    %endif
  %else
  %error Please define either OS_LINUX or OS_FREEBSD
  %endif
  %endmacro
```

Текст макроса, конечно, получился достаточно длинным, но это компенсируется сокращением объёма основного кода. Например, рассказывая о конвенциях системных вызовов, мы привели код программы, печатающей одну строку, в варианте для Linux (стр. 142) и FreeBSD (стр. 143). С использованием макроса kernel мы можем написать так:

```
section .data
msg     db "Hello world", 10
msg_len equ $-msg
section .text
global  _start
```

```
_start: kernel 4, 1, msg, msg_len
        kernel 1, 0
```

и всё, причём эта программа будет компилироваться и правильно работать под обеими системами, нужно только не забывать указывать NASM'у флаг `-dOS_LINUX` или `-dOS_FREEBSD`.

От используемой системы в нашей программе будет зависеть ещё один момент. При открытии копируемого файла на чтение второй параметр вызова `open` должен быть равен значению `O_RDONLY`, которое в обеих рассматриваемых системах представляет собой ноль; но при открытии *целевого* файла на запись нам придётся использовать комбинацию флажков `O_WRONLY`, `O_CREAT` и `O_TRUNC`, два из которых, как это обсуждалось на стр. 145, имеют различные числовые значения в ОС Linux и ОС FreeBSD. Второй параметр системного вызова `open` при работе в ОС Linux должен в данном случае иметь значение `241h`, а в `FreeBSD` — `601h` (см. табл. 3.4). Чтобы больше не вспоминать о различиях между двумя поддерживаемыми системами, мы введём специальный символ-метку, значение которого будет зависеть от системы, для которой производится трансляция:

```
%ifdef OS_FREEBSD
openwr_flags    equ 601h
%else   ; assume it's Linux
openwr_flags    equ 241h
%endif
```

Теперь сформируем секцию переменных. Нам потребуется буфер для временного хранения данных, в который мы будем считывать очередную порцию данных из первого файла, чтобы затем записать её во второй файл. Кроме того, дескрипторы файлов мы тоже расположим в переменных. Мы могли бы использовать и регистры, но проиграли бы в наглядности. Соответствующие переменные мы назовём `fdsrc` и `fddest`. Наконец, для удобства заведём переменные для хранения количества параметров командной строки и адреса начала массива указателей на параметры командной строки, назвав эти переменные `argc` и `argvp`. Все эти переменные не требуют начальных значений и могут, следовательно, располагаться в секции `.bss`:

```
section .bss
buffer  resb    4096
bufsize equ     $-buffer
fdsrc   resd    1
fddest  resd    1
argc    resd    1
argvp   resd    1
```

При запуске нашей программы пользователь может указать неправильное количество параметров командной строки; файл, указанный в качестве источника данных, может оказаться недоступным или несуществующим; наконец, мы по каким-то причинам можем не суметь открыть на запись файл, указанный в качестве целевого. В первом случае пользователю следует объяснить, с какими параметрами нужно запускать нашу программу, в остальных двух — просто сообщить о произошедшей ошибке. Выдавать сообщения об ошибках наша программа будет в стандартный поток диагностики, имеющий дескриптор 2. Все три сообщения об ошибках мы расположим в секции `.data` в виде инициализированных переменных:

```
section .data
helpmsg db 'Usage: copy <src> <dest>', 10
helplen equ $-helpmsg
err1msg db "Couldn't open source file for reading", 10
err1len equ $-err1msg
err2msg db "Couldn't open destination file for writing", 10
err2len equ $-err1msg
```

Теперь приступим к написанию секции `.text`, то есть самой программы. Первым делом убедимся, что нам передано ровно два параметра, для чего извлечём из стека лежащее на его вершине число, обозначающее количество элементов командной строки, занесём его в переменную `argc`. Заодно на всякий случай сохраним адрес текущей вершины стека в переменной `argvp`, но извлекать из стека больше ничего не будем, так что в области стека у нас окажется массив адресов строк–элементов командной строки. Проверим, что в переменной `argc` оказалось число `3` — правильная командная строка должна в нашем случае состоять из трёх элементов: имени самой программы и двух параметров. Если количество параметров окажется неверным, напечатаем пользователю сообщение об ошибке и выйдем:

```
section .text
global  _start
_start:
        pop dword [argc]
        mov [argvp], esp
        cmp dword [argc], 3
        je .args_count_ok
        kernel 4, 2, helpmsg, helplen
        kernel 1, 1
.args_count_ok:
```

Следующим нашим действием должно стать открытие файла, имя которого задано первым параметром командной строки, на чтение. Мы

помним, что в переменной argvp находится адрес в памяти (стековой), начиная с которого располагаются адреса элементов командной строки. Извлечём адрес из argvp в регистр ESI, затем возьмём четырёхбайтное значение по адресу [esi+4] — это и будет адрес первого параметра командной строки, то есть строки, задающей имя файла, который надо читать и копировать. Для хранения адреса воспользуемся регистром EDI, после чего сделаем вызов open. Нам придётся использовать два параметра — собственно адрес имени файла и режим его использования, который будет в данном случае равен 0 (O_RDONLY). Результат работы системного вызова обязательно надо проверить. Напомним, макрос kernel устроен так, чтобы значение EAX, равное -1, указывало на ошибку, а любое другое — на успешное выполнение вызова; в применении к вызову open результат успешного выполнения — дескриптор нового потока ввода-вывода, в данном случае это поток ввода, связанный с копируемым файлом. В случае успеха сохраним полученный дескриптор в переменной fdsrc, в случае неудачи — выдадим сообщение об ошибке и выйдем.

```
        mov esi, [argvp]
        mov edi, [esi+4]
        kernel 5, edi, 0        ; O_RDONLY
        cmp eax, -1
        jne .source_open_ok
        kernel 4, 2, err1msg, err1len
        kernel 1, 2
.source_open_ok:
        mov [fdsrc], eax
```

Настало время открыть второй файл на запись. Для извлечения его имени из памяти воспользуемся точно так же регистрами ESI и EDI, после чего выполним системный вызов open, в случае ошибки выдадим сообщение и выйдем, в случае успеха сохраним дескриптор в переменной fddest. Вызов open здесь будет выглядеть несколько сложнее. Во-первых, режим открытия на запись, как обсуждалось выше, зависит от системы и будет задаваться символом-меткой openwr_flags. Во-вторых, поскольку возможно создание нового файла, наш системный вызов должен получить ещё и третий параметр, который, как мы ранее отмечали, обычно равен 666o. С учётом всего этого получится такой код:

```
        mov esi, [argvp]
        mov edi, [esi+8]
        kernel 5, edi, openwr_flags, 0666o
        cmp eax, -1
        jne .dest_open_ok
        kernel 4, 2, err2msg, err2len
```

```
        kernel 1, 3
.dest_open_ok:
        mov [fddest], eax
```

Наконец, напишем основной цикл. В нём мы будем выполнять чтение из первого файла, анализировать его результат, и если достигнут конец файла (в `EAX` значение `0`) или произошла ошибка (значение `-1`), то будем выходить из цикла, ну а если чтение прошло успешно, то нужно будет записать всё прочитанное (то есть столько байтов из области памяти `buffer`, сколько прочитал `read`; это число содержится в `EAX`) во второй файл. Поскольку `read` не может вернуть число, бóльшее своего третьего параметра (в нашем случае — `4096`), мы можем объединить ситуации ошибки и конца файла, используя условие $\texttt{EAX} \leqslant 0$.

```
.again: kernel 3, [fdsrc], buffer, bufsize
        cmp eax, 0
        jle .end_of_file
        kernel 4, [fddest], buffer, eax
        jmp .again
```

Выход из цикла мы произвели переходом на метку `.end_of_file`; рано или поздно наша программа, достигнув конца первого файла, перейдёт на эту метку, после чего нам останется только закрыть оба файла вызовом `close` и завершить программу:

```
.end_of_file:
        kernel 6, [fdsrc]
        kernel 6, [fddest]
        kernel 1, 0
```

Отметим, что все метки в основной программе, кроме метки `_start`, мы сделали локальными (их имена начинаются с точки). Так делать не обязательно, но такой подход к меткам (все метки, к которым не предполагается обращаться откуда-то издалека, делать локальными) позволяет в более крупных программах избежать проблем с конфликтами имён.

Полностью текст нашего примера вы найдёте в файле `copy.asm`.

3.7. Раздельная трансляция

Мы уже сталкивались с построением программы из отдельных модулей, когда изучали Паскаль (см. т. 1, §2.17.1). Напомним основную идею раздельной компиляции: каждый модуль компилируется отдельно, в результате компиляции получается файл в некотором промежуточном формате, потом все эти файлы связываются воедино, образуя исполняемый файл. Выигрыш получается за счёт того, что финальная сборка

исполняемого файла из отдельных модулей в промежуточном представлении происходит много быстрее (в некоторых случаях — на несколько порядков), чем компиляция отдельных модулей в это промежуточное представление; это позволяет при внесении изменений в исходные тексты программы компилировать только те модули, которые были затронуты, а для остальных использовать промежуточные файлы, полученные ранее.

Компиляторы Паскаля обычно используют некое «своё» промежуточное представление откомпилированных модулей, но для языков низкоуровневого программирования — языка ассемблера и языка Си, который мы будем изучать позже — этот формат фиксирован и называется *объектным кодом*; с этим понятием мы уже встречались (см. §3.1.4). Как мы видели, для сборки исполняемого файла используется компоновщик, он же редактор связей, он же «линкер» — программа `ld`, но до сих пор мы всегда запускали `ld` для создания исполняемого файла из *одного* (главного) модуля; в этой главе мы научимся использовать его для финальной сборки исполняемого файла из произвольного набора объектных модулей.

Как мы уже отмечали при обсуждении модулей Паскаля, очень важным свойством модуля является наличие у него собственного ***пространства имён***, позволяющего скрыть от других модулей имена, используемые нашим модулем для внутренних целей, и избежать таким образом случайных конфликтов имён. В применении к языку ассемблера это означает, что метки, введённые в модуле, будут видны только из других мест того же модуля, если только мы специально не объявим их «глобальными»; напомним, что в языке ассемблера NASM это делается директивой `global`. Часто бывает так, что модуль вводит несколько десятков, а иногда и сотен меток, но все они оказываются нужны только в нём самом, а из всей остальной программы требуются обращения лишь к одной-двум процедурам. Это практически снимает проблему конфликта имён: в разных модулях могут появляться метки с одинаковыми именами, и это никак нам не мешает, если только они не глобальные. Технически это означает, что при трансляции исходного текста модуля в объектный код все метки, кроме объявленных глобальными, исчезают, так что в объектном файле содержится только информация об именах глобальных меток.

Кроме того, сокрытие деталей реализации (так называемая *инкапсуляция*) позволяет осуществить простейшую «защиту от дурака», не давая другим программистам воспользоваться возможностями нашего модуля не так, как мы это предполагали; кроме того, локальные имена можно не вносить в техническую документацию и смело изменять, не боясь, что при этом что-то «сломается» в других модулях.

3.7.1. Поддержка модулей в NASM

Ассемблер NASM поддерживает модульное программирование, вводя для этого два основных понятия: ***глобальные метки*** и ***внешние метки***. С первыми из них мы уже знакомы: такие метки объявляются директивой `global` и, как мы уже знаем, отличаются от обычных тем, что информация о них включается в объектный файл модуля и становится видна системному редактору связей. Что касается внешних меток, то это, напротив, метки, *введения которых мы ожидаем от других модулей*. Чаще всего это просто имя подпрограммы (реже — глобальной переменной), которая описана где-то в другом модуле, но к которой нам нужно обратиться. Чтобы это стало возможным, следует сообщить ассемблеру о существовании этой метки. Действительно, ассемблер во время трансляции видит текст только одного модуля и ничего не знает о том, что в других модулях объявлены те или иные метки, и если мы попытаемся обратиться к метке из другого модуля, никак не сообщив ассемблеру о факте её существования, мы получим сообщение об ошибке. Для объявления внешних меток ассемблер NASM вводит директиву `extern`. Например, если мы пишем модуль, в котором хотим обратиться к процедуре `myproc`, а сама эта процедура описана где-то в другом месте, то чтобы сообщить об этом, следует написать:

```
extern myproc
```

Такая строка приказывает ассемблеру буквально следующее: «метка `myproc` существует, хотя её и нет в текущем модуле; встретив эту метку, просто сгенерируй соответствующий объектный код, а конкретный адрес вместо метки потом подставит редактор связей».

3.7.2. Пример

В качестве многомодульного примера мы напишем простую программу, которая спрашивает у пользователя его имя, а затем здоровается с ним по имени. Работу со строками мы на этот раз организуем так, как это обычно делается в программах на языке Си: будем использовать нулевой байт в качестве признака конца строки. С таким представлением строк мы уже сталкивались при изучении параметров командной строки (§3.6.8) и даже написали подпрограмму `strlen`, которая вычисляет длину строки; она потребуется нам и на этот раз.

Головная программа будет зависеть от двух основных подпрограмм, `putstr` и `getstr`, каждую из которых мы вынесем в отдельный модуль. Подпрограмме `putstr` потребуется посчитать длину строки, чтобы напечатать всю строку за одно обращение к операционной системе; для такого подсчёта мы используем уже знакомую нам `strlen`, которую тоже вынесем в отдельный модуль. Наконец, ещё один модуль будет

содержать подпрограмму, организующую вызов `_exit`; её мы назовём `quit`. Все модули будут называться так же, как и вынесенные в них подпрограммы: `putstr.asm`, `getstr.asm`, `strlen.asm` и `quit.asm`.

Для организации системных вызовов мы используем макрос `kernel`, который мы описали на стр. 154. Его мы также вынесем в отдельный файл, но полноценным модулем этот файл быть не сможет. Действительно, модуль — это единица трансляции, тогда как макрос, вообще говоря, не может быть ни во что оттранслирован: как мы отмечали ранее, в ходе трансляции макросы полностью исчезают и в объектном коде от них ничего не остаётся. Это и понятно, ведь макросы представляют собой набор указаний не для процессора, а для самого ассемблера, и чтобы от макроса была какая-то польза, ассемблер должен, разумеется, видеть определение макроса в том месте, где он встретит обращение к этому макросу. Поэтому файл, содержащий наш макрос `kernel`, мы будем подсоединять к другим файлам директивой `%include` на стадии макропроцессирования (в отличие от модулей, которые собираются в единое целое с помощью редактора связей существенно позже — после завершения трансляции). Этот файл мы назовём `kernel.inc`; с него мы вполне можем начать, открыв его для редактирования и набрав в нём определение макроса, которое было дано на стр. 154; ничего другого в этом файле набирать не требуется.

Следующим мы напишем файл `strlen.asm`. Он будет выглядеть так:

```
;; asmgreet/strlen.asm ;;
global  strlen

section .text
; procedure strlen
; [ebp+8] == address of the string
strlen: push ebp
        mov ebp, esp
        xor eax, eax
        mov ecx, [ebp+8]        ; arg1
.lp:    cmp byte [eax+ecx], 0
        jz .quit
        inc eax
        jmp short .lp
.quit:  pop ebp
        ret
```

Первая строчка файла указывает, что в этом модуле будет определена метка `strlen` и эту метку нужно сделать видимой из других модулей. Директивы `global` и `extern` вообще лучше для наглядности размещать в самом начале текста модуля. Подробно комментировать код процедуры не будем, поскольку он нам уже знаком.

Имея в своём распоряжении процедуру `strlen`, напишем модуль `putstr.asm`. Процедура `putstr` будет вызывать `strlen` для подсчёта длины строки, а затем обращаться к системному вызову `write`; от процедуры `print_str`, которую мы писали в примере, печатающем аргументы командной строки, новая процедура будет отличаться использованием макроса `kernel`.

```
;; asmgreet/putstr.asm ;;
%include "kernel.inc"      ; нужен макрос kernel
global  putstr             ; модуль описывает putstr
extern  strlen             ; а сам использует strlen

section        .text
; procedire putstr
; [ebp+8] = address of the string
putstr: push ebp           ; стандартное начало
        mov ebp, esp       ;   подпрограммы
        push dword [ebp+8] ; вызываем strlen для
        call strlen        ;   подсчёта длины строки
        add esp, 4         ; результат теперь в EAX
        kernel 4, 1, [ebp+8], eax  ; вызываем write
        mov esp, ebp       ; стандартное завершение
        pop ebp            ;   подпрограммы
        ret
```

Теперь настал черёд самого сложного модуля — `getstr`. Процедура `getstr` будет получать на вход адрес буфера, в котором следует разместить прочитанную строку, а также длину этого буфера, чтобы не допустить его переполнения, если пользователю придёт в голову набрать строку, которая в буфер не поместится. Для упрощения реализации мы будем считывать строку по одному символу. Конечно, в настоящих программах так не делают, поскольку системный вызов — действие достаточно дорогостоящее с точки зрения времени выполнения программы, и тратить его на один символ несколько расточительно; но наша задача сейчас не в том, чтобы получить эффективную программу, так что мы вполне можем немного облегчить себе жизнь.

Подпрограмма `getstr` будет использовать регистр `EDX` для хранения адреса текущей позиции в буфере и регистр `ECX` для хранения общего количества прочитанных символов; в начале цикла `ECX` будет увеличиваться на единицу и уже новое его значение мы будем сравнивать со значением второго аргумента нашей процедуры (то есть с размером буфера). Это позволит нам при угрозе переполнения буфера завершить выполнение процедуры, записав в конец буфера ограничительный ноль — под него в буфере ещё хватит места, поскольку записывать мы его в этом случае будем *вместо* чтения очередного символа. Регистр `EDX`, мы тоже будем увеличивать на единицу, но уже *в конце* цикла,

после считывания очередного символа и проверки, не является ли он символом конца строки. При обнаружении конца строки мы передадим управление за пределы цикла, не увеличивая EDX, так что ограничительный ноль будет записан в буфер *поверх* символа конца строки. Существует также и третий случай, в котором цикл чтения символов будет завершён — возникновение на стандартном вводе ситуации «конец файла»; при этом в очередную ячейку буфера никакой символ считан так и не будет, зато вместо него в эту ячейку будет занесён ноль.

Поскольку наша процедура будет использовать только регистры ECX, EDX и AL, конвенция CDECL окажется соблюдена без дополнительных усилий. Поскольку макрос kernel тоже написан в соответствии с CDECL, следует ожидать, что он испортит значения регистров EAX, ECX и EDX; в EAX у нас ничего долговременного не хранится, мы только используем его младший байт (AL) для краткосрочного хранения кода считанного символа, чтобы сравнить его с кодом перевода строки; а вот ECX и EDX нам придётся перед вызовом kernel сохранить в стеке, а после — восстановить.

Полностью текст модуля getstr.asm будет выглядеть так:

```
;; asmgreet/getstr.asm ;;
%include "kernel.inc"        ; нужен макрос kernel
global getstr                ; экспортируется getstr
section .text
getstr:          ; arg1 - адрес буфера, arg2 - длина
        push ebp             ; стандартное начало
        mov ebp, esp         ;          процедуры
        xor ecx, ecx         ; ECX -- счётчик считанного
        mov edx, [ebp+8]     ; EDX -- текущий адрес в буфере
.again: inc ecx              ; увеличиваем счётчик сразу же
        cmp ecx, [ebp+12]    ; и сравниваем с размером буфера
        jae .quit            ; если места нет -- выходим
        push ecx             ; сохраняем регистры ECX
        push edx             ;           и EDX
        kernel 3, 0, edx, 1  ; читаем 1 символ в буфер
        pop edx              ; восстанавливаем
        pop ecx              ;          EDX и ECX
        cmp eax, 1           ; сист. вызов вернул 1?
        jne .quit            ; если нет, выходим
        mov al, [edx]        ; код прочитанного символа
        cmp al, 10           ;   -- это код перевода строки?
        je .quit             ; если да, выходим
        inc edx              ; увеличиваем текущий адрес
        jmp .again           ; продолжаем цикл
.quit:  mov [edx], byte 0    ; заносим ограничительный 0
        mov esp, ebp         ; стандартное завершение
        pop ebp              ; процедуры
        ret
```

Напишем теперь самый простой из наших модулей — `quit.asm`:

```
;; asmgreet/quit.asm ;;
%include "kernel.inc"
global  quit
section .text
quit:   kernel 1, 0
```

Все подпрограммы готовы; приступим к написанию головного модуля, который мы назовём `greet.asm`. Поскольку все обращения к системным вызовам вынесены в подпрограммы, в головном модуле макрос `kernel` (а, значит, и включение файла `kernel.inc`) нам не понадобится. Текст выдаваемых программой сообщений мы опишем, как обычно, в виде инициализированных строк в секции `.data`; надо только не забывать, что в этой программе все строки должны иметь ограничивающий их нулевой байт. Буфер для чтения строки мы разместим в секции `.bss`. Секция `.text`, будет состоять из сплошных вызовов подпрограмм.

```
;; asmgreet/greet.asm ;;
global  _start              ; это головной модуль
extern  putstr              ; он использует подпрограммы
extern  getstr              ;   putstr, getstr и quit
extern  quit

section .data               ; описываем текст сообщений
nmq     db      'Hi, what is your name?', 10, 0
pmy     db      'Pleased to meet you, dear ', 0
exc     db      '!', 10, 0

section .bss                ; выделяем память под буфер
buf     resb    512
buflen  equ     $-buf

section .text
_start: push dword nmq      ; начало головной программы
        call putstr         ; вызываем putstr для nmq
        add esp, 4
        push dword buflen   ; вызываем getstr
        push dword buf      ;   с параметрами buf и
        call getstr         ;   buflen
        add esp, 8
        push dword pmy      ; вызываем putstr для pmy
        call putstr
        add esp, 4
        push dword buf      ; вызываем putstr для
        call putstr         ;   строки, введённой
        add esp, 4          ;   пользователем
        push dword exc      ; вызываем putstr для exc
```

```
        call putstr
        add esp, 4
        call quit           ; вызываем quit
```

Итак, в нашей рабочей директории теперь находятся файлы `kernel.inc`, `strlen.asm`, `putstr.asm`, `getstr.asm`, `quit.asm` и `greet.asm`. Чтобы получить рабочую программу, нам понадобится отдельно вызвать NASM для каждого из модулей (напомним, что `kernel.inc` модулем не является):

```
nasm -f elf -dOS_LINUX strlen.asm
nasm -f elf -dOS_LINUX putstr.asm
nasm -f elf -dOS_LINUX getstr.asm
nasm -f elf -dOS_LINUX quit.asm
nasm -f elf -dOS_LINUX greet.asm
```

Отметим, что флажок `-dOS_LINUX` необходим только для тех модулей, которые используют `kernel.inc`, так что мы могли бы при компиляции `strlen.asm` и `greet.asm` его не указывать. Однако практика показывает, что проще указывать такие флажки всегда, нежели помнить, для каких модулей они нужны, а для каких — нет.

Результатом работы NASM станут пять файлов с суффиксом «`.o`», представляющие собой *объектные модули* нашей программы. Чтобы объединить их в исполняемый файл, мы вызовем редактор связей `ld`:

```
ld greet.o strlen.o getstr.o putstr.o quit.o -o greet
```

Результатом на сей раз станет исполняемый файл `greet`, который мы, как обычно, запустим на исполнение командой `./greet`:

```
avst@host:~/work$ ./greet
Hi, what is your name?
Andrey Stolyarov
Pleased to meet you, dear Andrey Stolyarov!
avst@host:~/work$
```

3.7.3. Объектный код и машинный код

Из приведённых выше примеров видно, что каждый объектный модуль, кроме всего прочего, характеризуется списком символов (в терминах ассемблера — меток), которые он предоставляет другим модулям, а также списком символов, которые ему самому должны быть предоставлены другими модулями. Буквально переведя с английского языка названия соответствующих директив (`global` и `extern`), мы можем назвать такие символы «глобальными» и «внешними»; чаще, однако, их называют «экспортируемыми» и «импортируемыми».

Ясно, что при трансляции исходного текста ассемблер, видя обращение к внешней метке, не может заменить эту метку конкретным адресом, поскольку этот адрес ему не известен — ведь метка определена в другом модуле, которого ассемблер не видит. Всё, что может сделать ассемблер — это оставить под такой адрес свободное место в итоговом коде и записать в объектный файл информацию, которая позволит редактору связей расставить все пропущенные адреса, когда их значения уже будут известны. При ближайшем рассмотрении оказывается, что заменить метки конкретными адресами ассемблер не может не только в случае обращений к внешним меткам, но **вообще никогда**. Дело в том, что, коль скоро программа состоит из нескольких (сколько угодно) модулей, ассемблер при трансляции одного из них никак не может предугадать, каким по счёту этот модуль будет стоять в итоговой программе, какого размера будут все предшествующие модули и, следовательно, не может знать, в какой области памяти (даже виртуальной) будет располагаться код, который ассемблер сейчас генерирует.

С другой стороны, известно, что редактор связей не видит исходных текстов модулей, да и не может их видеть, поскольку предназначен для связи модулей, полученных различными компиляторами из исходных текстов на, вполне возможно, разных языках программирования. Следовательно, вся информация, необходимая для окончательного превращения объектного кода в исполняемый машинный, должна быть записана в объектный файл. Таким образом, объектный код, который получается в качестве результата ассемблирования, представляет собой некий «полуфабрикат» машинного кода, в котором вместо абсолютных (числовых) адресов находится некая информация о том, как эти адреса вычислить и в какие места кода их следует расставить.

Отметим, что информацию о символах, содержащихся в объектном файле, можно узнать с помощью программы `nm`. В качестве упражнения попробуйте применить эту программу к объектным файлам написанных вами модулей (либо модулей из приведённых выше примеров) и попытаться проинтерпретировать результаты.

3.7.4. Библиотеки

Чаще всего программы пишутся не «с абсолютного нуля», как это в большинстве примеров делали мы, а используют комплекты уже готовых подпрограмм, оформленные в виде *библиотек*. Естественно, такие подпрограммы входят в состав модулей, а сами модули удобнее иметь в заранее откомпилированном виде, чтобы не тратить время на компиляцию; разумеется, полезно иметь в доступности и исходные тексты этих модулей, но в заранее откомпилированной форме библиотеки используются чаще. Вообще говоря, различают программные библиотеки разных видов; например, бывают библиотеки макросов, которые, естественно,

не могут быть заранее откомпилированы и существуют только в виде исходных текстов. Здесь мы, однако, рассмотрим более узкое понятие, а именно то, что под термином «библиотека» понимается на уровне редактора связей.

С технической точки зрения библиотека подпрограмм — это файл, объединяющий в себе некоторое количество объектных модулей и, как правило, содержащий таблицы для ускоренного поиска имён символов в этих модулях.

Необходимо отметить одно важнейшее свойство объектных файлов: каждый из них может быть включён в итоговую программу **только целиком** либо не включён вообще. Это означает, например, что если вы объединили в одном модуле несколько подпрограмм, а кому-то потребовалась лишь одна из них, в исполняемый файл всё равно войдёт код всего модуля (то есть всех подпрограмм). Это необходимо учитывать при разбиении библиотеки на модули; так, системные библиотеки, поставляемые вместе с операционными системами, компиляторами и т. п., обычно строятся по принципу «одна функция — один модуль».

Для построения библиотеки из отдельных объектных модулей необходимо использовать специально предназначенные для этого программы. В ОС Unix соответствующая программа называется `ar`. Изначально её предназначение не ограничивалось созданием библиотек (само название `ar` означает «архиватор»), так что при вызове программы нужно указать с помощью параметра командной строки, чего мы от неё добиваемся. Например, если бы мы захотели объединить в библиотеку все модули программы `greet` (кроме, разумеется, главного модуля, который не может быть использован в других программах), это можно было бы сделать следующей командой:

```
ar crs libgreet.a strlen.o getstr.o putstr.o quit.o
```

Результатом станет файл `libgreet.a`; это и есть библиотека. После этого скомпоновать программу `greet` с помощью редактора связей можно, например, так:

```
ld greet.o libgreet.a
```

или так:

```
ld greet.o -l greet -L .
```

В отличие от монолитного объектного файла, библиотека, будучи упакована в один файл, продолжает, тем не менее, быть именно *набором объектных модулей*, из которых редактор связей выбирает только те, которые ему нужны для удовлетворения неразрешённых ссылок. Подробнее об этом мы расскажем в следующем параграфе.

3.7.5. Алгоритм работы редактора связей

Редактору связей в командной строке указывается список объектов, каждый из которых может быть либо объектным файлом, либо библиотекой, при этом объектные файлы могут быть заданы только по имени файла, тогда как библиотеки могут задаваться двумя способами: либо явным указанием имени файла, либо — с помощью флага `-l` — указанием ***имени библиотеки***, которое может упрощённо пониматься как имя файла библиотеки, от которого отброшены префикс `lib` и суффикс `.a`[46]. Так, в примере из предыдущего параграфа файл библиотеки назывался `libgreet.a`, а соответствующее *имя библиотеки* представляло собой слово `greet`. При использовании флага `-l` редактор связей пытается найти файл библиотеки с соответствующим именем в системных директориях (`/lib`, `/usr/lib` и т. п.), но можно указать ему дополнительные директории с помощью флага `-L`; так, «`-L .`» означает, что следует сначала попробовать найти библиотеку в текущей директории, и лишь затем начинать поиск в системных директориях.

В своей работе редактор связей использует два *списка символов*: список известных (разрешённых, от английского *resolved*) символов и список *неразрешённых ссылок (unresolved links)*. В первый список заносятся символы, *экспортируемые* объектными модулями (в своих текстах на языке ассемблера NASM мы помечали такие символы директивой `global`), во второй список заносятся символы, к которым уже есть обращения, то есть имеются модули, *импортирующие* эти символы (для NASM это символы, объявленные директивой `extern` и затем использованные), но которые пока не встретились ни в одном из модулей в качестве экспортируемых.

Редактор связей начинает работу, инициализировав оба списка символов как пустые, и шаг за шагом продвигается слева направо по списку объектов, указанных в его командной строке. В случае, если очередным указанным объектом будет объектный файл, редактор связей «принимает» его в формируемый исполняемый файл. При этом все символы, экспортируемые этим модулем, заносятся в список известных символов; если некоторые из них присутствовали в списке неразрешённых ссылок, они оттуда удаляются. Символы, импортируемые модулем, заносятся в список неразрешённых ссылок, если только они к этому времени не фигурируют в списке известных символов. Объектный код из модуля принимается редактором связей к последующему преобразованию в исполняемый код и вставке в исполняемый файл.

Если же очередным объектом из списка, указанного в командной строке, окажется библиотека, действия редактора связей будут более

[46] Мы здесь не рассматриваем случай так называемых разделяемых библиотек, файлы которых имеют суффикс `.so`; концепция динамической загрузки требует дополнительного обсуждения, которое выходит за рамки данного пособия.

сложными и гибкими, поскольку возможно, что принимать *все* составляющие библиотеку модули ни к чему. Прежде всего редактор связей сверится со списком неразрешённых ссылок; если этот список пуст, библиотека будет полностью проигнорирована как ненужная. Однако обычно список в такой ситуации не пуст (иначе программист не стал бы указывать библиотеку), и следующим действием редактора связей становятся поочерёдные попытки найти в библиотеке такие модули, которые экспортируют один или несколько символов с именами, фигурирующими в текущем списке неразрешённых ссылок; если такой модуль найден, редактор связей «принимает» его, соответствующим образом модифицирует списки символов и начинает рассмотрение библиотеки снова, и так до тех пор, когда ни один из оставшихся в библиотеке непринятых модулей не будет пригоден для разрешения ссылок. Тогда редактор связей прекращает рассмотрение библиотеки и переходит к следующему объекту из списка. Таким образом, из библиотеки берутся только те модули, которые нужны, чтобы удовлетворить потребности предшествующих модулей в импорте символов, плюс, возможно, те модули, в которых нуждаются уже принятые модули из той же библиотеки. Так, при сборке программы `greet` из предыдущего параграфа редактор связей сначала принял из библиотеки `libgreet.a` модули `getstr`, `putstr` и `quit`, поскольку в них присутствовали символы, импортируемые ранее принятым модулем `greet.o`; затем редактор связей принял и модуль `strlen`, поскольку в нём нуждался модуль `putstr`.

Редактор связей выдаёт сообщения об ошибках и отказывается продолжать сборку исполняемого файла в двух основных случаях. Первый из них возникает, когда список объектов (модулей и библиотек) исчерпан, а список неразрешённых ссылок не опустел, то есть как минимум один из принятых модулей ссылается в качестве внешнего на символ, который так ни в одном из модулей и не встретился; такая ошибочная ситуация называется ***неопределённой ссылкой*** (англ. *undefined reference*). Второй случай ошибочной ситуации — появление в очередном принимаемом модуле экспортируемого символа, который к этому моменту уже значится в списке известных; иначе говоря, два или более принятых к рассмотрению модуля экспортируют один и тот же символ. Это называется ***конфликтом имён***[47].

Интересно, что **редактор связей никогда не возвращается назад** в своём движении по списку объектов, так что если некоторый модуль из состава библиотеки не был принят на момент, когда редактор до этой библиотеки добрался, то потом он не будет принят тем более, даже если в каком-либо из последующих модулей появится импортиру-

[47] Современные редакторы связей в угоду нерадивым программистам позволяют не считать некоторые случаи конфликта имён ошибкой; это используется, например, компиляторами языка Си++. Постарайтесь, насколько возможно, не полагаться на подобные возможности.

емый символ, который можно было бы разрешить, приняв ещё модули из ранее обработанной библиотеки. Из этого факта вытекает важное следствие: объектные модули следует указывать **раньше**, чем библиотеки, в которых эти модули нуждаются. Вторым важным следствием является то, что **библиотеки никогда не должны перекрёстно зависеть друг от друга**, то есть если одна библиотека использует возможности второй, то вторая не должна использовать возможности первой. Если подобного рода перекрёстные зависимости возникли, такие две библиотеки следует объединить в одну.

Наконец, можно сделать ещё один вывод. До тех пор, пока библиотеки вообще не зависят друг от друга, мы можем не слишком волноваться о порядке параметров для редактора связей: достаточно сначала указать в произвольном порядке все объектные файлы, составляющие нашу программу, а затем, опять-таки в произвольном порядке, перечислить все нужные библиотеки. Если же зависимости между библиотеками появляются, порядок их указания становится важен, и при его несоблюдении программа не соберётся. Таким образом, зависимости библиотек друг от друга, даже не перекрёстные, порождают определённые проблемы. Поэтому, прежде чем полагаться при разработке одной библиотеки на возможности другой, следует многократно и тщательно всё обдумать.

Знание принципов работы редактора связей пригодится вам не только (и не столько) в учебном программировании на языке ассемблера, но и в практической работе на языках программирования высокого уровня, в особенности на языках Си и Си++. Не принимая во внимание содержание этого параграфа, вы рискуете, с одной стороны, перегрузить свои исполняемые файлы ненужным (неиспользуемым) содержимым, а с другой — спроектировать свои библиотеки так, что сами начнёте в них путаться.

3.8. Арифметика с плавающей точкой

До сих пор мы рассматривали только команды для работы с целыми числами. Между тем, ещё во введении мы рассказывали о вычислениях в *числах с плавающей точкой* (см. т. 1, §1.6.3), а при изучении Паскаля даже сами с ними работали (вспомните тип `real`). Напомним, что число с плавающей точкой обычно считается представляющим некую величину *приблизительно*, а в ходе работы при выполнении арифметических операций возникают *ошибки округления*; это неизбежная плата за представление непрерывных по своей сути величин дискретным способом.

В ранних процессорах линейки x86 (вплоть до 80386) возможности работы с числами с плавающей точкой отсутствовали; их можно было либо эмулировать программно, и работала такая эмуляция очень медленно, либо установить в компьютер дополнительную микросхему,

Таблица 3.5. Форматы чисел с плавающей точкой

Название «точности»	размер (бит)	порядок		мантисса (бит)
		размер	смещение	
обычная точность	32	8	127	23[†]
двойная точность	64	11	1023	52[†]
повышенная точность	80	15	16383	64[††]

[†] Целая часть мантиссы не хранится и подразумевается равной 1.
[††] Старший бит мантиссы считается её целой частью (обычно равной 1).

называемую ***арифметическим сопроцессором***: 8087 для 8086, 80287 для 80286, и, наконец, 80387 для 80386. Практически все компьютеры на основе 386-го процессора были оснащены сопроцессором; спроса на компьютеры без такового не было, поскольку незначительное удешевление системы не компенсировало отвратительно медленной работы машины с любыми мало-мальски заметными расчётными задачами. Поэтому при разработке очередного процессора в линейке (486DX) схемы сопроцессора были включены в одну физическую микросхему с основным процессором. Тем не менее, с точки зрения выполняющейся программы арифметический сопроцессор по-прежнему (до сих пор) представляет собой отдельный процессор со своей системой регистров, совсем не похожих на регистры основного процессора, со своими флагами, которые приходится копировать в основной регистр флагов специальными командами, и со своими своеобразными принципами функционирования.

3.8.1. Форматы чисел с плавающей точкой

Представление чисел с плавающей точкой в соответствии со стандартом IEEE-754 мы рассматривали во вводной части первого тома (см. т. 1, §1.6.3). Напомним, что такое представление состоит из трёх частей — ***знакового бита***, ***смещённого порядка*** (англ. *biased exponent*) и ***мантиссы***. На рассматриваемой нами аппаратной платформе используется три формата чисел с плавающей точкой — обычной, двойной и повышенной точности (см. табл. 3.5). Мантисса обычно удовлетворяет соотношению $1 \leqslant m < 2$, что позволяет не хранить её целую часть, подразумевая её равной 1, хотя в формате повышенной точности целая часть мантиссы всё равно хранится (под неё отводится один бит). Арифметическое значение числа с плавающей точкой определяется как

$$(-1)^s \cdot 2^{p-b} \cdot m$$

где s — знаковый бит, p — значение порядка (как беззнакового целого), b — смещение порядка для данного формата, m — мантисса.

Во всех форматах значение порядка, состоящее из одних нулей или из одних единиц, рассматривается как признак специальной формы числа. Из всех этих форм обычным числом оказывается только обыкновенный ноль, представление которого состоит из одних нулей — и в знаке, и в порядке, и в мантиссе. В «специальные случаи» ноль угодил по одной простой причине: он, очевидно, не может быть представлен числом с мантиссой, заключённой между 1 и 2. Все остальные «специальные случаи» свидетельствуют о том, что что-то пошло не так, вопрос лишь в серьёзности этого «не так».

В частности, число, знаковый бит которого установлен в единицу, а все остальные биты — и в мантиссе, и в порядке — нулевые, означает «минус ноль». Возникновение «минус нуля» в вычислениях указывает на то, что на самом деле там должно быть отрицательное число, столь малое по модулю, что его вообще никак невозможно представить хоть с одним значащим битом в мантиссе. Впрочем, такая же ситуация может возникнуть и с «плюс нулём» (слишком малое по модулю *положительное* число), но обычный ноль всё-таки может быть действительно нулём, а не результатом ошибки округления, тогда как «минус ноль» всегда являет собой результат такой ошибки. Насколько при этом всё «серьёзно», зависит от решаемой задачи. В большинстве прикладных расчётов разница между нулём и, например, числом 2^{-1024} никакого значения не имеет, столь малыми величинами оказывается возможно просто пренебречь: скажем, если вычисления показали, что автомобиль движется со скоростью 10^{-10} км/ч, то в любом разумном смысле слова следует считать, что этот автомобиль стоит на месте.

Ненулевая мантисса при нулевом порядке означает ***денормализованное число***. В этом случае подразумевается, что смещённый порядок равен своему минимально допустимому значению (-126, -1022, -16832; величина порядка без учёта смещения при этом составляет единицу), а в мантиссе целая часть равна нулю. Появление денормализованных чисел в стандарте IEEE-754 вызвало и продолжает вызывать серьёзную критику, поскольку резко усложняет реализацию процессоров и программного обеспечения, не давая при этом никакого ощутимого выигрыша. Так или иначе, сопроцессор может проводить вычисления с денормализованными числами, но если вы обнаружили денормализацию в своих расчётах, то будет правильнее, например, сменить используемые величины (например, расчёт ёмкости конденсатора проводить не в фарадах, а в пикофарадах) или каким-то ещё образом перейти к величинам, имеющим более высокие порядки.

Порядок, состоящий из одних единиц, используется для обозначения разнообразных видов «не-чисел»: плюс-бесконечности, минус-бесконечности, неопределённости, «тихого не-числа», «сигнального не-числа» и даже «неподдерживаемого числа». В нормальных расчётах ничего подобного встретиться не может; так, «бесконечности» возни-

кают, когда выполняется деление на ноль, но процессор настроен так, чтобы не инициировать при этом исключительную ситуацию (внутреннее прерывание) и не отдавать управление операционной системе. При проведении обычных расчётов процессор всегда инициирует исключение при делении на ноль и в других подобных обстоятельствах. Прикладное применение расчётов, не останавливающихся при выполнении операций над заведомо некорректными исходными данными, представляет собой отдельный достаточно нетривиальный предмет, который мы рассматривать не будем.

Поскольку сопроцессор умеет работать с вещественными числами, хранящимися в памяти в любом из трёх вышеперечисленных форматов, ассемблеру приходится поддерживать обозначения для восьмибайтных и десятибайтных областей памяти. Для обозначения таких размеров операндов в командах ассемблер NASM предусматривает ключевые слова `qword` (от слов *quadro word*, «учетверённое слово») и `tword` (от *ten word*). Есть и соответствующие псевдокоманды для описания данных (`dq` задаёт восьмибайтное значение, `dt` — десятибайтное), а также для резервирования неинициализированной памяти (`resq` резервирует заданное количество восьмибайтных элементов, `rest` — заданное количество десятибайтных).

В псевдокомандах `dw` и `dq` можно использовать наряду с обычными целочисленными константами также числа с плавающей точкой, а `dt` только их в качестве инициализаторов и допускает. NASM отличает число с плавающей точкой от целого по наличию десятичной точки; допускается использование «научной нотации», то есть, например, `1000.0` можно записать как `1.0e3`, `1.0e+3` или `1.0E3`, а `0.001` — как `1.0e-3` и т. п. NASM также поддерживает запись констант с плавающей точкой в шестнадцатеричной системе счисления, но эта возможность используется редко.

Отметим, что сам сопроцессор, если ему не мешать, все действия выполняет с числами повышенной точности, а числа других форматов использует только при загрузке и выгрузке.

3.8.2. Устройство арифметического сопроцессора

Арифметический сопроцессор имеет восемь 80-битовых регистров для хранения чисел, которые мы условно обозначим `R0`, `R1`, ..., `R7`; регистры образуют своеобразный *стек*, то есть один из регистров `R`n считается вершиной стека и обозначается `ST0`, следующий за ним обозначается `ST1` и т. д., причём считается, что следом за `R7` идёт `R0` (например, если `R7` в настоящий момент обозначен как `ST4`, то роль `ST5` будет играть регистр `R0`, `ST6` будет в `R1` и т. д.) На рис. 3.10 показана ситуация, когда вершиной стека объявлен регистр `R3`; роль вершины стека может играть любой из регистров `R`n, причём при занесении нового значения в

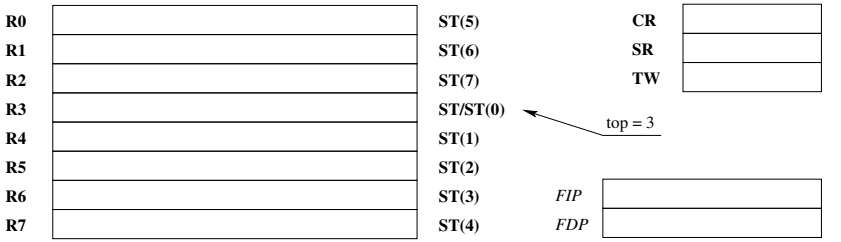

Рис. 3.10. Регистры арифметического сопроцессора

этот стек все значения, которые там уже хранились, остаются на своих местах, а меняется только номер регистра, играющего роль вершины, то есть если в стек, показанный на рисунке, внести новое значение, то роль вершины — `ST0` — перейдёт к регистру `R2`, регистр `R3` станет обозначаться `ST1`, и так далее. При удалении значения из стека происходит обратное действие. Отметим, что к этим регистрам можно обратиться только по их текущему номеру в стеке, то есть по именам `ST0`, `ST1`, ..., `ST7`. Обратиться к ним по их постоянным номерам (`R0`, `R1`, ..., `R7`) нельзя, процессор не даёт такой возможности.

Обозначения `ST0`, `ST1`, ..., `ST7` соответствуют соглашениям NASM. В других ассемблерах используются другие обозначения; в частности, MASM и некоторые другие ассемблеры обозначают регистры арифметического сопроцессора с использованием круглых скобок: `ST(0)`, `ST(1)`, ..., `ST(7)`, и именно такие обозначения чаще всего встречаются в литературе. Не удивляйтесь этому.

Регистр состояния `SR` (*state register*) содержит ряд флагов, описывающих, как следует из названия, состояние арифметического сопроцессора. В частности, биты 13-й, 12-й и 11-й (всего три бита) содержат число от 0 до 7, называемое `TOP` и показывающее, какой из регистров `R`n в настоящий момент считается вершиной стека. Флаги `C0` (бит 8), `C2` (бит 10) и `C3` (бит 14) соответствуют по смыслу флагам центрального процессора `CF`, `PF` и `ZF`. Остальные разряды регистра `ST` указывают на такие особые ситуации, как переполнение или антипереполнение стека (`SF`), потерю точности (`P`), слишком большой или слишком маленький результат последней операции (`O` и `U`), деление на ноль (`Z`) и др.

Регистр управления `CR` также состоит из отдельных флагов, но, в отличие от регистра статуса, эти флаги обычно устанавливаются программой и предназначены для *управления* сопроцессором, то есть для задания режима его работы. Например, биты 11 и 10 этого регистра задают режим округления результата операции: `00` — к ближайшему числу, `01` — в сторону уменьшения, `10` — в сторону увеличения, `11` — в сторону нуля (то есть в сторону уменьшения абсолютной величины).

Регистр тегов `TW` содержит по два бита для обозначения состояния каждого из регистров `R0`–`R7`: `00` — регистр содержит число, `01` — регистр содержит ноль, `10` — в регистре число специального вида (NAN, беско-

нечность или денормализованное число), `11` — регистр пуст. Исходно все восемь регистров помечены как пустые, по мере добавления чисел в стек соответствующие регистры помечаются как заполненные, при извлечении чисел из стека — снова как пустые. Это позволяет отслеживать переполнение и антипереполнение стека — такие ситуации, когда в стек заносится девятое по счёту число (которое некуда поместить), либо, наоборот, делается попытка извлечь число из пустого стека. Эти три регистра мы подробно рассмотрим в §3.8.9.

Служебные регистры `FIP` и `FDP` предназначены для хранения адреса и операнда последней выполняемой сопроцессором машинной команды и используются операционной системой при анализе причин возникновения ошибочной (исключительной) ситуации.

Мнемонические обозначения всех машинных команд, имеющих отношение к арифметическому сопроцессору, начинаются с буквы `f` от английского *floating* (плавающий; словосочетание «плавающая точка» по-английски звучит как *floating point*). Большинство таких команд не имеет операнда или имеет один операнд, но встречаются и команды с двумя операндами. В качестве операнда могут выступать регистры сопроцессора, обозначаемые `ST`n, либо операнды типа «память».

3.8.3. Обмен данными с сопроцессором

Команда `fld` (*float load*), имеющая один операнд, позволяет занести в регистровый стек число из заданного места, в качестве которого может выступать операнд типа «память» размера `dword`, `qword` или `tword`, либо регистр `ST`n. Например, команда

```
fld st0
```

создаёт копию вершины стека, а команда

```
fld qword [matrix+ecx*8]
```

загружает в стек из массива `matrix`, состоящего из восьмибайтовых чисел, элемент с номером, хранящимся в регистре `ECX`. При этом в регистре `SR` уменьшается значение числа `TOP`, так что вершина стека сдвигается вверх, старая вершина получает имя `ST1` и т. д.

Извлечь результат из сопроцессора (с вершины регистрового стека) можно командами `fst` и `fstp`, имеющими один операнд. Чаще всего это операнд типа «память», но можно указать и регистр из стека, например, `ST6`, важно только, что этот регистр должен быть пустым. Основное отличие между этими двумя командами в том, что `fst` просто читает число, находящееся на вершине стека (т. е. в регистре `ST0`), тогда как `fstp` *извлекает* число из стека, помечая `ST0` как свободный и увеличивая значение `TOP`. Собственно говоря, буква «`p`» в имени команды `fstp`

обозначает слово *pop*. Команда `fst` почему-то не умеет работать с 80-битными операндами типа «память», у `fstp` такого ограничения нет. Отметим ещё один момент: команда

```
fstp st0
```

сначала записывает содержимое `ST0` в него же самого, а затем выталкивает `ST0` из стека; таким образом, эффект от этой команды состоит в *уничтожении значения на вершине стека*. Так обычно делают в случае, если число, находящееся на вершине стека, в дальнейших вычислениях не нужно.

Часто бывает нужно перевести целое число в формат с плавающей точкой и наоборот. Команда `fild` позволяет взять из памяти целое число и записать его в стек сопроцессора (естественно, уже в «плавающем» формате). Команда имеет один операнд, обязательно типа «память», размера `word`, `dword` или `qword` (в этом случае имеется в виду восьмибайтное целое). Команды `fist` и `fistp` производят обратное действие: берут число, находящееся в `ST0`, округляют его до целого в соответствии с установленным режимом округления и записывают результат в память по адресу, заданному операндом. По аналогии с командами `fst` и `fstp`, команда `fist` никак не изменяет сам стек, а команда `fistp` убирает число из стека. Операнд команды `fistp` может быть размера `word`, `dword` или `qword`, команда `fist` умеет работать только с `word` и `dword`.

Команда `fxch` позволяет обменять местами содержимое вершины стека (`ST0`) и любого другого регистра `ST`n, который указывается в качестве её операнда. Регистры не должны быть пустыми. Чаще всего `fxch` используют, чтобы поменять местами `ST0` и `ST1`, в этом случае операнд можно не указывать.

Сопроцессор поддерживает ряд команд, позволяющих загрузить в стек часто употребляемые константы: `fld1` (загружает 1.0), `fldz` (загружает +0.0), `fldpi` (загружает π), `fldl2e` (загружает $\log_2 e$), `fldl2t` (загружает $\log_2 10$), `fldln2` (загружает $\ln 2$), `fldlg2` (загружает $\lg 2$). Все эти команды не имеют операндов; в результате выполнения каждой из них значение `TOP` уменьшается, и в новом регистре `ST0` оказывается соответствующее значение. От установленного режима округления зависит, в какую сторону будет отличаться загруженное приближённое значение от математического.

3.8.4. Команды арифметических действий

Простейший способ выполнения четырёх действий арифметики на сопроцессоре — это команды `fadd`, `fsub`, `fsubr`, `fmul`, `fdiv` и `fdivr` с *одним* операндом, в качестве которого может выступать операнд типа «память» размера `dword` или `qword`. Команды `fadd` и `fmul` выполняют

соответственно сложение и умножение регистра ST0 со своим операндом, команда fsub вычитает операнд из ST0, команда fdiv делит ST0 на свой операнд, fsubr, наоборот, вычитает ST0 из своего операнда, fdivr делит свой операнд на ST0; результат всех команд записывается обратно в ST0. Все шесть команд могут быть использованы и без операндов, в этом случае роль операнда играет ST1.

Все перечисленные команды имеют также форму с двумя операндами, при этом в роли обоих операндов могут выступать только регистры STn, причём одним из них обязан быть ST0 (но он может быть как первым, так и вторым операндом). В этом случае команды выполняют заданное действие над первым и вторым операндами и результат помещают в первый операнд.

Кроме того, каждая из этих команд имеет ещё и «выталкивающую» форму, которая обозначается, соответственно, faddp, fsubp, fsubrp, fmulp, fdivp и fdivrp; в этой форме команды имеют всегда два операнда-регистра STn, причём *второй* операнд должен быть ST0; после выполнения операции и занесения результата в первый операнд эти команды убирают из стека ST0, то есть он помечается как пустой и значение TOP увеличивается на единицу; вытесненное из стека число никуда не записывается. Например, «fsubp st1, st0» вычтет из значения ST1 значение ST0, результат занесёт в ST1, но после с вершины стека будет убрано значение (бывший ST0), так что вычисленное значение разности окажется как раз на (новой) вершине стека.

Команды в «выталкивающей» форме можно также записать без операндов, в этом случае в качестве операндов используются ST1 и ST0; действие в этом случае можно описать фразой «взять из стека два операнда, произвести над ними заданное действие, результат положить обратно в стек». Подчеркнём, что значение, взятое с вершины стека (из ST0), используется как *правый аргумент* выполняемой операции, а *левым аргументом* служит значение, извлечённое из стека следующим (то есть из ST1). Поскольку оба исходных значения из стека удаляются, а новое записывается, оно оказывается, естественно, в роли новой вершины стека (ST0), при этом располагается на том же месте, где до этого располагался регистр ST1.

Некоторые программисты считают достойной применения только эту форму команд. Действительно, так можно вычислить любое арифметическое выражение, если только оно не содержит слишком много вложенных скобок (иначе нам не хватит глубины стека). Для этого выражение нужно представить в так называемой польской инверсной записи (ПОЛИЗ), в которой сначала пишутся операнды, потом знак операции; операнды могут быть сколь угодно сложными выражениями, также записанными в ПОЛИЗе. Например, выражение $(x + y) * (1 - z)$ в ПОЛИЗе будет записано так: x y + 1 z - *. Пусть x, y и z у нас описаны как области памяти (переменные) длины qword и содержат

числа с плавающей точкой. Тогда для вычисления нашего выражения мы можем просто перевести запись в ПОЛИЗе в запись на языке ассемблера, при этом каждый элемент ПОЛИЗа превратится в одну команду:

```
fld     qword [x]     ; x
fld     qword [y]     ; y
faddp                 ; +
fld1                  ; 1
fld     qword [z]     ; z
fsubp                 ; -
fmulp                 ; *
```

Результат вычисления окажется в `ST0`. Впрочем, применение других форм арифметических команд способно изрядно укоротить текст программы; как несложно убедиться, следующий фрагмент делает абсолютно то же самое:

```
fld     qword [x]
fadd    qword [y]
fld1
fsub    qword [z]
fmulp
```

Иногда бывают полезны имеющие один операнд команды `fiadd`, `fisub`, `fisubr`, `fimul`, `fidiv` и `fidivr`, выполняющие соответствующее арифметическое действие над `ST0` и своим операндом, который должен быть типа «память» размера `word` или `dword` и рассматривается как **целое** число.

На всякий случай отметим, что перевод произвольного арифметического выражения из традиционной инфиксно-скобочной формы в ПОЛИЗ производится с помощью довольно простого алгоритма, известного как ***алгоритм Дейкстры***.

Для перевода в ПОЛИЗ исходное выражение просматривается слева направо, при этом по мере возможности выписываются очередные элементы ПОЛИЗа. В процессе просмотра используется вспомогательный стек, в который в некоторых случаях помещаются символы операций и *открывающие* круглые скобки. Алгоритм устроен таким образом, что операнды (константы и переменные), а также закрывающие круглые скобки в стек никогда не попадают.

В начале работы делаем текущей первую позицию исходного выражения. ПОЛИЗ считаем пустым. Стек очищаем и заносим в него открывающую круглую скобку. Теперь рассматриваем текущий элемент выражения (пока таковые есть);

- если этот элемент — операнд (переменная или константа), выписываем его в качестве очередного элемента ПОЛИЗа;
- если этот элемент — открывающая скобка, заносим ее в стек;
- если этот элемент — закрывающая фигурная скобка, извлекаем элементы из стека и выписываем их в качестве очередных элементов ПОЛИЗа до тех пор, пока на вершине стека не окажется открывающая скобка; её также извлекаем, но в ПОЛИЗ не записываем;

- наконец, если этот элемент – символ операции, то:
  - если на вершине стека находится открывающая скобка, либо если приоритет операции на вершине стека *меньше* приоритета текущей операции, заносим текущую операцию в стек;
  - если, напротив, на вершине стека находится операция, имеющая *такой же или более высокий* приоритет, чем текущая, то извлекаем операцию из стека, выписываем извлеченную из стека операцию в качестве очередного элемента ПОЛИЗа, а текущую операцию снова сравниваем (возможно, что из стека будет в итоге извлечено больше одной операции); когда операции такого же или более высокого приоритета на стеке кончились, заносим текущую операцию в стек.

Когда выражение полностью считано (больше нерассмотренных элементов нет), извлекаем элементы из стека и выписываем их в качестве очередных элементов ПОЛИЗа до тех пор, пока на вершине стека не окажется открывающая скобка. Её также извлекаем, но в ПОЛИЗ не записываем. Проверяем, что стек пуст. Если он не пуст, это свидетельствует о наличии в выражении незакрытой скобки; если же в процессе работы стек опустел раньше, чем кончилось выражение, то это, напротив, означает, что в выражении было больше закрывающих скобок, чем открывающих.

Например, трансляция выражения $a-b+c*d$ будет происходить следующим образом:

- помещаем в стек скобку;
- выписываем операнд a;
- помещаем в стек минус;
- выписываем операнд b;
- поскольку приоритет плюса не выше приоритета минуса, извлекаем из стека и выписываем минус (теперь ПОЛИЗ содержит $a\,b-$);
- поскольку на вершине стека опять скобка, заносим в стек плюс;
- выписываем операнд c (теперь ПОЛИЗ содержит $a\,b-c$);
- поскольку на вершине стека сейчас плюс, а очередной элемент выражения — знак умножения, и поскольку умножение имеет более высокий приоритет, чем сложение, заносим умножение в стек (теперь стек содержит умножение, плюс и скобку);
- выписываем операнд d, получаем ПОЛИЗ $a\,b-c\,d$;
- поскольку выражение закончилось, выбираем из стека по одному элементу и выписываем их, пока не встретим скобку: сначала умножение, потом сложение и получаем ПОЛИЗ $a\,b-c\,d*+$;
- извлекаем из стека скобку; поскольку после этого стек оказался пуст и наше выражение тоже пусто, трансляция прошла успешно.

В заключение разговора о простейшей арифметике упомянем ещё три команды. Команда `fabs` вычисляет модуль `ST0`, команда `fchs` (от слов *change sign* — сменить знак) меняет знак `ST0` на противоположный, команда `frndint` округляет `ST0` до целого в соответствии с установленным режимом округления. Результат записывается обратно в `ST0`. Все три команды имеют только одну форму — без операндов.

Команды `fprem`, `fprem1`, `fscale`, `fxtract` оставляем любознательным читателям для самостоятельного изучения.

3.8.5. Команды вычисления математических функций

Команды `fsin`, `fcos` и `fsqrt` вычисляют, соответственно, синус, косинус и квадратный корень числа, лежащего в `ST0`, результат помещается обратно в `ST0`. Команда `fsincos` чуть сложнее: она извлекает из стека число, вычисляет его синус и косинус и кладёт их в стек, так что синус оказывается в `ST1`, косинус в `ST0`, а всего в стеке оказывается на одно число больше, чем было до выполнения команды.

Несколько экзотично ведёт себя команда `fptan`, вычисляющая тангенс. Она берёт аргумент из `ST0`, вычисляет его тангенс, заносит результат обратно в `ST0`, но после этого *добавляет в стек ещё число* 1, так что в стеке оказывается на одно число больше, чем до выполнения команды, и при этом в `ST0` находится единица, а результат вычисления тангенса находится в `ST1`. Цель всех этих плясок — упрощение вычисления котангенса: его теперь можно получить уже знакомой нам командой `fdivrp`; если же котангенс не нужен, избавиться от единицы можно, разделив на неё, то есть командой `fdivp`, или просто выкинуть её из стека командой `fstp st0`.

Команда `fpatan` вычисляет $\operatorname{arctg}\frac{y}{x}$, где x — значение в `ST0`, y — значение в `ST1`. Эти два числа из стека изымаются, результат записывается в стек, так что в стеке оказывается на одно число меньше, чем было. Знак результата совпадает со знаком y, модуль результата не превосходит π.

Кроме того, сопроцессор предусматривает команды `f2xm1`, `fyl2x` и `fyl2xp1`. Команда `f2xm1` вычисляет $2^x - 1$, где x — значение `ST0`, результат заносит обратно в `ST0`. Аргумент не должен по модулю превосходить 1, иначе результат неопределён. Команды `fyl2x` и `fyl2xp1` вычисляют $y \times \log_2 x$ и $y \times log_2(x+1)$, где x — значение `ST0`, y — значение `ST1`; эти значения из стека убираются, а результат добавляется в стек, так что в итоге в стеке остаётся на одно число меньше, чем было, и на вершине находится результат вычисления. При выполнении `fyl2xp1` значение x не должно по модулю превосходить $1 + \frac{\sqrt{2}}{2}$, в противном случае результат неопределён. Читателю предлагается самостоятельно догадаться, для чего нужны эти три команды и как ими пользоваться.

Операнды у всех команд из этого параграфа не предусмотрены.

3.8.6. Сравнение и обработка его результатов

Общая идея сравнения и действий в зависимости от его результатов для чисел с плавающей точкой такая же, как и для целых: сначала производится сравнение, по итогам которого устанавливаются флаги, а затем используется команда условного перехода в зависимости от состояния флагов. Всё несколько осложняется тем, что у арифметического сопроцессора своя система флагов, причём основной процессор не имеет команд условного перехода по этим флагам. Поэтому в привычную схе-

му приходится добавить ещё и установку флагов основного процессора в соответствии с текущим состоянием флагов сопроцессора.

Сравнение можно выполнить командами `fcom`, `fcomp` и `fcompp`. Команды `fcom` и `fcomp` имеют один операнд — либо типа «память» размера `dword` или `qword`, либо регистр `ST`n; операнд можно опустить, тогда в его роли выступит `ST1`. Команды сравнивают `ST0` со своим операндом (или с `ST1`, если операнд не указан). Команда `fcomp` отличается от `fcom` тем, что выталкивает из стека `ST0`. Команда `fcompp`, не имеющая операндов, сравнивает `ST0` с `ST1` и выталкивает их оба из стека.

В результате выполнения команд сравнения устанавливаются флаги `C3` и `C0` в регистре `SR` (см. стр. 175) следующим образом: при *равенстве* сравниваемых чисел `C3` устанавливается в единицу, `C0` — сбрасывается в ноль; в противном случае `C3` сбрасывается, и если первое из сравниваемых (то есть число, находившееся в регистре `ST0`) *больше* второго (заданного операндом или регистром `ST1`), то `C0` устанавливается в единицу, если же *меньше* — то сбрасывается. Флаг `C3` оказывается, таким образом, по смыслу аналогичным флагу `ZF`, а флаг `C0` — флагу `CF` (при сравнении беззнаковых целых).

На самом деле команды сравнения устанавливают ещё и флаг `C2`, причём если всё в порядке — то он сбрасывается в ноль, если же числа *несравнимы* (например, оба числа — «плюс бесконечности», или одно из них — «не-число») и сопроцессор при этом настроен так, чтобы не инициировать прерывания в этих ситуациях — то `C2` устанавливается в единицу.

Чтобы результатом сравнения можно было воспользоваться для условного перехода, следует скопировать флаги из `CR` в регистр `FLAGS` основного процессора. Это делается командами

```
fstsw ax
sahf
```

Первая из них копирует `SR` в регистр `AX`, а вторая загружает некоторые (не все!) флаги в `FLAGS` из `AH`. В частности, после выполнения этих двух команд значение флага `C3` копируется в `ZF`, а значение `C0` — в `CF`[48], что полностью соответствует нашим потребностям: теперь мы можем воспользоваться для условного перехода любой из команд, предусмотренных для беззнаковых целых чисел: `ja`, `jb`, `jae`, `jbe`, `jna` и т. д. (см. табл. 3.3 на стр. 66). Подчеркнём ещё раз, что использование именно этих команд обусловлено только тем, что после выполнения `fstsw` и `sahf` результат сравнения оказался во флагах `CF` и `ZF`, больше ничего общего между числами с плавающей точкой и беззнаковыми целыми, вообще говоря, нет.

Пусть, например, у нас есть переменные `a`, `b` и `m` размера `qword`, содержащие числа с плавающей точкой, и мы хотим занести в `m` меньшее из `a` и `b`. Это можно сделать так:

[48] Отметим на всякий случай, что флаг `C2` при этом копируется в `PF`.

```
        fld qword [b]   ; b на вершину стека (в ST0)
        fld qword [a]   ; теперь a в ST0, b в ST1
        fcom            ; сравниваем их
        fstsw ax        ; копируем флаги в AX
        sahf            ;    и оттуда - во FLAGS
        ja lpa          ; если a>b - прыгаем
        fxcn            ; иначе меняем числа местами
lpa:              ; теперь большее в ST0, меньшее в ST1
        fstp st0        ; ликвидируем ненужное большее
        fstp qword [m]  ; записываем в память меньшее
```

«Ненужное» число можно было бы убрать из стека и иначе. Вместо предпоследней команды можно было бы дать две команды: сначала `ffree st0`, которая пометит регистр `ST0` как свободный, потом `fincstp`, которая увеличит значение `TOP` на единицу. Эти команды рассматриваются в §3.8.9.

В ряде случаев могут оказаться полезны также команды `ficom` и `ficomp`, всегда имеющие один операнд типа «память» размера `word` или `dword` и рассматривающие этот операнд как целое число. В остальном они аналогичны командам `fcom` и `fcomp`: первым операндом сравнения выступает `ST0`, по результатам сравнения устанавливаются флаги `C3`, `C2` и `C0`. Команда `ficomp`, в отличие от `ficom`, выталкивает `ST0` из стека. Наконец, команда `ftst`, не имеющая операндов, сравнивает вершину стека с нулём.

3.8.7. Исключительные ситуации и их обработка

В результате выполнения вычислений с плавающей точкой могут возникать ***исключительные ситуации***, что в некоторых случаях свидетельствует об ошибке в программе или входных данных, а в других случаях может отражать вполне штатные особенности хода вычислений. Различают шесть таких ситуаций.

1. Недопустимая операция (Invalid Operation, `#I`) может означать одно из двух: недопустимый операнд или ошибку стека. Ситуация недопустимого операнда фиксируется при попытке использования «не-чисел» в качестве операндов, извлечения квадратного корня или логарифма из отрицательного числа и т. п. Под ошибкой стека понимается попытка записать новое число в заполненный стек (то есть когда все восемь регистров заняты), либо попытка вытолкнуть число из стека, когда в стеке нет ни одного числа, либо попытка использовать в качестве операнда регистр, который в настоящее время пуст. Дополнительный флаг `SF` позволяет отличить ошибку стека от ситуации недопустимого операнда.
2. Денормализация (Denormalized, `#D`) — попытка выполнения операции над денормализованным числом, либо попытка загрузить

денормализованное число обычной или двойной (но не расширенной) точности из памяти в регистр сопроцессора.

3. Деление на ноль (Zero divider, `#Z`) — попытка деления на ноль.
4. Переполнение (Overflow, `#O`) — результат очередной операции столь велик, что не может быть представлен в виде числа с плавающей точкой имеющихся размеров (частным случаем этой ситуации является перевод числа из внутреннего десятибайтного представления в четырёх- или восьмибайтное представление с помощью, например, команды `fst` в случае, если в новое представление число «не влезает»).
5. Антипереполнение (Underflow, `#U`) — результат очередной операции столь мал по модулю, что не может быть представлен в виде числа с плавающей точкой указанного в команде размера (в том числе при выполнении команды `fst` с операндом типа «память» размера `qword` или `dword`). Отличие `#U` от `#D` (денормализации) состоит в том, что речь идёт о результате вычисления или перевода в другой формат, а не об исходно денормализованном операнде.
6. Потеря точности (Precision, `#P`) — результат операции не может быть представлен точно имеющимися средствами; в большинстве случаев это абсолютно нормально.

В каждом из регистров `CR` и `SR` младшие шесть бит соответствуют исключительным ситуациям в том порядке, в котором они перечислены: бит №0 соответствует недопустимой операции, бит №1 — денормализации, и т. д.; бит №5 соответствует потере точности. Кроме того, в регистре `SR` бит №6 устанавливается, если недопустимая операция, повлекшая установку бита №0, связана с ошибкой стека. При этом биты регистра `CR` *управляют* тем, что процессор должен сделать при возникновении исключительной ситуации. Если соответствующий бит сброшен, то при возникновении исключения будет инициировано *внутреннее прерывание* (см. §3.6.3). Если же бит установлен, исключительная ситуация считается *замаскированной* и процессор при её возникновении никаких прерываний инициировать не будет; вместо этого он постарается синтезировать, насколько это возможно, релевантный результат (например, при делении на ноль результатом будет «бесконечность» соответствующего знака; при потере точности результат округлится до числа, представимого в используемом формате, в соответствии с установленным режимом округления, и т. д.)

При возникновении любой исключительной ситуации сопроцессор устанавливает в единицу соответствующий бит (флаг) в регистре `SR`. Если ситуация не замаскирована, этот бит пригодится операционной системе в обработчике прерывания, чтобы понять, что произошло; если же ситуация замаскирована и прерывания не произойдёт, установленные флаги можно использовать в программе, чтобы отследить возникшие

исключения. Следует учитывать, что эти флаги сами не сбрасываются, их можно сбросить только явно, и это делается командой `fclex`. Команды для взаимодействия с регистрами `CR` и `SR` мы подробно рассмотрим в §3.8.9.

3.8.8. Исключения и команда `wait`

С обработкой исключительных ситуаций связана одна неочевидная особенность: инструкция, выполнение которой привело к исключению, только взводит флаг в регистре `SR`, но не инициирует внутреннее прерывание, даже если соответствующий флаг в `CR` не установлен. В таком состоянии сопроцессор остаётся до тех пор, пока не начнётся выполнение следующей команды. Проблема здесь может возникнуть в том случае, если команда, ставшая причиной исключительной ситуации, использует операнд, находящийся в памяти (например, целочисленный), при этом между этой командой и следующей за ней командой арифметического сопроцессора присутствует команда, выполняемая основным процессором, которая изменяет значение размещённого в памяти операнда. В этом случае к тому моменту, когда внутреннее прерывание всё же будет инициировано, значение, послужившее его причиной, будет уже потеряно. Например, если при выполнении последовательности инструкций

```
fimul    dword [k]
mov      [k], ecx
fsqrt
```

результатом работы `fimul` станет переполнение, за которым должно последовать внутреннее прерывание, то это прерывание произойдёт только когда сопроцессор «доберётся» до команды `fsqrt`, но к тому времени операционная система уже не сможет узнать, какое значение операнда (переменной `k`) привело к возникновению исключения. В данном примере проблема снимается изменением порядка команд:

```
fimul    dword [k]
fsqrt
mov      [k], ecx
```

Процессор поддерживает специальную команду `fwait` (или просто `wait`, это два обозначения для одной машинной команды), которая проверяет регистр статуса сопроцессора на предмет наличия незамаскированных исключительных ситуаций и при необходимости инициирует прерывание. Этой командой стоит воспользоваться, если последняя из f-команд могла стать причиной исключения, а вы при этом больше не собираетесь выполнять действий с «плавучими» числами.

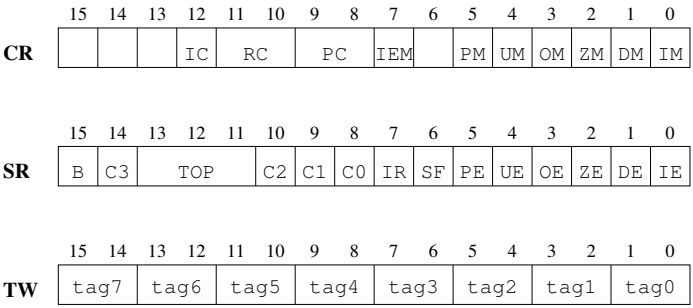

Рис. 3.11. Разряды регистров CR, SR и TW

Интересно, что некоторые мнемонические обозначения команд сопроцессора на самом деле соответствуют двум машинным командам: сначала идёт команда `wait`, затем — команда, выполняющая нужное действие. Примером такой мнемоники является уже знакомая нам `fstsw`: на самом деле, это две команды — `wait` и `fnstsw`; при необходимости можно использовать `fnstsw` отдельно, без ожидания, но для этого необходимо твёрдо понимать, что́ именно вы делаете. Точно так же устроена команда `fclex` из предыдущего параграфа: это обозначение соответствует машинным командам `wait` и `fnclex`. Команды `fnstsw` и `fnclex` представляют собой примеры команд арифметического сопроцессора, которые перед выполнением основной работы не производят проверку наличия неотработанных исключительных ситуаций.

3.8.9. Регистры управления сопроцессором

Как уже говорилось, управление режимом работы сопроцессора осуществляется установкой содержимого регистра `CR` (*Control Register*), а по результатам выполнения операций процессор устанавливает содержимое регистра `SR` (*Status Register*), которое можно проанализировать. Наконец, текущее состояние регистров, составляющих стек, отражено в регистре `TW` (*Tag Word*). Назначение разрядов, составляющих регистр управления `CR`, регистр состояния `SR` и регистр меток `TW`, показано на рис. 3.11. Бóльшая часть этих разрядов нам уже известна; так, младшие шесть бит в регистрах `CR` и `SR` представляют собой соответственно маски и флаги для шести типов исключительных ситуаций (см. §3.8.7). Биты `IC` и `IEM` регистра `CR` в современных процессорах не используются. Биты `RC` (*Rounding Control*) управляют режимом округления: `00` — к ближайшему числу, `01` — в сторону уменьшения, `10` — в сторону увеличения, `11` — в сторону нуля. Биты `PC` (*Precision Control*) задают точность выполняемых операций: `00` — 32-битные числа, `10` — 64-битные числа, `11` — 80-битные числа (по умолчанию используется именно этот режим, и необходимость его изменить возникает крайне редко).

В регистре SR флаги C3, C2 и C0 обычно используются как признак результата операции сравнения (см. §3.8.6); флаг C1 обычно не используется; флаг SF указывает на ошибку стека и позволяет понять, какого рода ошибка произошла, при констатации исключения #I (Invalid Operation): связана ли некорректная операция со стеком или нет. Флаг IR (*Interrupt Request*) указывает на возникновение незамаскированной исключительной ситуации, в результате чего инициировано внутреннее прерывание; увидеть этот флаг установленным можно только в обработчике прерывания внутри операционной системы, так что нас он не касается. Значение TOP, как уже говорилось, задаёт текущую позицию вершины стека (см. §3.8.2). Наконец, бит B (*Busy*) означает, что сопроцессор в настоящий момент занят асинхронным выполнением команды. Надо сказать, что в современных процессорах этот бит тоже невозможно увидеть установленным иначе как в обработчике прерывания.

Регистр TW мы уже рассматривали на стр. 175.

Для работы с регистром CR предусмотрены команды fstcw, fnstcw и fldcw. Команда fstcw, как обычно, означает две машинные инструкции wait и fnstcw. Все три команды имеют один операнд, в качестве которого может выступать **только операнд типа «память»** размера word. Первые две команды записывают содержимое регистра CR в заданное место в памяти, последняя команда, наоборот, загружает содержимое регистра CR из памяти. Например, следующими командами мы можем установить режим округления «в сторону нуля» вместо используемого по умолчанию режима «к ближайшему» (отметим, что в стеке мы выделим четыре байта, чтобы не нарушать его выравнивание, но использовать будем только два):

```
sub esp, 4          ; выделяем память в стеке
fstcw [esp]         ; получаем в неё содержимое CR
or word [esp], 0000110000000000b
            ; принудительно устанавливаем биты 11 и 10
fldcw [esp]         ; загружаем полученное обратно в CR
add esp, 4          ; освобождаем память
```

Содержимое регистра SR можно получить уже знакомой нам командой fstsw, операнд которой может быть либо регистром AX (и больше никаким), либо типа «память» размера word. Имеется также команда fnstsw, причём fstsw представляет собой обозначение для двух машинных инструкций wait и fnstsw. Отметим, что обратная операция (загрузка значения) для SR не предусмотрена, что вполне логично: этот регистр нужен, чтобы анализировать происходящее. Тем не менее, некоторые команды воздействуют на этот регистр напрямую. Так, значение TOP можно увеличить на единицу командой fincstp и уменьшить на единицу командой fdecstp (обе команды не имеют операндов). Использовать эти команды следует осторожно, поскольку статус «занятости»

регистров стека они не меняют; иначе говоря, `fdecstp` приводит к тому, что регистром `ST0` становится «пустой» регистр, а `fincstp` приводит к тому, что `ST7` оказывается «занят» (поскольку это бывший `ST0`). Ещё одно активное действие с регистром `SR`, которое может выполнить программист — это очистка флагов исключительных ситуаций. Такая очистка производится командами `fclex` (*Clear Exceptions*) и `fnclex`, которые мы уже упоминали в предыдущем параграфе.

Перед командой `fldcw` рекомендуется всегда выполнять команду `fclex`, иначе может случиться так, что запись регистра `CR` «демаскирует» какое-нибудь из исключений, флаг которого уже взведён, отчего произойдёт прерывание.

Регистр `TW` не может быть напрямую ни считан, ни записан, но одна команда, напрямую воздействующая на него, всё же есть. Она называется `ffree`, имеет один операнд — регистр `ST`n, а её действие — пометить заданный регистр как «свободный» (или «пустой»). В частности, следующие команды убирают число с вершины стека «в никуда»:

```
ffree st0
fincstp
```

3.8.10. Инициализация, сохранение и восстановление

Если на момент начала вычислений вам неизвестно (или вызывает сомнения) состояние арифметического сопроцессора, но при этом вы точно знаете, что никакой полезной для вас информации его регистры не содержат, можно привести его «в исходное состояние» с помощью команды `finit` или `fninit` (`finit` представляет собой обозначение для `wait fninit`, см. §3.8.8). При этом в регистр `CR` заносится значение `037Fh` (округление в ближнюю сторону, наибольшая возможная точность, все исключения замаскированы); регистр `SR` обнуляется, что означает `TOP`=0, все флаги сброшены, включая флаги исключительных ситуаций; регистры `FIP`, `FDP`, `TW` также обнуляются; регистры, составляющие стек, никак не изменяются, но поскольку `TW` обнулён, все они считаются свободными (не содержащими чисел).

С помощью команды `fsave` можно сохранить всё состояние сопроцессора, то есть содержимое всех его регистров, в области памяти, чтобы потом восстановить его. Это полезно, если нужно временно прекратить некий вычислительный процесс, выполнить какие-то вспомогательные вычисления, затем вернуться к отложенному. Для сохранения вам потребуется область памяти длиной 108 байт; команда `fsave` имеет один операнд, это операнд типа «память», причём указывать его размер не нужно. Мнемоника `fsave` на самом деле обозначает две машинные команды — `wait` и `fnsave`. После сохранения состояния в памяти сопроцессор приводится «в исходное состояние» точно так же, как при команде `finit` (см. выше), так что после `fsave` отдельно давать команду

`finit` не нужно. Восстановить сохранённое ранее состояние сопроцессора можно командой `frstor`; как и `fsave`, эта команда имеет один операнд типа «память», для которого не нужно указывать размер, поскольку используется область памяти размером 108 байт.

Иногда возникает потребность сохранить или восстановить только вспомогательные регистры сопроцессора. Это делается командами `fsetenv`, `fnsetenv` и `fldenv` с использованием области памяти длиной 28 байт; подробное описание этих команд оставляем за рамками пособия.

В завершение разговора о сопроцессоре упомянем команду `fnop`. Как можно догадаться, это очень важная команда: она *не делает ничего*.

Заключительные замечания

Конечно, мы не рассмотрели и десятой доли возможностей процессора i386, если же говорить о расширениях его возможностей, появившихся в более поздних процессорах (например, `MMX`-регистры), то доля изученного нами окажется ещё скромнее. Однако *писать программы на языке ассемблера* мы теперь можем, и это позволит нам получить опыт программирования в терминах машинных команд, что, как было сказано в предисловии, является необходимым условием качественного программирования *вообще на любом языке*: нельзя создавать хорошие программы, не понимая, что на самом деле происходит.

Читатели, у которых возникнет желание изучить аппаратную платформу i386 более глубоко, могут обратиться к технической документации и справочникам, которые в более чем достаточном количестве представлены в сети Интернет. Хочется, однако, заранее предупредить всех, у кого возникнет такое желание, что процессор i386 (отчасти «благодаря» тяжелому наследию 8086) имеет одну из самых хаотичных и нелогичных систем команд в мире; особенно это становится заметно, как только мы покидаем уютный мир ограниченного режима и «плоской» модели памяти, в котором нас заботливо устроила операционная система, и встречаемся лицом к лицу с программированием дескрипторов сегментов, нелепыми прыжками между кольцами защиты и прочими «прелестями» платформы, с которыми приходится бороться создателям современных операционных систем.

Так что если вас всерьёз заинтересовало низкоуровневое программирование, мы можем посоветовать поизучать другие архитектуры, например, процессоры SPARC или ARM. Впрочем, любопытство в любом случае не порок, и если вы готовы к определённым трудностям — то найдите любой справочник по i386 и изучайте на здоровье `:-)`

Часть 4

Программирование на языке Си

4.1. Феномен языка Си (вместо предисловия)

К настоящему моменту у вас уже есть опыт программирования на Паскале и языке ассемблера. Язык Си во многом напоминает Паскаль: во всяком случае, здесь тоже есть переменные и их типы, есть операторы, в том числе хорошо знакомые нам операторы ветвления и цикла, есть и подпрограммы, которые здесь всегда называются «функциями»; впрочем, аналоги паскалевских процедур в Си тоже присутствуют, хотя их так и не называют. Точно так же, как в Паскале, в Си активно применяется ***составной оператор***, хотя выглядит он на первый взгляд совсем иначе: вместо паскалевских слов `begin` и `end` в Си для группировки операторов используются фигурные скобки.

Интересно, что *визуально* при переходе от Паскаля к Си как раз бросается в глаза вот эта замена `begin` и `end` на скобки; находятся даже люди, которые, если их спросить, в чём разница между Паскалем и Си, тут же вспомнят именно про это, а больше толком ничего сказать не смогут; некоторые ещё добавят, что в Паскале присваивание обозначается символом `:=`, тогда как в Си — простым знаком равенства, как будто это так важно. Как правило, это означает, что минимум одного из двух языков человек не знает. Дело в том, что *на самом деле* совершенно неважно, как именно изображается та или иная сущность; суть, наоборот, в том, что и «там», и «здесь» есть присваивание и есть ***операторные скобки***, с помощью которых создаётся составной оператор. Вспомните, ведь и при обсуждении Паскаля мы всегда называли `begin` и `end` операторными скобками, так что применение в этой роли

настоящих скобок (пусть и фигурных) делает синтаксис ближе к используемой терминологии, но *по сути* ничего в этом плане не меняется. Как правило, к этой «разнице» адаптироваться проще всего. Когда визуальные различия перестают резать глаз (а происходит это почти сразу, люди вообще неплохо умеют адаптироваться), становится видна *концептуальная* разница между двумя языками, и чем дальше продвигается изучение языка, тем эта разница серьёзнее. Визуально её ухватить невозможно, необходимо понимание происходящего и определённый опыт — в частности, если вы не будете активно писать программы на Си, либо если вы в прошлом не дали себе труда плотно попрограммировать на Паскале, идеологические различия этих языков, скорее всего, пройдут мимо вас. Между тем, именно эти различия, будучи, возможно, не столь важны в процессе непосредственного написания кода (в самом деле, мы ведь пишем на каком-то одном языке, что нам за дело до другого), при этом позволяют существенно повысить свой уровень восприятия инструмента, перейти от вопроса «как устроен язык» к вопросу «почему он устроен именно так» и со временем дойти до высших уровней постижения предмета, на которых обсуждается вопрос о том, как должен быть устроен идеальный язык программирования; отметим, что, разумеется, ответа на этот вопрос не знает никто, иначе такой язык уже давно был бы создан. Впрочем, не будучи в силах ответить на вопрос, каков же идеал языка программирования, мы вполне можем, имея достаточный уровень понимания этих материй, обоснованно показать, в чём состоят *недостатки* каждого конкретного языка программирования, и вот с этим квалифицированному программисту весьма желательно уметь справляться.

Вернёмся, впрочем, с небес на землю. Язык Си может показаться странным; больше того, мы не слишком погрешим против истины, если заявим, что это так и есть — язык действительно странный. Адекватной замены этому языку нет и не предвидится, есть такие классы задач, для которых просто нет других подходящих языков; придётся, как следствие, терпеть Си таким, каков он есть. Принять этот язык как феномен вам поможет понимание того, откуда он взялся, почему он именно таков, почему, несмотря на все свои выверты и выкрутасы, этот язык продолжает удерживать первое место по популярности[1] и почему, наконец, программиста, не знающего Си, всегда и везде будут воспринимать как сотрудника второго сорта, *даже если писать на Си от него не требуется.*

История создания языка Си неразрывно связана с возникновением ОС Unix, о чём мы уже рассказывали в первом томе (см. §1.2). Первая версия этой системы была написана Кеном Томпсоном на автокоде (ассемблере) для машины PDP-7, которая к тому времени была уже

[1] По объёму программного кода, находящегося в активном использовании, Си был и остаётся бесспорным мировым лидером.

устаревшей и не представляла особого интереса; зато Кена Томпсона с его игрушками с этой машины некому было согнать. Имевшееся для этой машины системное программное обеспечение Томпсона не устроило, так что он написал операционную систему сам; так и появился Unix. В процессе дальнейшей работы Кену Томпсону понадобился язык высокого уровня, достаточно простой, чтобы его можно было реализовать быстро и без существенных трудозатрат. Для этого он придумал очень усечённую версию существовавшего ранее языка BCPL, назвав результат просто B (читается «Би»), и написал интерпретатор этого языка.

Позже Томпсон попытался переписать на Би всю свою систему, с тем чтобы её не нужно было каждый раз переписывать с нуля при переносе на другие машины. К тому времени система уже работала на популярной тогда PDP-11, которая была несовместима с PDP-7 по машинному коду, но ассемблерные мнемоники на этих компьютерах использовались почти одинаковые, так что перенести систему с одной машины на другую удалось сравнительно быстро; однако своей очереди ждали другие аппаратные архитектуры, и перспектива каждый раз переписывать всю систему Томпсону совершенно не нравилась.

Попытка использовать Би в качестве нового языка реализации ядра системы успехом не увенчалась, слишком уж примитивен был этот язык; достаточно сказать, что *типов данных в нём не было*, вся работа с данными проходила в терминах машинных слов. Положение удалось исправить Дэннису Ритчи, который вовремя присоединился к Томпсону в его экспериментах. Язык, созданный на основе Би, назвали, недолго думая, следующей буквой английского алфавита; так появился язык C (Си).

Дэнниса Ритчи часто называют автором языка Си. Это утверждение основано на том, что его соавтор по книге «Язык Си» [2] Брайан Керниган заявил в одном из своих поздних интервью, что не принимал участия в создании языка и что всё это результат работы Ритчи; но, судя по всему, правильней считать, что у языка Си было два автора: Дэннис Ритчи и Кен Томпсон; в конце концов, Си был во многом инспирирован языком Би, который придумал Томпсон, и именно Томпсон стал первым активным пользователем этого языка, переписав на нём свою операционную систему.

Во всей этой истории выделяются три основных фактора, позволяющих понять Си как явление. Во-первых, язык был создан *в качестве заменителя языка ассемблера*; во-вторых, одним из важнейших соображений при его создании была *простота реализации*; в-третьих, язык был, что называется, «слеплен» под конкретную задачу, вставшую здесь и сейчас, и делали его, как говорят в таких случаях, «на коленке». Вряд ли Дэннис Ритчи, в то время ещё очень молодой, мог предполагать, что создаваемый им язык *переживёт своего создателя* и, уже просущество-

вав больше сорока лет (а описываемые события происходили в начале 1970-х годов), всё ещё не будет выказывать никаких признаков надвигающейся старости; сейчас можно достаточно смело предсказывать, что ещё по меньшей мере полтора-два десятка лет языку Си ничего не грозит, даже если вдруг кому-то удастся создать ему полноценную замену.

С технологической точки зрения ситуация сейчас выглядит так. Имеются по меньшей мере две области задач, где единственной альтернативой Си оказывается язык ассемблера: это, во-первых, ядра операционных систем, и, во-вторых, прошивки для микроконтроллеров — специализированных компьютеров, используемых для управления техникой (лифтами, стиральными машинами и т. п.); даже билет на метро в Москве представляет собой не что иное, как компьютер на основе микроконтроллера. К вопросу о том, *почему* это так и чем здесь не устраивают другие языки, мы вернёмся позже, ближе к концу этой части; пока просто отметим, что работа на языке ассемблера оказывается в десятки раз более трудоёмкой, причём если когда-то давно ещё можно было ссылаться на более высокую эффективность ассемблерных программ, то сейчас оптимизаторы кода, встроенные прямо в компилятор, добиваются таких результатов по быстродействию, что человек, работая вручную, в большинстве случаев просто не может сделать лучше. Поэтому, когда дело доходит до низкоуровневого программирования, язык ассемблера используется лишь для тех редких и незначительных по объёму фрагментов, которые *не могут быть сделаны даже на Си*; примером такого фрагмента может служить точка входа в обработчик прерывания в ядре ОС или, например, обращение к портам ввода-вывода в драйвере, и это обычно несколько строчек. Бóльшую часть низкоуровневой программы пишут именно на Си, что позволяет экономить дорогостоящее время программистов.

В последующих параграфах мы увидим, что Си во многом нелогичен, несуразен и вообще кошмарен, его рождение в виде «наколенной поделки» даёт себя знать. Как говорят в таких случаях, *но любим мы его не за это*: писать на языке ассемблера было бы ещё хуже, а других вариантов может просто не быть. Кроме уже упоминавшихся ядер операционных систем и прошивок для микроконтроллеров, то есть программ, работающих «непосредственно на железе», которые реально больше ни на чём не написать, низкоуровневое программирование часто применяется и в других областях, как правило, ради сокращения *зависимостей от внешних условий*. Известно, что именно на чистом Си написано большое количество *переносимых программ*, таких, которые без существенных изменений запускаются и работают в совершенно разных операционных средах и на разных аппаратных платформах. Само по себе это выглядит несколько неожиданно, ведь программа на низком уровне *учитывает особенности платформы*, на то это и низкий

уровень; практика показывает, что учесть особенности разнообразных платформ в программе на Си оказывается во многих случаях проще, нежели обеспечивать единообразие работы интерпретаторов, компиляторов и библиотек, поддерживающих программирование на высоком уровне, абстрагированном от машины; сама программа, конечно, при этом может вообще никак не учитывать особенности аппаратуры, вот только библиотеки и трансляторы сами по себе оказываются программами низкого уровня, а их объём и сложность, естественно, могут быть гораздо больше, нежели объём и сложность нашей программы.

Итак, у нас *есть причины терпеть язык Си*, притом именно таким, каков он есть. Главное при этом — не забывать, что мы его именно что *терпим* и не начинать, как это часто происходит с любителями этого языка, на нём *думать*, полностью игнорируя его недостатки и даже воспринимая их как достоинства. К счастью, Си для вас будет не первым языком программирования, так что вы уже избегли части опасностей, которые содержит в себе этот язык для неокрепшего мозга начинающего программиста, но это не повод утрачивать бдительность.

Бдительности от вас потребуют также предпринятые в последние полтора десятилетия судорожные попытки *улучшить* язык Си путём принятия новых «стандартов» — сначала C99, а потом, уже совсем недавно, C11. Эти стандарты привели к появлению совершенно нелепого явления, растерявшего большинство основных привлекательных черт языка Си, но сохранившего все его недостатки. При этом комитеты по стандартизации фактически заявляют всему сообществу, что, мол, язык Си, к которому вы привыкли, отныне представляет собой не то, что было раньше, и вам всем придётся с этим смириться. Следует отметить, что так происходит не только в области языков программирования. В последние десятилетия технические стандарты во всём мире всё чаще становятся инструментом крупных корпораций по вытеснению с рынка независимых разработчиков и насаждению «решений», предлагаемых крупными игроками. В довершение картины ISO *запрещает* открытую публикацию своих стандартов и взимает плату за их тексты; если считать одним из основных предназначений технической стандартизации *совместимость* продукции различных производителей, то такая политика ISO очевидным образом напрямую противоречит целям, которым, по идее, должна соответствовать.

Нельзя сказать, что все технические стандарты одинаково плохи. Например, ранние стандарты сети Интернет, выраженные в документах серии RFC, можно приводить в пример как образец простоты, понятности и логичности. К сожалению, прекрасные традиции, заданные автором целого ряда ранних RFC Йоном Постелом, оказались во многом разрушены после его смерти в 1998 году; современные RFC часто попросту невозможно читать.

Среди удачных стандартов можно назвать также POSIX, который описывает основные средства интерфейса системных вызовов ОС Unix, а также некоторые возможности командной строки. Эта спецификация изначально создавалась по принципу *пересечения*, то есть в неё включались только такие возможности, которые уже существовали во всех основных системах семейства Unix; здоро-

вый консерватизм создателей POSIX позволяет этой спецификации сохранять популярность так же, как и самим Unix-системам.

К сожалению, исключения лишь подтверждают общее правило, и в области стандартизации языков программирования это особенно хорошо видно. Мы уже упоминали «стандартный Паскаль», который не имеет ничего общего с реальностью и нигде никогда не встречается. Этот стандарт, к счастью, не мешает программистам использовать Паскаль — на практике стандарт просто игнорируется.

Меньше всего повезло в этом плане языку Си++: с принятием первого же стандарта этот некогда уникальный по своим свойствам язык оказался фактически уничтожен, превратившись в заурядный язык высокого уровня. Довершили дело принятые подряд в 2011 и 2014 гг. ещё более извращённые спецификации, с учётом которых полученный на выходе монстр нежизнеспособен и непригоден не только к использованию, но и к изучению. В одной из следующих частей нашей книги мы рассмотрим Си++, но предметом нашего изучения станет его усечённое подмножество, в которое включены только средства, существовавшие до принятия первого стандарта, и даже эти средства мы будем рассматривать не все.

С чистым Си ситуация несколько лучше — во всяком случае, язык пока не утратил жизнеспособности, несмотря на все усилия стандартизаторов. Тем не менее, такие одиозные возможности, как массивы переменной длины (см. сноску на стр. 256), комплексные числа, строки из многобайтных символов и тому подобное ничуть не делают язык лучше: использовать эти возможности *недопустимо* в силу массы различных причин, но наличие их в языке вынуждает разумных людей тратить драгоценное время на объяснение другим причин этой недопустимости, причём объяснения часто не достигают цели.

Так или иначе, любой компилятор позволяет отключить возможности, пришедшие из нелепых стандартов, и писать на том языке Си, который существовал до того, как за него всерьёз принялись стандартизаторы. Именно так мы и намерены поступить.

4.2. Примеры программ на Си

Прежде чем приступить к рассмотрению примеров, сделаем пару важных замечаний. При работе на Паскале мы могли привыкнуть к тому, что компилятор игнорирует различие между верхним и нижним регистром. Для языка Си это не так: ключевые слова обязательно записываются в нижнем регистре, а, к примеру, слова `wordcount`, `WordCount`, `WORDCOUNT` и `WoRdCoUnT` — это четыре **разных** идентификатора. Традиции языка Си предписывают использование идентификаторов, записанных целиком в нижнем регистре; если имя состоит из нескольких слов, их разделяют подчёркиваниями, например, так: `word_count`. Из этого правила есть одно исключение, до которого мы со временем доберёмся.

И ещё одно. Для записи ***комментариев*** в Си используются комбинации «`/*`» (начало комментария) и «`*/`» (конец комментария), причём **вложенные комментарии компилятор не понимает**: если коммен-

тарий уже начался, компилятор готов обратить внимание только на «*/», а «/*» игнорирует, как и всё остальное. Как следствие, комментарии Си нельзя использовать для временного исключения из компиляции кусков кода; для этого следует пользоваться директивами условной компиляции, о которых пойдёт речь в главе о препроцессоре (см. §4.10.6).

Современные компиляторы поддерживают также «строчные» комментарии, начинающиеся с двух слэшей «//» и заканчивающиеся переводом строки. Этот стиль комментариев характерен для языка Си++ (именно там такие комментарии изначально появились); использование их в чистом Си часто рассматривается как дурной тон.

4.2.1. Программа «Hello, world»

Обучение языку Си традиционно начинают с примеров программ, пояснение к которым позволяет создать общее впечатление — на что похож этот язык и как с ним следует обращаться. Поступим так и мы, начав с традиционного[2] примера. Вот одна из самых коротких программ на Си; она просто печатает на стандартный вывод фразу `Hello, world`:

```
#include <stdio.h>

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

Оставим пока что в покое самую первую строчку программы — директиву `#include`[3] и перейдём сразу к следующей строке, которая представляет собой *заголовок функции*. С функциями мы уже встречались в Паскале; в языке Си функция представляет собой ровно то же самое: обособленный фрагмент программы, имеющий собственное имя, выполнение которого может зависеть от параметров и результатом которого будет вычисленное значение определённого типа. Кстати, в данном случае функция называется *main*, а слово `int` задаёт её тип возвращаемого значения; в языке Си этим словом обозначается тип *целое*

[2] С этого примера начиналась книга «Язык Си» Кернигана и Ритчи как в ранних, так и в более поздних её изданиях; говорят, что эту программу придумал Брайан Керниган. Отметим, что наш пример отличается от приведённого там, поскольку современные компиляторы хотя и готовы скушать тот пример в его исходном виде, всё же выдадут предупреждения в двух местах, тогда как наш вариант скомпилируется молча.

[3] С аналогичной директивой мы уже сталкивались при изучении языка ассемблера NASM; действительно, эта директива делает почти то же самое, но некоторые нюансы всё же есть, и мы поясним их чуть позже.

число. Пустые круглые скобки означают, что параметров эта функция не принимает[4].

На этом этапе у нас может возникнуть резонный вопрос, зачем потребовалась функция (то есть *подпрограмма*) в такой простой программе, выполняющей всего одно действие. Дело в том, что в языке Си нет никакого аналога паскалевской *головной программы*. Программа на Си состоит, грубо говоря, *из функций*, то есть любые операторы могут находиться только в телах функций и более нигде[5]. В этом плане функция `main` ничем принципиальным от других функций не отличается, за одним маленьким исключением: действует соглашение, что *функция с именем* `main` *используется в качестве точки входа в программу*. Иначе говоря, исполняемый файл, полученный из программы на Си, обычно[6] устроен так, что после загрузки его в оперативную память управление получит код, скомпилированный из функции `main`; с некоторой натяжкой можно считать, что функцию `main` вызывает операционная система при старте программы.

Фигурные скобки мы уже обсуждали, они играют ту же роль, что и слова `begin` и `end` в Паскале: объединяют несколько операторов в один *составной*, а также обрамляют тело подпрограммы (т. е. функции, поскольку это единственный вид подпрограмм в Си). В отличие от Паскаля, в Си операторные скобки — это действительно скобки.

Строчка

```
printf("Hello, world\n");
```

представляет собой *вызов библиотечной функции*, которая называется `printf`. Отметим один крайне важный идеологический момент. **В отличие от Паскаля, где операторы ввода-вывода являются частью языка, в язык Си как таковой никакие средства ввода-вывода не входят**. В этот язык вообще, как мы вскоре убедимся, мало что входит: в следующих параграфах мы перечислим встроенные типы переменных, арифметические операции, из которых строятся выражения, и, наконец, операторы, то есть управляющие конструкции вроде ветвлений и циклов (которых в Си **всего одиннадцать**), и в итоге практически все средства языка Си исчерпаем.

---

[4]Если следовать букве стандарта, пустые скобки означают, что функция, возможно, принимает произвольное количество параметров, но все их благополучно игнорирует; впрочем, можете не обращать на это внимания.

[5]Вне функций могут располагаться описания типов, глобальных переменных и т.п., но они операторами не являются и никаких действий не выполняют; действовать могут только функции.

[6]Именно «обычно», поскольку это не всегда так; к этому вопросу мы вернёмся в самом конце части, посвящённой Си, и заодно узнаем, каким образом на самом деле функции `main` отдаётся управление.

Функция `printf` **не является частью языка Си** по меньшей мере в том смысле, что компилятор о ней ничего не знает[7]. Более того, **сама функция** `printf` **написана на Си**.

Мы использовали функцию `printf`, чтобы напечатать строку, но её возможности гораздо шире: она умеет печатать значения всех встроенных типов языка Си, причём целые числа можно печатать в разных системах счисления, можно управлять количеством печатаемых знаков и т. п. Буква `f` в названии функции происходит от слова *formatted*, то есть эта функция осуществляет ***форматированный вывод***. Основные возможности `printf` мы рассмотрим чуть позже.

Отметим ещё один момент. Функция `printf` — это действительно *функция*; она возвращает целое число, равное количеству напечатанных символов, то есть мы могли бы, например, написать

```
x = printf("Hello, world\n");
```

и в переменную `x` оказалось бы занесено число `13`; но нам это не нужно, поэтому мы просто игнорируем возвращённое функцией значение. В таких случаях говорят, что *функция вызвана ради побочного эффекта*, или, в более общем случае, что ради побочного эффекта было вычислено арифметическое выражение. При работе на языке Си (опять же, в отличие от Паскаля) это происходит очень часто[8].

Отдельного рассмотрения заслуживает выражение `"Hello, world\n"`, которое представляет собой ***строковую константу*** или, как часто говорят, ***строковый литерал***. Следует обратить внимание на *двойные* кавычки; апострофы в Си тоже используются, но для других целей. Кроме того, внимание привлекает конструкция `\n`; это обозначение ***символа перевода строки***, то есть символа с кодом 10. Можно вспомнить, что в Паскале мы в подобной ситуации применяли оператор `writeln`:

```
writeln('Hello, world');
```

то есть заставляли нашу программу сначала напечатать данную строку, а затем перевести строку на печати, то есть фактически напечатать ещё и символ перевода строки. В вышеприведённом примере мы просто включили этот символ в состав печатаемой строки.

Отметим, что мы могли бы и в Паскале поступить точно так же; сделать это можно одним из следующих способов:

```
write('Hello, world'#10);
write('Hello, world'^J);
```

[7] А если и знает, то обязан делать вид, что не знает.

[8] Как мы увидим чуть позже, практически любая программа на Си по меньшей мере на две трети, если не больше, состоит из *операторов вычисления выражения ради побочного эффекта*.

или даже вот так:

```
s := 'Hello, world'^J;
write(s);
```

Надо отметить, однако, что обозначение \n, в котором буква n взята от слова *new [line]*, гораздо проще запомнить, нежели код символа и тем более загадочное ^J.

Прежде чем закончить разбор этой строки программы, обратим внимание на символ точки с запятой. В принципе, роль этого символа в языке Си подобна паскалевской, но есть одно важное отличие: **если в Паскале точка с запятой *разделяла операторы*, то в Си точка с запятой *является частью синтаксиса оператора***, так что, в частности, её наличие никак не зависит от положения оператора непосредственно перед закрывающей фигурной скобкой.

Рассмотрим теперь следующую строку, «`return 0;`». Слово `return` в переводе с английского означает «возврат»; в языке Си это ***оператор возврата из функции***. Он делает две вещи: во-первых, работа функции на этом заканчивается, что можно, например, использовать для досрочного её завершения (подобно паскалевскому `exit`); во-вторых, его параметр — выражение, написанное после слова `return`, в данном случае `0` — задаёт то значение, которое вернёт функция (вспомним, что наша функция `main` описана как возвращающая целое число). В данном случае наша функция `main` вернёт число `0`. Здесь может естественным образом возникнуть вопрос, *кому и зачем* нужно это число, ведь функцию `main` в нашей программе вроде бы никто не вызывает и её результат не анализирует. Краткий ответ состоит в том, что значение, возвращаемое из `main`, предназначается операционной системе, а `0` в данном случае означает, что всё в порядке, т. е. программа, завершаясь, считает, что возложенную на неё миссию успешно выполнила. Вспомнив системный вызов `_exit`, который мы рассматривали при изучении языка ассемблера, мы сможем выразиться более точно: значение, которое возвращается из функции `main`, служит аргументом вызова `_exit`, то есть кодом завершения программы. Мы вернёмся к этому вопросу в следующем параграфе.

Нам осталось рассмотреть, пожалуй, самую заковыристую строку в программе: директиву `#include <stdio.h>`. Как уже, несомненно, догадался читатель, эта директива в тексте программы *заменяет сама себя на полное содержимое файла* `stdio.h`, а нужно это, чтобы компилятор узнал про функцию `printf` — как уже говорилось, сам по себе он о ней не знает. Здесь следует отметить сразу несколько интересных моментов. Во-первых, обращают на себя внимание используемые угловые скобки, которые означают, что включаемый файл следует искать в системных директориях; точнее говоря, считается, что именно так

следует включать любой файл, который не является сам по себе частью нашей программы; когда же мы пишем программу, состоящую из многих файлов, то для включения своих собственных файлов мы используем `#include` с параметром в двойных кавычках, например:

```
#include "mymodule.h"
```

Файл `stdio.h` — это вполне реальный файл; скорее всего, он находится в вашей системе в директории `/usr/include`, так что вы можете просмотреть его, например, командой `less`:

```
less /usr/include/stdio.h
```

Важно понимать, что в этом файле содержится только ***заголовок*** функции `printf`, а самой функции там нет; отсюда используемый суффикс «`.h`», от слова *header*, то есть «заголовок» — в английской терминологии это называется *header file*, а по-русски — «заголовочный файл». Всё, что компилятор узнает, увидев заголовок — это что *где-то* (и притом совершенно неважно, где) есть функция с таким-то именем, принимающая столько-то параметров таких-то типов; как функция выглядит, что она делает — этого компилятору знать не нужно. Дело в том, что компилятор генерирует, как мы это уже выяснили при изучении ассемблера, не готовый ***машинный код***, а ***объектный модуль***, в котором пока что не хватает некоторых адресов; в данном случае результатом работы компилятора становится модуль, **не содержащий** самой функции `printf`. Вместо неё модуль содержит указание на то, что редактору связей (линкеру) следует *откуда-то* добыть функцию с таким именем, а её адрес подставить куда следует. Иначе говоря, реальная функция `printf` появится в нашей программе только на этапе окончательной сборки, когда работа компилятора будет завершена[9].

Обсудив всё это, отметим одну ошибку, столь же грубую, сколь часто встречающуюся, которую делают, к сожалению, не только начинающие программисты, но и некоторые преподаватели. На вопрос о том, что́ же делает директива `#include`, можно очень часто услышать ответ, что-де она «подключает библиотеку», а сам файл `stdio.h` — это якобы, соответственно, библиотека. Так вот, стоит осознать раз и навсегда, что **директива `#include` не имеет никакого отношения к подключению библиотек**, а заголовочные файлы, разумеется, библиотеками не являются — они являются именно заголовочными файлами и ничем иным. Язык Си вообще не предусматривает возможности *подключить библиотеку* из текста программы, потому что компилятор, попросту говоря, этим не занимается, это не его дело. Заявлять противоположное —

[9] Больше того, в реальной ситуации, скорее всего, функция printf вообще не будет содержаться в исполняемом файле нашей программы; после её запуска функция будет подгружена из динамической библиотеки.

это грубейшая ошибка, проявление вопиющего невежества. Кстати, в следующем параграфе мы будем рассматривать пример, который реально потребует подключения библиотеки, заодно и увидим, как на самом деле это делается.

Теперь, когда мы закончили разбор программы «Hello, world», самое время попробовать её запустить. Итак, включаем компьютер, входим в систему, запускаем текстовый редактор; учтите, что ваш файл должен иметь суффикс «`.c`», поскольку вы собираетесь в нём набрать программу на Си. Например, имя `hello.c` для вашего файла вполне подойдёт. Набираем ровно такой текст программы, как показано на стр. 196, и сохраняем его. Теперь нам нужно запустить компилятор, который в данном случае называется `gcc`[10]. Это название образовано от слов *Gnu Compiler Collection*, то есть «коллекция компиляторов Gnu»; компилятор умеет обрабатывать программы не только на Си, но и на языке Си++, а при использовании дополнений, так называемых фронт-эндов, тот же компилятор можно применить для Objective-C, Фортрана, Ады и многих других языков.

Сразу же отметим, что компилятор `gcc` поддерживает множество разнообразных флажков командной строки, из которых нам потребуются по меньшей мере два: `-Wall`, который включает *все разумные предупреждения*, и `-g`, который заставляет компилятор сгенерировать отладочную информацию. Эти флажки мы при запуске `gcc` указываем **всегда**, то есть вообще всегда, следует выработать привычку к этому на уровне, как говорят, спинного мозга: отсутствие любого из этих флажков может очень дорого обойтись. Не указывать эти флаги можно разве что тогда, когда компилятор запускает не программист, а конечный пользователь, чтобы из присланных ему исходных текстов получить исполняемую программу. В этом случае предполагается, что программа уже полностью готова и отлажена, а даже если она и содержит ошибки, то конечный пользователь всё равно не станет их исправлять. Но это ситуация не наша: мы пользуемся компилятором как инструментом программиста.

Ещё один полезный флаг, `-o`, позволяет задать имя файла, в который будет записан результат компиляции. Если этот флажок не указать, итоговый исполняемый файл будет иметь имя `a.out`, что не всегда удобно. Как мы помним, обычно исполняемые файлы в ОС Unix не имеют суффикса. Именно так мы поступим и сейчас — отправим результат компиляции в файл `hello`:

```
avst@host:~/work$ gcc -Wall -g hello.c -o hello
avst@host:~/work$
```

[10]На самом деле в системах семейства Unix доступно много разных компиляторов Си, и возможно даже, что gcc из них сейчас не самый лучший; на момент написания этого текста gcc был просто наиболее распространённым вариантом. Если очень любопытно, попробуйте самостоятельно освоить компилятор `clang`.

Полностью эта команда означает следующее: «Возьми исходный файл `hello.c`, откомпилируй его, выдавая при этом все разумные предупреждения, добавь в полученный модуль отладочную информацию, потом вызови редактор связей, скормив ему наш модуль и *стандартную библиотеку языка Си* (!), а результат пусть он запишет в файл `hello`». Промежуточный объектный файл нам в этот раз не нужен, поскольку наша программа состоит из одного модуля; компилятор поместит объектный код во временный файл, который по завершении работы сотрёт.

Если всё сделать правильно, компилятор отработает полностью молча, не выдав ни слова, а в текущей директории появится файл `hello`, который можно будет запустить:

```
avst@host:~/work$ ./hello
Hello, world
avst@host:~/work$
```

4.2.2. О завершении программы

Как мы отмечали выше, с некоторой небольшой натяжкой можно считать, что *в роли вызывающего для функции main выступает операционная система*, и именно операционной системе предназначено число, которое эта функция возвращает. Более строго это значение называется *кодом завершения программы*. Этот код используется, чтобы показать операционной системе, считает ли завершающаяся программа, что её выполнение прошло успешно. Если, по мнению самой программы, всё было хорошо и ей удалось выполнить ту задачу, ради которой её запускали, она завершается с кодом `0`, как в примере из предыдущего параграфа; если что-то пошло не так, например, она не смогла открыть нужный ей файл, не смогла установить связь с сервером или эта связь неожиданно разорвалась — мало ли ошибок происходит при работе программ — то, чтобы оповестить о неудаче операционную систему, программа завершается с кодом `1`, `2` и так далее. Максимально возможное значение такого кода — 255, поскольку он восьмибитный, но большие значения обычно не используются, в реальной жизни код завершения редко превышает 10.

Между прочим, на этом основаны две *самые короткие* программы на языке Си, которые называются `true` и `false`: обе они ничего не делают, завершаясь немедленно после запуска, при этом первая завершается успешно, а вторая — неуспешно. Самая простая реализация программы `true` будет такой:

```
int main() { return 0; }
```

Программа `false` выглядит почти так же:

```
int main() { return 1; }
```

Подчеркнём, что в обоих случаях написанная строчка — это **текст программы целиком**, никаких директив `#include` и чего-либо ещё не требуется. Эти две программы часто применяются при написании скриптов на командно-скриптовых языках, вроде рассмотренного нами ранее Bourne Shell.

Следует сразу же предостеречь читателя от довольно частого варианта ошибочного понимания происходящего. Поскольку все наши программы пока что состояли из одной функции `main`, оператор `return` в них завершал программу; но на самом деле этот оператор завершает выполнение **одной функции**, и если это будет не `main`, а какая-то другая функция, то, очевидно, `return` к завершению программы не приведёт. Больше того, можно вызвать и саму функцию `main` из какой-нибудь другой функции или даже из неё самой, сделав её рекурсивной; обычно так не делают, но в теории это не запрещено, и тогда даже в функции `main` выполнение `return` не станет завершением программы. В наших примерах `return` завершал программу лишь постольку, поскольку он завершал работу того вызова `main`, с которого работа программы началась.

Читатель может вспомнить, что с кодом завершения задачи мы уже дважды встречались ранее. Программы, написанные на Паскале, мы досрочно завершали с помощью `halt` и отмечали (см. т. 1, стр. 266), что при этом возможно указание необязательного целочисленного параметра, который, естественно, представлял собой всё тот же код завершения. Программируя на языке ассемблера, мы рассматривали системный вызов `_exit` (см. стр. 141), которому соответствующий параметр требуется обязательно. Здесь можно заметить, что и с помощью `_exit`, и с помощью паскалевского `halt` программу можно немедленно завершить, находясь в любом её месте, а не только в головной её части. Такая возможность есть и в языке Си, точнее говоря, в его стандартной библиотеке: для этого можно воспользоваться библиотечной функцией `exit`, имеющей один целочисленный параметр. Прототип этой функции описан в заголовочном файле `stdlib.h`; подключив этот файл с помощью `#include`, мы сможем в любой из функций нашей программы написать что-то вроде

```
exit(0);
```

— и выполнение такого оператора программу немедленно завершит. Вместо `0` можно подставить любой другой код, следует только помнить, что для него допустимы значения от `0` до `255`, но обычно используется число не больше 10.

4.2.3. Квадратное уравнение

Вторая программа, которую мы рассмотрим, будет решать квадратное уравнение[11]. Несмотря на простоту задачи, при её решении мы узнаем очень много нового.

Чтобы достичь поставленной цели, нам потребуются переменные, способные хранить число с плавающей точкой; в языке Си для этого лучше всего подходит тип `double`[12]. Вычисление дискриминанта мы вынесем в отдельную функцию, заодно увидим, как выглядит функция с параметрами. А вот дальше нас ждут настоящие приключения. Во-первых, нам придётся узнать, как печатать числа (выдавать их представление в поток стандартного вывода) с помощью уже знакомой нам функции `printf` и как читать числа из потока стандартного ввода (с клавиатуры) с помощью другой функции, которая называется `scanf`. Во-вторых, мы обнаружим, что все параметры в Си передаются исключительно как значения, то есть никакого аналога var-параметрам («параметрам-переменным») Паскаля здесь нет; между тем, как-то нужно объяснить `scanf`'у, куда (то есть в какие переменные) девать свежепрочитанные значения. Победив коварный ввод-вывод, мы, возможно, вздохнём с облегчением, но ненадолго: дело в том, что функции, работающие с плавающей точкой, не входят в основную библиотеку, которую компилятор подключает по умолчанию, и к функции вычисления квадратного корня сие тоже относится; так что, прежде чем программа заработает, нам придётся ещё выяснить, как же **в действительности** подключаются библиотеки (см. замечание на стр. 200).

Начнём с замечания, что функция вычисления квадратного корня называется `sqrt`, а её заголовок, наряду с заголовками других математических функций (таких как синус, логарифм, экспонента и т. п.), располагается в заголовочном файле `math.h`. Кроме того, поскольку мы собираемся использовать функции ввода-вывода, нам, как и в предыдущей программе, потребуется заголовочник `stdio.h`. Традиционно директивы `#include` располагают в начале программы, поступим так и мы; открываем в редакторе текстов новый файл (например, `qe.c` от слов *quadratic equation*, то есть «квадратное уравнение») и пишем для начала следующие две строки:

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

Вспомним теперь, что мы хотели написать отдельную функцию для вычисления дискриминанта. Для этого, как известно, нужны три

[11] Как известно, квадратное уравнение всегда имеет ровно два корня, но поскольку здесь мы занимаемся программированием, а не математикой, случай комплексных корней мы рассматривать не будем, ограничившись лаконичным школьным «корней нет».

[12] Встроенные типы мы подробно обсудим в следующем параграфе.

коэффициента уравнения, а сам дискриминант вычисляется по формуле $D = b^2 - 4ac$. Выше мы уже договорились, что для работы с дробными числами будем использовать тип `double`, то есть и коэффициенты, и сам вычисленный дискриминант должны быть как раз этого типа. Теперь уже написать функцию не составит особого труда:

```
double discrim(double a, double b, double c)
{
    return b*b - 4*a*c;
}
```

Заметим, что описания параметров имеют вид «тип-имя»; в языке Си это традиционный способ описания, причём не только для переменных, но и, например, для функций (в заголовке тоже сначала указывается тип — в данном случае это тип возвращаемого значения, — а потом уже имя функции).

Отметим, что мы могли бы написать и так:

```
double discrim(double a, double b, double c)
{
    double d;
    d = b*b - 4*a*c;
    return d;
}
```

и почему-то начинающие обычно так и пишут. Здесь используется ***локальная*** переменная `d`, в которую сначала заносится вычисленное значение дискриминанта, а затем значение этой переменной возвращается из функции. С технической точки зрения это не имеет никакого смысла, но если вам так понятнее, пишите так.

Начнём теперь писать функцию `main`. Как уже говорилось, она должна возвращать значение типа `int`, которое служит *кодом завершения программы*. Параметров она у нас пока что не принимает. Заметим, что нам обязательно потребуются локальные переменные — как минимум для хранения коэффициентов, а ещё довольно удобно будет сохранить в отдельной переменной вычисленное значение дискриминанта. Добавим к этому ещё одну переменную типа `int` (для хранения целого числа), зачем она нам нужна — мы поймём чуть позже. Локальные переменные в языке Си описываются сразу после фигурной скобки, открывающей функцию[13]; с учётом этого начало нашей функции `main` будет выглядеть так:

```
int main()
{
    double p, q, r, d;
    int n;
```

[13]Интересно, что, в отличие от Паскаля, в Си можно описать локальную переменную **в начале любого составного оператора**, то есть, например, можно завести свои локальные переменные в теле цикла и т. п.

Здесь можно заметить, что мы сэкономили объём текста, указав тип *один* раз и перечислив имена переменных через запятую. В описании переменных так делать можно, а вот в заголовке функции, к сожалению, нельзя, то есть мы не смогли бы добиться аналогичной экономии, например, в заголовке функции `discrim`. Если бы мы написали что-то вроде `double discrim(double a, b, c)`, то получили бы синтаксическую ошибку.

Ну что же, простая часть решения, собственно говоря, позади, а теперь начнётся самое интересное. Прежде чем идти дальше, давайте посмотрим, какие средства нам понадобятся, чтобы сначала ввести три числа, задающие коэффициенты уравнения, а затем два вычисленных корня напечатать. Начнём со второго. Как уже говорилось, функция `printf` умеет печатать не только строки, но и значения любых встроенных типов. Ранее мы использовали её с одним параметром (строкой), но на самом деле параметров у неё может быть сколько угодно, причём первым параметром в эту функцию передаётся так называемая ***форматная строка***, которая задаёт некий неизменный шаблон того, что должно быть напечатано, а дополнительные параметры, если они есть, задают то, что от случая к случаю может измениться. Функция `printf` просматривает свою форматную строку слева направо по одному символу, и если очередной символ — не «`%`», то функция такой символ просто печатает. Таким образом, если в форматной строке так и не встретилось ни одного «процента», то вся эта строка будет попросту напечатана как есть, чем мы и воспользовались в программе «Hello, world».

Гораздо интереснее функция будет действовать, если очередным символом окажется магический «процент». В большинстве случаев это означает, что нужно взять очередной параметр из списка параметров, преобразовать его в текстовое представление и напечатать. Один или несколько символов, идущих сразу после символа процента в форматной строке, указывают, какого типа будет этот параметр и в каком виде его печатать. Например, комбинация «`%d`» означает, что очередной параметр в списке следует рассматривать как целое число, а напечатать нужно его десятичное представление («`d`» в данном случае происходит от слова *decimal*). Комбинация «`%x`» означает практически то же самое, но напечатано число будет в шестнадцатеричной системе; ну а, к примеру, «`%05d`» означает, что печатаемое целое число будет содержать не менее пяти знаков, а если в числе знаков меньше, то оно будет дополнено слева нужным количеством нулей. Например, если мы напишем в программе

```
x = 17;
printf("%d times %d is %d\n", x, x, x*x);
```

то она напечатает «`17 times 17 is 289`» и переведёт строку; если же мы напишем

```
x = 378;
printf("%d %x %06d\n", x, x, x);
```

то напечатано будет «`378 17a 000378`» (и перевод строки).

Подробный разговор о возможностях `printf` у нас впереди; пока достаточно заметить, что для печати числа типа `double` (а равно и чисел типа `float`; функция их не различает) можно воспользоваться комбинацией «`%f`» (от слова *float*), но результат будет несколько странно выглядеть; правильнее будет явно указать, сколько знаков после десятичной точки[14] мы хотим видеть, и делается это так: «`%.5f`» (здесь мы указали, что в дробной части должно быть пять цифр).

Для ввода коэффициентов с клавиатуры нам придётся применить функцию `scanf`, у которой есть довольно много общего с `printf`. Эта функция также может принимать произвольное количество параметров, причём первым из них должна быть уже знакомая нам *форматная строка*. Её подробный разбор мы на некоторое время отложим, заметив, что в большинстве случаев форматная строка для `scanf` должна состоять из всё тех же комбинаций, начинающихся с «процента», между которыми ставят пробелы. В частности, строка `"%lf"` будет означать, что необходимо прочитать из потока стандартного ввода число типа `double` и записать его в переменную того же типа[15]. Переменную, соответственно, нужно *как-то указать в виде параметра*, и вот тут самое время вспомнить, что в Си есть только один способ передачи параметра — а именно, передача по значению. То есть передать «переменную как таковую», как мы делали это в Паскале с помощью var-параметров, нельзя — тут нет ничего похожего на var-параметры.

Чтобы понять, как решается эта проблема, вспомним для начала, что ***переменная*** (во всяком случае, в императивных языках программирования) — это не что иное, как *область оперативной памяти*, а к области памяти не обязательно обращаться через имя переменной, достаточно знать её (области) ***адрес***. Иначе говоря, если мы предложим функции `scanf` прочитать с клавиатуры число и дадим ей *адрес области памяти*, куда следует положить прочитанное число, она с этим вполне справится.

Уместно будет отметить, что var-параметры Паскаля на самом деле реализуются в машинном коде точно так же — передачей адреса переменной, но Паскаль от нас всю эту механику скрывает, тогда как Си придуман людьми, привыкшими к языку ассемблера, так что вполне понятно, почему они не сочли нужным ничего скрывать.

---

[14] Напомним на всякий случай, что в программировании для отделения дробной части от целой используется десятичная *точка*, а не десятичная запятая.

[15] Отметим, что, в отличие от *printf*, здесь *float* и *double* различаются; для *float* мы применяли бы комбинацию *"%f"*, тогда как для *double* вынуждены применять *"%lf"*. от слов *long float*, то есть «длинное с плавающей точкой»; почему это так, мы подробно обсудим в параграфе, посвящённом форматированному вводу-выводу.

Итак, чтобы прочитать три числа с клавиатуры, мы вызовем функцию `scanf` и передадим ей, во-первых, форматную строку, чтобы проинструктировать её читать именно числа с плавающей точкой, и именно три таких числа; и, во-вторых, мы передадим ей дополнительными параметрами *адреса* трёх переменных типа `double`. Отметим, что ***операция взятия адреса*** обозначается в Си символом «`&`», то есть значением выражения `&t` будет адрес переменной `t`.

Отметим ещё один важный момент. Функция `scanf` *возвращает значение*, и это тот случай, когда возвращаемым значением лучше всё-таки воспользоваться. Дело в том, что пользователь может ввести что-нибудь такое, что наша функция никак не сможет превратить в число — например, произвольную белиберду из букв. В этом случае `scanf` немедленно прекращает чтение и в переданные ей области памяти ничего не записывает, то есть там так и останется мусор. В такой ситуации продолжать решать квадратное уравнение несколько глупо, гораздо правильнее будет сообщить пользователю, что он неправ. Для этого как раз и используется значение, которое `scanf` возвращает как функция, а возвращает она *целое число, равное успешно обслуженным «процентикам»*. Поскольку таких у нас три, то и вернуть `scanf` должна число `3`; иное будет означать, что пользователь ввёл что-то некорректное.

Следующие несколько строк программы будут выглядеть так[16]:

```
n = scanf("%lf %lf %lf", &p, &q, &r);
if(n != 3) {
    printf("Error: wrong input.\n");
    return 1;
}
```

С присваиванием мы уже знакомы, а лексема `!=` обозначает логическую операцию «не равно» (то, что в Паскале мы привыкли обозначать как `<>`). С оператором `if` тоже всё, скорее всего, понятно по аналогии с Паскалем, стоит лишь обратить внимание на круглые скобки вокруг условия (они здесь обязательны) и отсутствие какого-либо аналога паскалевскому слову `then` — при наличии обязательных круглых скобок такое слово не нужно.

Обратите внимание, что, получив некорректный ввод и выдав по этому поводу сообщение об ошибке, наша функция `main` немедленно завершается с помощью уже знакомого нам оператора `return`, но при этом возвращает не ноль, как мы это делали раньше, а единицу. Таким способом она извещает операционную систему, что решить поставленную задачу не удалось (действительно, нельзя же решить квадратное уравнение, не зная его коэффициентов).

[16] Кто-нибудь мог бы заявить нам, что сообщения об ошибках следует выдавать не на стандартный вывод, а в специально предназначенный для этого поток диагностики; терпение, господа, всему своё время!

Дальше всё уже довольно просто. Если первый коэффициент равен нулю, то это уравнение не квадратное и решать его нужно совсем иначе; но наша программа предназначена для решения квадратных уравнений, а не каких-то других, так что она просто выдаст ошибку:

```
if(p == 0) {
    printf("Error: Not a quadratic equation!\n");
    return 2;
}
```

Как уже, несомненно, догадался читатель, знаком `==` обозначается сравнение, то есть приведённый оператор читается как «если `p` равно нулю, то...». Возвращаем мы в этот раз двойку, это тоже показывает системе наше недовольство, как и любое число, отличное от нуля. Но почему не единица, как в прошлый раз? Что ж, можно было вернуть и единицу тоже, но ведь ошибка-то другая. Возможно, кто-нибудь захочет написать другую программу, которая будет запускать нашу программу для решения какого-нибудь уравнения, и если мы будем выдавать разные коды ошибок, это позволит вызывающему понять, какая из ошибок имела место.

Убедившись, что первый коэффициент отличен от нуля, мы можем с полным на то основанием посчитать дискриминант, проверить его на неотрицательность и извлечь из него квадратный корень, причём сохранить этот корень можно в той же переменной, ведь сам дискриминант нам больше не понадобится:

```
d = discrim(p, q, r);
if(d < 0) {
    printf("No roots\n");
    return 0;
}
d = sqrt(d);
```

Обратите внимание, что в случае, если уравнение не имеет корней, мы завершаем программу с кодом 0 (успех). Дело в том, что отсутствие корней — это, вообще говоря, не ошибка, уравнение вполне может не иметь корней. Иначе говоря, корней мы не нашли, но *задачу как таковую решили*, ведь сообщение об отсутствии корней — это тоже ответ, притом правильный.

Случай с совпадающими корнями мы выделять не будем, так что всё, что осталось сделать — это напечатать вычисленные корни и завершить программу:

```
    printf("%.5f %.5f\n", (-q-d)/(2*p), (-q+d)/(2*p));
    return 0;
}
```

Полностью текст программы получился таким:

```
/* qe.c */
#include <stdio.h>
#include <math.h>

double discrim(double a, double b, double c)
{
    return b*b - 4*a*c;
}
int main()
{
    double p, q, r, d;
    int n;
    n = scanf("%lf %lf %lf", &p, &q, &r);
    if(n != 3) {
        printf("Error: wrong input.\n");
        return 1;
    }
    if(p == 0) {
        printf("Error: Not a quadratic equation!\n");
        return 2;
    }
    d = discr(p, q, r);
    if(d < 0) {
        printf("No roots\n");
        return 0;
    }
    d = sqrt(d);
    printf("%.5f %.5f\n", (-q-d)/(2*p), (-q+d)/(2*p));
    return 0;
}
```

Как ни странно, это отнюдь не конец истории. Сохраним файл и попробуем откомпилировать полученный результат:

```
avst@host:~/work$ gcc -Wall -g qe.c -o qe
/tmp/ccePjy5n.o: In function ‘main’:
/home/avst/work/qe.c:26: undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

Ключевым тут является словосочетание `undefined reference`, которое переводится как «неопределённая ссылка». Ошибку нам в этот раз выдал не компилятор, а редактор связей; компиляция-то как раз прошла успешно.

Вообще вышеприведённая диагностика может многое рассказать любопытному читателю. Для начала можно обратить внимание на имя файла `/tmp/ccePjy5n.o`; мы уже отмечали, что компилятор создаёт *временный объектный файл*, чтобы отдать его редактору связей, а потом стереть (см. стр. 202).

Здесь мы видим наглядное проявление этого факта: именно этот загадочный `ccePjy5n` и есть тот самый временный файл.

Заслуживает внимания и последняя строчка, где упоминается уже знакомая нам по урокам ассемблера команда `ld` — напомним, что это и есть редактор связей. Компилятор, завершив свою часть работы, сам вызвал программу `ld`, которая, не сумев выполнить возложенную на неё задачу (в данном случае из-за отсутствия функции `sqrt`), выдала диагностику — первые две строчки того, что мы увидели — и *завершилась с кодом 1*, тем самым сообщив вызвавшемому её компилятору, что произошла ошибка. Последнюю строчку напечатал уже сам компилятор. Как видим, вопрос о том, какое число возвращать из функции `main` в качестве кода завершения — отнюдь не праздный.

Ошибка произошла, потому что функция `sqrt` находится вне той части стандартной библиотеки языка Си, которая подключается по умолчанию. Компилятор узнал из заголовочного файла `math.h`, что функция `sqrt` существует, а также какие она принимает параметры и какого типа возвращает значение; всё это вполне совпало с тем, как мы эту функцию используем в программе, поэтому сам компилятор никаких проблем не заметил, вызвал редактор связей, и ошибка произошла уже на этапе финальной сборки. Чтобы исправить ошибку, нам нужно **действительно подключить библиотеку**. Нужная библиотека называется «`m`» (от слова *mathematics*), а подключается она указанием в командной строке компилятора флажка `-lm`:

```
gcc -Wall -g -lm qe.c -o qe
```

В этот раз никаких проблем не возникнет; теперь мы на собственном опыте убедились, что заголовочные файлы — это никакие не библиотеки, а кроме них есть ещё и *настоящие* библиотеки.

Попробуем потестировать нашу программу на простых данных:

```
avst@host:~/work$ echo "1 -2 1" | ./qe
1.00000 1.00000
avst@host:~/work$ echo "1 2 1" | ./qe
-1.00000 -1.00000
avst@host:~/work$ echo "1 -5 6" | ./qe
2.00000 3.00000
avst@host:~/work$ echo "1 1 1" | ./qe
No roots
avst@host:~/work$ echo "1 -1 -4" | ./qe
-1.56155 2.56155
```

4.2.4. Как узнать имя нужного заголовочного файла

В разобранных выше примерах нам встретилось два заголовочных файла: для использования функций ввода-вывода мы подключали `stdio.h`, а для функции вычисления квадратного корня нам понадобился `math.h`. В дальнейшем нам потребуются и другие заголовочные

файлы, а общее число заголовочников, покрывающих одни только возможности стандартной библиотеки, составляет несколько десятков.

К счастью, помнить, в каком заголовочнике расположена какая функция, нет ни малейшей необходимости. Если вы забудете подключить заголовочный файл, компилятор выдаст диагностику, начинающуюся с предупреждения примерно такого рода:

```
myprog.c:14:5: warning: implicit declaration of function 'malloc'
```

Конкретная формулировка предупреждения может зависеть от версии компилятора, но ключевые слова *implicit declaration* будут присутствовать обязательно. Поясним, что они переводятся как «неявное объявление»; дело в том, что по традиции, заложенной ранними версиями Си, функцию в некоторых случаях можно использовать, не объявив, поэтому компилятор не считает это ошибкой. Использовать эту возможность ни в коем случае не следует, ведь если мы не дали компилятору информации о том, каковы параметры нашей функции и какое значение она возвращает, то проверить наш вызов на соответствие типов компилятор уже не сможет — а значит, сделав ошибку, мы об этом не узнаем, или, точнее говоря, узнаем, но уже во время исполнения, и может уйти очень много времени на выяснение, что за ерунда происходит.

Так или иначе, зная имя функции, которая «не понравилась» компилятору, мы можем легко понять, какого заголовочного файла не хватает нашей программе. Для этого достаточно дать команду `man` с именем функции в качестве параметра:

```
man malloc
```

Мы получим на экран текст, описывающий заданную функцию, и в самом начале этого текста будут указаны заголовочные файлы, которые необходимо подключить для её использования. Если этих файлов больше одного, а такое иногда бывает, — это означает, что подключить их необходимо все, причём именно в таком порядке, как указано в описании, но при этом можно подключать их не подряд (то есть между любыми из них подключить какой-то ещё заголовочник).

Имена некоторых функций совпадают с именами команд командной строки; именно так обстоят дела, например, с хорошо знакомой нам функцией `printf`. Командный интерпретатор поддерживает одноимённую встроенную команду, которая несколько напоминает функцию Си по своему принципу работы; если мы напишем

```
man printf
```

то увидим описание *команды printf*, а не функции, и это совсем не то, что нам нужно. С этой проблемой легко справиться, зная, что тексты, выдаваемые командой `man`, сгруппированы в так называемые *секции*,

каждая из которых имеет свой номер. В частности, секция № 1 описывает команды и программы, которые установлены в системе; секция № 2 содержит описания системных вызовов, то есть таких функций Си, которые представляют собой обёртку для обращения к ядру операционной системы; наконец, секция № 3 посвящена функциям стандартной библиотеки языка Си. Есть и другие секции: например, секция № 5 содержит описания форматов файлов, а секция № 8 — описания команд, предназначенных для системного администратора, но нам пока достаточно первых трёх. Получить описание именно библиотечной функции, а не чего-то другого, можно, указав номер секции в явном виде:

```
man 3 printf
```

Если в этой секции нужного вам описания не найдётся, можно попробовать секцию 2 на случай, если функция, которую вы ищете, окажется системным вызовом:

```
man 2 wait
```

Отметим, что таким же точно образом — путём чтения соответствующего текста, выдаваемого командой `man` — можно узнать также о необходимости тех или иных библиотек; в частности, описание математических функций, включая `sqrt`, содержит указание на необходимость использования флага `-lm`.

4.3. Базовые средства языка Си

Предыдущая глава позволила нам получить первое впечатление о том, что представляет собой программирование на Си; теперь приступим к его систематическому изучению. При рассказе о Паскале мы вводили его средства постепенно, приводя примеры и решая задачи на каждую описанную возможность. С языком Си мы поступим проще, по крайней мере в самом начале: просто расскажем, что в нём есть.

4.3.1. Структура программы; объявления и описания функций

Программа на языке Си состоит из отдельных ***единиц трансляции*** — модулей (файлов), каждый из которых обрабатывается компилятором отдельно от других, т. е. компилятор, обрабатывая один модуль, ничего не знает о содержимом других модулей. Результатом трансляции модуля становится файл с объектным кодом; набор таких файлов связывается в готовую программу с помощью системного редактора связей (линкера).

Отдельно взятый файл, написанный на языке Си, состоит из ***глобальных объявлений*** и ***глобальных описаний***; и те, и другие вводят (*объявляют* и *описывают*) глобальные имена — это могут быть имена функций, имена глобальных переменных и констант, имена типов. Слово «глобальный» соответствует области видимости имени и означает, что такое имя видно *как минимум* от его объявления или описания до конца модуля, а в некоторых случаях может быть доступно и в других модулях.

Разница между объявлением и описанием состоит в том, что *объявление* лишь сообщает компилятору, что объект, соответствующий данному имени, *существует где-то* (возможно, в том же модуле, а возможно, что и в другом), тогда как *описание* даёт исчерпывающую информацию, связанную с именем, и предписывает компилятору создать соответствующий объект здесь и сейчас, расположив его в генерируемом объектном коде.

В частности, мы уже видели ***описания функций***, которые состоят из ***заголовка функции*** и ***тела функции***; при этом в заголовке записывается[17] тип значения, которое возвращает функция, затем имя функции и список её параметров, заключённый в круглые скобки, а тело функции, заключённое в фигурные скобки, состоит из необязательной секции описаний локальных имён и последовательности операторов. В предыдущем параграфе мы уже дважды описывали функцию с именем `main` и, кроме того, на стр. 205 была описана функция для вычисления дискриминанта квадратного уравнения. Приведём ещё один пример описания функции:

```
char case_up(char c)
{
    if(c >= 'a' && c <= 'z')
        return c - ('a' - 'Z');
    else
        return c;
}
```

Эта функция принимает в качестве параметра код символа, определяет, не соответствует ли этот код строчной латинской букве, и если это так, то возвращает код соответствующей заглавной буквы, в противном случае возвращает параметр без изменения. Здесь мы пользуемся известным фактом, что в таблице ASCII коды заглавных латинских букв идут подряд в соответствии с их алфавитным порядком и то же самое можно сказать о строчных буквах. Для нашего примера всё это неважно, нам просто нужна была какая-нибудь функция.

Обратите внимание, что здесь присутствуют все части описания функции — заголовок, состоящий из типа возвращаемого значения

[17] В некоторых экзотических случаях тип возвращаемого значения приходится задавать частично перед именем функции, частично — наоборот, уже после; мы столкнёмся с подобной ситуацией ближе к концу этой части книги. Отметим, что подобной экзотики всегда можно избежать.

`char`, имени `case_up` и списка формальных параметров `(char c)`, а также тело, состоящее из оператора `if`. Увидев такое описание, компилятор немедленно преобразует тело в объектный код, который поместит в *секцию кода* в формируемом объектном модуле; иначе говоря, компилятор выполняет наше предписание о создании функции `case_up` здесь и сейчас.

Мы могли бы ввести функцию `case_up` иначе — написав только её заголовок, а вместо тела поставив точку с запятой:

```
char case_up(char c);
```

Это уже не описание, а всего лишь *объявление* функции. Увидев его, компилятор не станет формировать какой бы то ни было объектный код и где-то его располагать, тем более что у него для этого просто нет необходимой информации — ведь мы же не показали компилятору тело функции, то есть фактически саму функцию. Компилятор просто *примет к сведению*, что *где-то* есть функция с именем `case_up`, принимающая на вход один параметр типа `char` и возвращающая тоже `char`; это означает, что, встретив *вызов* этой функции, компилятор сверит типы параметра и возвращаемого значения в контексте такого вызова с имеющейся информацией и, если всё в порядке, успешно откомпилирует такой вызов. Что касается самой функции, то она может появиться позже в том же модуле, а может не появиться вообще — компилятор в этом случае будет предполагать, что функция находится в другой единице трансляции, а заботу о её поиске оставит редактору связей.

Раз уж речь зашла о функциях, стоит сказать, что, хотя в языке Си нет отдельного понятия для *процедур*, некий аналог паскалевской процедуры можно получить, заявив, что функция ничего не возвращает. Это делается указанием ключевого слова `void` вместо типа функции. Пусть, например, нам нужна подпрограмма для печати заданного символа заданное число раз. В Паскале мы бы, разумеется, оформили такую подпрограмму в виде процедуры, но в Си процедур вроде бы нет; однако это не отменяет того факта, что нам попросту *нечего* возвращать из функции, имеющей такое предназначение, она всегда будет вызываться только ради побочного эффекта. Этот факт мы можем отразить в нашей программе, указав, что функция, хоть и называется функцией, всё-таки никаких значений не возвращает:

```
void print_char_n(char c, int n)
{
    int k;
    for(k = 0; k < n; k++)
        printf("%c", c);
}
```

Как вы уже, несомненно, догадались, `%c` (от слова *char*) в форматной строке `printf` означает печать символа с заданным кодом; код в данном случае задаётся значением переменной `c`. С циклом `for` всё несколько сложнее; в данном случае переменная `k` будет пробегать значения от `0` до `n-1` включительно, примерно как если бы мы на Паскале написали что-то вроде

```
for k := 0 to n - 1 do
```

Полностью с возможностями `for` мы ознакомимся в §4.3.6.

4.3.2. Переменные и их описание

Объявлять и описывать мы можем не только функции. В частности, если поместить описание переменной *вне тел функций*, например:

```
int dangerous_global_variable;
```

то такая переменная будет *глобальной*, то есть будет видна от места своего описания до конца единицы трансляции, и доступ к ней можно осуществлять из любой функции; больше того, такую переменную можно в принципе «достать» и из других модулей. Си позволяет не только описывать, но и *объявлять* глобальные переменные, а также ограничивать видимость переменных (и функций тоже) текущим модулем, но об этом речь пойдёт в главе, посвящённой раздельной трансляции программ. Пока же считаем уместным напомнить, что **использовать глобальные переменные крайне нежелательно** (см. т. 1, стр. 251).

Коль скоро речь зашла об описаниях переменных, отметим ещё одну интересную возможность. При описании переменной можно в явном виде задать её начальное значение, например, так:

```
int dangerous_global_variable = 42;


int f()
{
    int local_var = 42;
    /* ... */
```

Такая конструкция называется ***инициализацией***, а выражение справа от знака равенства — ***инициализатором***. Необходимо подчеркнуть, что **инициализация не имеет ничего общего с присваиванием**, хотя и обозначается тем же знаком. Присваивание — это *деструктивная* операция, она разрушает имеющееся значение, записывая вместо него новое; инициализация ничего не разрушает, ведь переменной до момента её описания просто не было, и, значит, не было значения переменной.

Как мы увидим позже, с помощью инициализаторов можно сделать гораздо больше, чем с помощью присваиваний: например, массивы присваивать нельзя, а инициализировать можно.

Если в описании переменной нет инициализатора, то её исходное значение зависит от того, локальная она или глобальная[18]. Глобальные переменные, для которых начальное значение не задано, заполняются нулями; локальные переменные в отсутствие инициализаторов никаких начальных значений не получают, то есть в локальной переменной может оказаться абсолютно произвольный мусор.

Припоминая сведения из части, посвящённой языку ассемблера, поясним, что глобальные переменные, для которых задано начальное значение, располагаются в секции `.data`, а неинициализированные глобальные переменные компилятор помещает в секцию `.bss` (см. §3.2.2); при загрузке исполняемого файла в память операционная система заполняет нулями всю область, отведённую под секцию `.bss`, чем и объясняются начальные нулевые значения для неинициализированных глобальных переменных. Что касается локальных переменных, то они располагаются *в стековых фреймах* (см. §3.3.6), которые, как мы помним, создаются при вызове функции и уничтожаются при её завершении. Если очистка (принудительное заполнение нулями) секции `.bss` происходит один раз при загрузке программы и много времени не отнимает (на фоне времени, которое тратится на чтение в память машинного кода из исполняемого файла, на зануление `.bss` можно вообще не обращать внимания), то очистка стековых фреймов, если бы кто-то решил её сделать, потребовала бы лишнего цикла (записи нулей) *при каждом вызове функции*; это происходило бы миллионы, миллиарды, триллионы раз за время выполнения программы, и эффективность пострадала бы весьма заметно. Поэтому стековые фреймы никто не очищает, и в ячейках памяти, отведённых под очередной фрейм, остаются те значения, которые там были (остались от выполнения ранее вызывавшихся функций). Это и есть мусор, который мы наблюдаем в неинициализированных локальных переменных. Использование значения такой переменной до того, как в неё будет какое-то значение занесено — грубейшая ошибка, которую к тому же в общем случае не может «поймать» компилятор, хотя если он всё же смог её обнаружить, он выдаст вам предупреждение.

Отметим, что инициализация глобальных переменных ничего нам не стоит, кроме нескольких байтов в исполняемом файле, поскольку образ секции `.data`, содержащий все эти начальные значения, целиком размещается в исполняемом файле, при старте программы одним махом копируется в оперативную память, так что начальные значения глобальных переменных оказываются ровно там, где нужно; программа не тратит никакого времени на их инициализацию. В противоположность этому, инициализация локальных переменных на уровне машинного кода реализована точно так же, как обыкновенное присваивание, так что если вы,

[18] Из этого правила есть одно важное исключение — так называемые локальные статические переменные, которые будут рассмотрены позднее.

например, в самом начале функции присваиваете локальной переменной некое значение, задавать для этой переменной инициализатор не нужно: экономия двух-трёх тактов процессорного времени, конечно, не слишком серьёзна, но и терять эти такты на пустом месте не стоит.

4.3.3. Встроенные типы

Надо сказать, что по сравнению с Паскалем набор встроенных (базовых) типов языка Си может показаться весьма компактным и даже аскетичным. Фактически здесь есть только целые числа (различающиеся разрядностью и знаковостью) и числа с плавающей точкой (различающиеся разрядностью). И на этом, собственно, всё: нет ни специального типа для символов, ни логического типа, ни строкового.

Достигнуто это довольно просто. Программируя на языке ассемблера, мы на собственном опыте убедились, что для чисел с плавающей точкой у нас есть отдельный «сопроцессор» со своими регистрами, тогда как всё остальное — и символы, и логические значения, и вообще всё, с чем нам взбредёт в голову работать, процессор «перемалывает», используя свои основные регистры, которые хранят не что иное, как *целые числа*. Иначе говоря, если то, с чем мы работаем, не является числом с плавающей точкой (или, возможно, несколькими такими числами), то оно является (на уровне машинных команд) целым числом или набором целых чисел. При создании языка Си никому не пришло в голову этот факт каким-либо образом маскировать, поэтому, в частности, **в качестве логического значения в Си обычно используются целые числа**, причём `0` означает «ложь», а всё остальное — «истину»[19]. Так, если у нас есть целочисленная переменная `a`, то вместо `if(a != 0)` мы могли бы написать просто `if(a)` — работать это будет точно так же.

Как мы знаем, ***символ*** представляется в памяти машины своим ***кодом***, то есть, опять же, целым числом. Как несложно догадаться, авторы Си не стали проводить привычное нам по Паскалю различие между самим символом и его кодом, то есть в Си **символ и его код — это одно и то же**. Для обозначения символов (в отличие от строк!) в Си используются одиночные апострофы, например так: `'a'`; но это для компилятора *абсолютно то же самое*[20], как если бы мы написали просто `97` (именно таков код буквы `a` в таблице ASCII). Мало того, в Си совершенно легитимным оказывается выражение `'a'+'b'`, оно равно сумме кодов букв `a` и `b` (то есть числу `195`), хотя, конечно, смысла в таком выражении никакого нет и писать так не следует.

[19] Забегая вперёд, отметим, что в качестве логического значения можно также использовать адресное выражение; нулевой указатель в этом случае будет обозначать «ложь», а любой другой — «истину».

[20] Для наиболее «продвинутых» читателей сообщим, что выражение `'a'` даже имеет тип `int`, а не `char`, несмотря на наличие в Си такого типа.

Начнём с рассмотрения целочисленных типов. Разрядность (то есть, грубо говоря, количество памяти, отводимое под соответствующее значение) этих типов варьируется в довольно широких пределах; для обозначения разрядности используются ключевые слова `char`, `short`, `int` и `long`, а наибольшей возможной разрядности можно достичь, используя тип `long long` («длинное длинное»).

Наименьшую разрядность имеет тип, который называется `char`; в большинстве случаев он имеет размер 1 байт (то есть 8 бит), хотя теоретически возможны аппаратные платформы, где у этого типа разрядность окажется какая-то другая. Строго говоря, тип `char` на любой платформе равен по размеру *минимальному адресуемому*[21], то есть минимальной области памяти, у которой имеется свой собственный адрес. Мы знаем, что ячейка памяти на всех современных машинах (как и на большинстве не очень современных) составляет именно байт, так что можем на эту тему особенно не беспокоиться. Интереснее другое: тип `char` может оказаться как знаковым, так и беззнаковым, то есть, грубо говоря, если мы опишем переменную этого типа, то она может оказаться способна принимать значения от 0 до 255 либо от -128 до 127, и зависит это исключительно от компилятора. Впрочем, язык позволяет взять этот момент под контроль, явным образом указав ***знаковость***, для чего предусмотрены слова `signed` и `unsigned`[22]; с учётом этих слов у нас появляются *ещё два типа* — `signed char` (знаковый) и `unsigned char` (беззнаковый), что же касается обычного `char`, то он совпадает с одним из этих двух, но вот с каким именно — неизвестно.

Чтобы понять причину столь странной ситуации, следует вспомнить, что тип `char`, как это следует из его названия, исходно предназначался для хранения кода символа, а таблица ASCII содержит всего 128 позиций, от 0 до 127. В те времена и в том месте, когда и где язык Си обретал более-менее устойчивые очертания, никому не было никакого дела до кодирования символов, не попавших в ASCII, так что «расширенных ASCII-таблиц» вроде koi8-r или cp1251 не было даже в проекте, ну а современные многобайтовые кодировки, основанные на Unicode, не могли бы присниться создателям Си даже в самом страшном ночном кошмаре.

С учётом этого внезапно оказывается, что знаковость типа `char` не так уж важна: все коды ASCII-таблицы благополучно покрываются обоими вариантами.

Сразу же отметим, что такая ситуация наблюдается *только с типом* ***`char`***, а все остальные целочисленные типы по умолчанию знаковые и становятся беззнаковыми, только если написать слово `unsigned`; применять с ними слово `signed` можно, но бессмысленно, то есть фактически это слово нужно лишь затем, чтобы гарантированно сделать знаковым этот вот «хитрый» `char`. То, что в Си аж целое ключевое слово введено

[21] Заметим, что Керниган и Ритчи в своей книге вообще не упоминают никакие «минимальные адресуемые», а говорят просто «байт»; всё остальное — происки комитетов.

[22] Читается приблизительно как «сайнт» и «ансайнт».

для столь несуразной цели, можно рассматривать как очень характерный пример на тему общей неряшливости построения этого языка; впрочем, это далеко не самый убедительный пример, в дальнейшем мы увидим много выкрутасов подобного рода. Так или иначе, как уже говорилось, мы язык Си любим не за это.

Следующим по разрядности идёт тип `short` и его беззнаковый аналог `unsigned short`. Как водится, требования к языку Си не задают конкретной разрядности для этого типа, известно только, что он не имеет права быть меньше `char`'а (впрочем, трудно быть меньше, нежели минимальное адресуемое); однако есть и хорошая новость: вы можете считать, что переменная этого типа занимает два байта, и это окажется верно на любой машине, с которой вам доведётся столкнуться в реальной жизни; хотя, конечно, необходимо помнить, что *теоретически* возможно и иное. Как следствие, если вы пишете программу не «для себя», а для заказчика, лучше всё-таки не делать в ней подобных предположений.

Отметим ещё один интересный казус: исходно предполагалось применять слово `short` в качестве *модификатора*, то есть «прилагательного» к слову `int` подобно словам `signed` и `unsigned`; так можно делать и до сих пор, то есть можно описать переменную, к примеру, вот так:

```
unsigned short int x;
```

От этого стиля в основном отказались из-за довольно простого соображения: простите, а ***что ещё*** тут может быть «коротким»? Оказалось, что ничего. Поэтому сейчас слово `int` можно (и нужно) опускать.

После `short`'ов идёт уже знакомый нам `int` в компании с `unsigned int`'ом. С его разрядностью дела обстоят, пожалуй, самым забавным образом. На старых 16-битных платформах (например, в MS-DOS) `int` был 16-битным, то есть совпадал по своей разрядности с `short`'ом. При переходе к 32-битным архитектурам тип `int` вырос до четырёх байт и стал совпадать с типом `long`; бытовало даже мнение, что тип `int` на любой платформе должен соответствовать размеру машинного слова. Такое предположение не лишено оснований, ведь согласно традициям построения компиляторов Си, например, возврат значения из функции производится через регистр (обычно тот, который по какой-то причине можно считать «главным»; в случае i386 это регистр `EAX`; его часто называют «аккумулятором»), если только возвращаемое значение в этот регистр помещается; при этом до не слишком давнего времени считалось, что если для функции не указан тип возвращаемого значения, то она возвращает `int` (сейчас это тоже так, но компиляторы выдают предупреждение).

Несмотря на это, при переходе от 32-битных архитектур к 64-битным `int` так и остался четырёхбайтным, оно и понятно: если бы его «вырастили» вслед за разрядностью регистров, то целого типа шириной в четыре

байта в языке бы просто не осталось. Зато на 64-битных платформах «подрос» следующий тип, `long`; он стал 64-битным, в результате чего *совпадает с типом* `long long`. Но здесь мы уже забежали вперёд.

Очередная пара типов одинаковой разрядности — это `long` и `unsigned long`; до наступления эры 64-битных платформ эти типы всегда были четырёхбайтными, а с наступлением этой эры внезапно выросли. Как и слово `short`, изначально слово `long` планировалось использовать как прилагательное, а переменные описывались как-то вроде `long int x`. В отличие от слова `short`, слово `long` имеет ещё одно применение, о котором мы узнаем из обсуждения типов с плавающей точкой; тем не менее, для лучшего единообразия слово `int` разрешили опускать для случая «длинных» точно так же, как и для «коротких».

Пара самых больших целочисленных типов, обозначаемых как «длинное длинное»[23], то есть, соответственно, `long long` и `unsigned long long`, поддерживается не всеми компиляторами и не на всех платформах; но везде, где эти типы присутствуют, их разрядность составляет 64 бита. Даже при появлении 64-битных процессоров `long long` в размерах не вырос.

Отметим ещё один момент: действует соглашение, по которому на каждой архитектуре разрядности числа типа `long` должно хватать для представления указателей, хотя это соглашение не является частью официально утверждённых стандартов языка Си.

Говоря о размерах, следует всегда помнить, что любые разговоры о конкретной разрядности целочисленных типов языка Си возможны только с констатационной позиции, то есть по принципу «существующее положение вещей таково», с обязательным учётом того обстоятельства, что завтра может появиться или новая платформа, или новый компилятор под существующую платформу, и там всё будет не так. Гарантировать можно только одно: `short` не будет короче `char`'а, `int` не окажется короче `short`'а, `long` ни за что не станет короче `int`'а, ну и, конечно, `long long` не может стать короче, чем простой `long`. Но и только.

Система типов с плавающей точкой несколько проще, этих типов всего три: `float`, `double` и `long double`, причём беззнаковыми они не бывают. На платформе i386, как правило, `float` и `double` соответствуют числам обычной и двойной точности из стандарта IEEE-754, а тип

[23] Следует отметить, что адекватное звучание этого термина на русском языке достигается не тогда, когда мы представляем себе что-то вроде *длинное-длинное*, то есть *длинное-предлинное*, а когда мы первое слово воспринимаем как прилагательное, а второе — как существительное; то есть у нас имеется такая сущность «длинное» (в данном случае это существительное), а потом мы это «длинное» ещё сильнее удлинили, и получилось у нас не *простое* длинное, а вот это вот *длинное* длинное. Англоязычным людям в этом плане проще, там одна и та же словоформа почти всегда может использоваться и как существительное, и как прилагательное, и как глагол.

`long double` предоставляет доступ к числам *повышенной точности* (напомним, таковые занимают 10 байт); при этом в памяти переменная типа `long double` может занимать 12 или даже 16 байт. Если же говорить не о конкретной платформе, а «вообще», то сказать тут можно только одно: `double` не может быть короче, чем `float`, или длиннее, чем `long double`; при этом, например, все три типа могут между собой совпадать, что достаточно часто встречается, когда в процессоре отсутствует аппаратная поддержка арифметики с плавающей точкой.

Отметим ещё один момент: математические функции, такие как уже знакомый нам `sqrt`, а также всевозможные синусы, косинусы, экспоненты и логарифмы по умолчанию работают с типом `double`; больше того, для любых вычислений и даже для простой передачи параметров значения типа `float` всегда преобразуются к типу `double`, что делает использование типа `float` бессмысленным во всех случаях кроме одного: когда таких чисел очень много, а памяти не хватает (а уменьшение её расхода вдвое способно решить проблему). Иначе говоря, пока вы не встретились с нехваткой памяти, `float` применять не надо.

Язык Си предусматривает специальную возможность для определения размеров любых типов — ключевое слово `sizeof`. Выражение вида `sizeof(` *<тип>* `)` компилятор заменяет на целое число, равное размеру данного типа, причём *единицей измерения является* `char`. Выражение `sizeof(char)` по определению равно единице. Кроме имён типов, `sizeof` можно применять также к именам переменных и вообще произвольным выражениям, тип которых компилятор может определить. Например, `sizeof(1)` есть то же самое, что `sizeof(int)`.

Более того, синтаксически `sizeof`, формально говоря, является операцией, а не функцией, так что, применяя его к выражениям (но не к типам), можно не писать скобки; впрочем, ясности программе это не добавит, так что скобки лучше всё-таки поставить.

4.3.4. Литералы (константы) разных типов

Литералами обычно называют значения, записанные в программе в явном виде; простейший пример литерала — целое число, записанное как оно есть.

Язык Си позволяет записывать целые числа в десятичной, шестнадцатеричной и восьмеричной системах счисления. Если написать число обычными арабскими цифрами **без лидирующего нуля**, компилятор будет рассматривать его как десятичное; если же мы **допишем спереди ноль**, это число будет рассматриваться как записанное в **восьмеричной** системе. Например, `035` означает то же, что и `29`

$(3 \times 8 + 5 = 29)$[24]. Для записи шестнадцатеричных чисел используется префикс «0x», например, всё то же десятичное 29 может быть записано как 0x1d или 0x1D, а запись 0x7DF означает 2015.

Довольно нетривиальным образом обстоят дела с разрядностью и знаковостью целочисленных литералов. Десятичные литералы *в большинстве случаев* рассматриваются как знаковые, тогда как восьмеричные и шестнадцатеричные — всегда беззнаковые; но это только начало истории.

Если десятичный литерал представляет число, «умещающееся» в разрядность числа типа `int`, то он будет считаться имеющим тип `int`; если он перестаёт умещаться в `int`, но пока что помещается в `unsigned int`, то он будет считаться числом типа `unsigned int`. Если для записанного числа не хватает разрядности `unsigned int`, компилятор попытается представить его как `long`, затем как `unsigned long`, как `long long` и как `unsigned long long`, но если и тут не хватит разрядности, то будет, наконец, выдана ошибка.

С литералами восьмеричными и шестнадцатеричными ситуация примерно такая же, только компилятор не пытается делать их знаковыми. Иначе говоря, если хватает разрядности `unsigned int`'а, то он и используется, если же не хватает, компилятор пытается использовать `unsigned long` и `unsigned long long`.

Кроме того, программист может сам явным образом задать тип литерала, добавив в его конец комбинацию из суффиксов `L` (`long`), `LL` (`long long`)[25] и `U` (`unsigned`): например, 15 будет иметь тип «знаковый `int`», тогда как 15ULL будет типа `unsigned long long`. В принципе, эти суффиксы можно записывать также и в нижнем регистре (что-нибудь вроде 200ull), но делать так не рекомендуется, потому что букву `l` очень легко спутать с единицей.

Вопреки распространённому заблуждению, литералов типа `char` в языке Си нет; литералы вида `'a'`, `'Z'` и т. п. имеют тип `int`. Литералы типа `short` и `unsigned short` также не предусмотрены.

Числа с плавающей точкой традиционно записываются в виде литералов в десятичной[26] системе счисления. В общем случае такой литерал состоит из целой части, точки, дробной части и *порядка*, причём порядок не обязателен, можно указать одну только целую или одну только дробную часть, если же порядок присутствует, то можно опустить дробную часть вместе с точкой. Порядок записывается, как обычно, в виде буквы `e` или `E`, за которой следует десятичное целое число (возможно, снабжённое знаком), и означает, что исходное число следует домножить на 10^k, где k — число, указанное в порядке. Например, следующие литералы обозначают одно и то же число:

```
1500.0  1500.  15E2  15.E2  15E+2  0.15e4  .15e4  15000e-1
```

[24] На самом деле это не совсем одно и то же, поскольку десятичные константы по умолчанию знаковые, тогда как восьмеричные и шестнадцатеричные — беззнаковые, но в реальной жизни эта разница обычно не играет роли.

[25] В пресловутые стандарты Си суффикс `LL` вошёл только начиная с C99, но реально этот суффикс поддерживался везде, где присутствовал тип `long long`.

[26] В C99 была добавлена возможность записи шестнадцатеричных чисел с плавающей точкой, но мы её рассматривать не будем.

По умолчанию такие литералы имеют тип `double`, но вы можете принудительно превратить их во `float`'ы добавлением суффикса `f` или `F`, а также в `long double`'ы добавлением уже знакомого нам суффикса `L` (можно и `l`, но не надо).

Вообще говоря, все эти возможности нужны достаточно редко: для реализации математических вычислений есть более удобные языки программирования, нежели Си.

Как уже говорилось, в языке Си *символ* и *код символа* — это одно и то же, а для записи кодов символов используются ***символьные литералы***, записываемые с использованием апострофов, причём они имеют тип `int`. В частности, `'A'` — это то же самое, что и `65`, а `'1'` — то же, что `49`.

Для представления ***служебных символов*** предусмотрены так называемые ***escape-последовательности***, одну из которых — «`\n`» — мы уже много раз встречали. Поскольку код перевода строки — `10`, запись `'\n'` означает то же, что и `10`. Кроме перевода строки, предусмотрены также `'\r'` — возврат каретки (код `13`), `'\t'` — табуляция (код `9`), `'\a'` — «звонок» (от слова *alarm*, код `7`), `'\b'` — «забой» (от слова *backspace*, код `8`), `'\f'` — так называемый перевод страницы, *form feed* (код `12`) и совсем уже экзотическая `'\v'` — «вертикальная табуляция» (код `11`). Поскольку в escape-последовательностях используется символ обратного слэша «\», а сами последовательности применяются внутри символьных и строковых литералов, обрамляемых апострофами и двойными кавычками, предусмотрены вполне логичные способы представления этих трёх символов: `'\\'` обозначает код обратного слэша, `'\''` и `'\"'` — соответственно коды апострофа и кавычки; впрочем, код кавычки можно получить и проще: `'"'`.

Для представления ***строк*** используются ***строковые литералы***, обрамляемые двойными кавычками, и все перечисленные escape-последовательности действуют в них так же, как в символьных литералах, за исключением того, что символ апострофа здесь можно записать как есть (без escape-последовательности), тогда как кавычку, если она потребовалась вам в строке, придётся обязательно снабдить символом слэша, например: `"Never say \"never\""`.

Очень важно уяснить разницу между символьными литералами (в апострофах) и строковыми (в кавычках)! Например, когда компилятор видит литерал `'Z'`, он заменяет его на *целое число* 90 (код буквы `Z`), тогда как если компилятор увидит литерал `"Z"`, он сделает совершенно иное: расположит где-то (на самом деле — в секции кода) массив из двух элементов типа `char` с первым элементом, равным `90`, а вторым — равным нулю, при этом сам литерал компилятор заменит на *адрес* той области памяти, где расположен массив. Если это оказалось непонятно, ничего страшного, разговор о массивах и строках

у нас ещё впереди; пока просто запомните, что **строки и символы — это совершенно разные сущности**.

Кроме вышеперечисленных escape-последовательностей, язык Си предусматривает возможность задать символ его кодом в восьмеричной или шестнадцатеричной системе; например, `'\132'` и `'\x5a'` означают то же, что и `'Z'`. Если такая последовательность встречается в строковом литерале, то для восьмеричного варианта используется от одного до трёх символов, пригодных в качестве восьмеричной цифры (то есть, например, литералы `"a\13222"` и `"aZ22"` эквивалентны, как и литералы `"\tX"` и `"\11X"`); для шестнадцатеричного компилятор пытается «съесть» все символы, пригодные в роли шестнадцатеричной цифры, сколько бы их ни было, и выдаёт ошибку, если их слишком много.

В программах на Си часто можно встретить запись `'\0'` для обозначения сущности «символ с кодом ноль». Семантически это, разумеется, абсолютно то же самое, что и просто `0`, но со стилистической точки зрения такая запись оправдана — читателю программы сразу станет понятно, что имеется в виду именно символ, а не просто арифметический ноль.

Отметим ещё одну удобную возможность, связанную со строковыми литералами. Если несколько таких литералов записать один за другим, разделив их только пробельными символами, компилятор «склеит» их все в одну строку. Например, следующий фрагмент

```
"This " "is" " a string" " whi" "ch is" " long"
```

есть то же самое, что и

```
"This is a string which is long"
```

Это бывает полезно, если в программе необходима настолько длинная строковая константа, что она не влезает на экран (точнее, в 80 колонок) по ширине.

4.3.5. Операции и выражения

Арифметические выражения в языке Си играют существенно более важную роль, чем в привычном нам Паскале. Как мы увидим позже, в Си присутствует такая нетривиальная вещь, как *адресная арифметика*; кроме того, столь привычное нам *присваивание*, которое в Паскале было *оператором*, в Си совершенно неожиданно оказывается *операцией* и может, как следствие, быть частью арифметического выражения. Но — обо всём по порядку.

Арифметика, логика, биты

Начнём мы с того, что над всеми имеющимися в языке встроенными типами (а они, как мы помним, все числовые) определены обычные и ничем особенным не примечательные сложение «`+`», вычитание «`-`» и

умножение «*». Разумеется, присутствует также и деление, обозначаемое вполне ожидаемой на эту роль дробной чертой «/», но здесь уже необходимы пояснения. Дело в том, что семантика операции деления **зависит от типа операндов**: если операнды представляют собой числа с плавающей точкой, то это будет обычное арифметическое деление, тогда как если операнды целые — то деление будет *целочисленным*, подобно паскалевскому `div`. Так, значение выражения `5.0/2.0` *приблизительно* равно `2.5`, тогда как значение выражения `5/2` равно `2`, притом безо всяких «приблизительно» (операции над целыми, как мы знаем, выполняются точно).

В завершение рассказа о простых арифметических операциях отметим, что *взятие остатка от деления* в языке Си обозначается символом процента «%».

Здесь у читателя может возникнуть целый ряд вопросов. Что будет, например, если у операции деления один операнд целый, а другой «плавучий»? Как работает взятие остатка, если один или оба операнда отрицательны? Можно ли применять взятие остатка к дробным числам?

На все эти вопросы можно дать вполне однозначные ответы, однако делать этого мы не будем: не надо засорять мозг информацией подобного свойства. Если вопросы такого рода возникают из чистого любопытства, можно взять любую более-менее серьёзную книжку по Си, можно поискать (и практически мгновенно найти) ответы в Интернете. Важнее другое: *в процессе практического программирования такие вопросы не должны возникать.* В частности, применять деление к операндам разных типов *не надо* — операнды следует привести к одному типу или хотя бы сделать их или оба целыми, или оба «плавающими», а брать остаток от деления следует *только при целых положительных операндах*; если вам кажется, что возникла потребность в ином, нужно проанализировать, откуда такая мысль пришла в голову: знайте, что-то тут пошло не так, в нормальных условиях такое понадобиться не должно.

Набор *операций сравнения* в Си вполне привычен; порядковые операции записываются так же, как в Паскале: `<`, `>`, `<=`, `>=`, что же касается сравнения на равенство и неравенство, то здесь они пишутся иначе: «==» (**двойной** знак равенства) означает «равно», а комбинация «`!=`» — «не равно». **Обратите особое внимание на удвоенный знак равенства!** Дело в том, что одиночный знак равенства, как мы уже видели, в Си означает присваивание, причём выше уже упоминалось, что присваивание может встречаться в выражениях. Если, к примеру, вы напишете `if(a = 3)`, программа успешно откомпилируется и запустится, но работать будет совсем не так, как если бы вы написали `if(a == 3)`. Хорошая новость для начинающих состоит здесь в том, что если вы не забываете про флажок `-Wall` при запуске компилятора, то программа, конечно, всё равно откомпилируется, но вы хотя бы получите предупреждение; плохая новость для лентяев — без флага `-Wall` компилятор просто молча проглотит конструкцию, в большинстве случаев ошибочную.

Результатом операции сравнения становится *целое число*, причём, как уже говорилось, `0` означает «ложь»; в принципе, в качестве представления «истины» эти операции всегда возвращают единицу, что позволяет, например, написать вот такую вот белиберду:

```
x = (a < b) + (a > c) - (c != t);
```

Так вот, **не надо так делать**, даже если вы найдёте ситуацию, в которой подобная конструкция вам покажется осмысленной. При чтении программы подобные трюки приходится подолгу анализировать, чтобы понять, что́ имелось в виду.

Введя логические отношения, мы естественным образом приходим к вопросу о логических операциях. В Си они представлены стандартным набором «конъюнкция, дизъюнкция, отрицание»: операция «логическое и» записывается знаком «`&&`» (два амперсанда), «логическое или» — знаком «`||`» (две вертикальные черты), унарная операция отрицания обозначается восклицательным знаком «`!`». Необходимо сразу же отметить одно интересное свойство бинарных логических операций: **если значение выражения уже определено, второй операнд не вычисляется**. Иначе говоря, если в программе имеется выражение `f() || g()` и функция `f()` вернула значение, отличное от нуля (т. е. «истину»), функция `g()` вообще не будет вызвана, что особенно важно учитывать, если у этой функции имеется побочный эффект. Аналогично при вычислении выражения `f() && g()` функция `g()` не будет вычисляться, если `f()` вернула ноль.

Проницательный читатель может спросить, почему используется *два* амперсанда, а не один, и *две* «палочки», а не одна. Дело в том, что *одним* амперсандом обозначается операция «и», но не логическая, а *побитовая*, то есть эта операция производится по отдельности над первыми битами обоих операндов, над вторыми, над третьими и т. д., и результаты коньюнкции отдельных битов образуют результат всей операции; например, значением выражения `25 & 62` будет `24` (чтобы понять, почему это так, переведите 25 и 62 в двоичную систему). Аналогичным образом одной «палочкой» обозначается побитовое «или», так что, например, значением `12 | 56` будет `60`.

В Паскале, напомним, ключевые слова `and` и `or` обозначают как логические, так и побитовые варианты соответствующих операций; компилятор Паскаля различает эти случаи по типам операндов: если это `boolean`'ы, то выполняется логическая операция, если целые числа любой разрядности — то побитовые. В Си так сделать нельзя, поскольку отдельного типа для обозначения логических значений тут нет.

Побитовое отрицание обозначается знаком «`~`» (тильда); например, `~0` будет равно `-1` (все биты — единицы). Кроме того, в Си присутствует операция побитового «исключающего или» (знакомый нам `XOR`), которая

обозначается символом «^». Логического варианта для этой операции не предусмотрено.

Завершая разговор о побитовых операциях, опишем побитовые сдвиги влево (знак «<<») и вправо («>>»); левый операнд обеих операций — исходное сдвигаемое число, правый — количество битов, на которое необходимо произвести сдвиг. Сдвиг влево выполняется одинаково для всех целочисленных типов: биты сдвигаются влево на заданное число позиций, справа дописываются нули (как если бы число умножили на двойку в соответствующей степени). Со сдвигом вправо всё несколько сложнее, его выполнение зависит от *знаковости* сдвигаемого числа. Биты в представлении числа, естественно, в любом случае сдвигаются вправо на заданное число позиций, но слева (то есть в старшие разряды) для беззнакового числа записываются нули, тогда как для знакового числа — *значение старшего бита исходного числа*; знак числа, таким образом, всегда сохраняется, а сама операция всегда остаётся эквивалентной целочисленному делению на соответствующую степень двойки.

Поскольку с операциями побитовых сдвигов мы уже сталкивались при изучении языка ассемблера, отметим, что для беззнаковых чисел сдвиг выполняется командами SHL и SHR, а для знаковых — командами SAL и SAR; напомним, что в действительности SHL и SAL — это одна и та же команда, тогда как SHR и SAR — разные.

Присваиваивания и модификации

Привычные операции на этом кончаются и начинается всевозможная экзотика. Одна из «фирменных» особенностей Си — ***операция присваивания*** (а не оператор присваивания, как в большинстве языков, которые вообще предусматривают присваивание). Присваивание обозначается, как мы уже видели, простым знаком равенства. Хитрость в том, что конструкция вида `a = b` представляет собой *выражение*, и это выражение имеет значение — оно равно тому значению, которое было только что присвоено, то есть занесено в переменную слева от присваивания. Например, в Си допустима такая конструкция:

```
x = (y = 17) + 1;
```

Здесь в переменную `y` заносится значение `17`, это же значение оказывается значением выражения `(y = 17)`, к нему прибавляется единица, получается `18`, что и заносится в `x`.

Другой классический пример присваивания в качестве операции выглядит так:

```
a = b = c = d = e + 5;
```

Здесь будет взято значение переменной `e`, к нему прибавят `5`, результат будет занесён в `d`, причём результатом выражения `d = e + 5` станет только что присвоенное выражение, его в результате занесут в `c`, результатом выражения `c = d = e + 5` станет всё то же значение, и оно таким же образом попадёт в `b`, потом в `a`.

Чтобы понять, почему такие странные возможности проникли в язык, нужно, во-первых, припомнить, что Си изначально придуман как заменитель языка ассемблера, и, во-вторых, принять во внимание, что оптимизаторов в те времена ещё не было. Очевидно, что если в программе встречается присваивание `x = y`, то на выходе сгенерируется что-то вроде

```
mov     eax, [x]
mov     [y], eax
```

В отсутствие оптимизатора каждый оператор транслируется в объектный код по заданной для этого оператора схеме без учёта соседних операторов. Например, для последовательности

```
a = x;
b = x;
c = x;
```

компилятор выдаст

```
mov     eax, [x]
mov     [a], eax
mov     eax, [x]
mov     [b], eax
mov     eax, [x]
mov     [c], eax
```

Очевидно, что две команды тут лишние, ведь присваиваемое значение уже лежит в `EAX`, и загружать его из `x` трижды — совершенно бессмысленно. В более общем смысле можно заметить, что *присвоенное значение остаётся в аккумуляторе*, и если следующая операция использует свежеприсвоенное значение (а так происходит довольно часто: следующая операция может, например, обращаться к только что присвоенной переменной), можно сэкономить объём машинного кода и время исполнения, если заставить компилятор использовать значение, которое уже имеется где надо, вместо того чтобы его туда заносить. Конструкция вида `a = b = c = x` позволяет сократить код:

```
mov     eax, [x]
mov     [c], eax
mov     [b], eax
mov     [a], eax
```

Конечно, в современных условиях это уже не столь важно, оптимизатор такое сокращение кода произведёт без каких-либо подсказок с нашей стороны, но присваивание в роли именно операции (а не оператора) открывает очень интересные возможности при записи циклов. Об этом речь пойдёт, когда мы доберёмся до управляющих конструкций языка.

Кроме операции простого присваивания, в Си предусмотрены операции присваивания переменной значения, которое получено из её же старого значения путём выполнения некоторого действия. Например, `a += 7` означает «присвоить переменной `a` её же значение, увеличенное на 7».

Часто можно встретить утверждение, что `a += b` — это *сокращённая запись* операции `a = a + b`, но это, вообще говоря, не так. Дело в том, что *переменная* (точнее, как говорят в формальных текстах, ***леводопустимое выражение***, то есть то, что стоит или может стоять слева от знака присваивания) может иметь довольно сложную форму. Мы пока не рассматривали работу с массивами, но поскольку операция индексирования в Си выглядит очень похоже на аналогичную операцию Паскаля, следующий пример будет понять несложно. Итак, допустим, у нас есть массив `v` и некая функция `f()`, возвращающая целое число. Рассмотрим две записи:

```
v[f(x)] += 10;
v[f(x)] = v[f(x)] + 10;
```

Ясно, что это совершенно не одно и то же, ведь в первом случае `f()` будет вычислена один раз, тогда как во втором случае она будет вычисляться дважды. То обстоятельство, что двойное вычисление *неэффективно* (вхолостую расходуется процессорное время) — это ещё полбеды, на это при определённых обстоятельствах можно было бы не обратить внимания. Гораздо серьёзнее всё становится, если *функция `f()` имеет побочные эффекты*; в качестве самого простого и безобидного примера такой ситуации можно назвать наличие внутри `f()` операции вывода, например простой надписи на экран — в первом случае такая надпись появится один раз, во втором — дважды. А совсем всё станет странно, если окажется, что *возвращаемое значение функции `f()` различается от раза к разу даже при одинаковом значении аргумента*; в этом случае значение будет извлечено из одного элемента массива, а занесено — совсем в другой, причём такой эффект оказывается полной неожиданностью для читателя программы (и чаще всего для её автора тоже).

Аналогично операции `+=` работают `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=` и `>>=`: в каждом случае из стоящей слева переменной извлекается значение, над этим значением и правым операндом производится операция, символ которой стоит перед знаком равенства, а результат операции заносится обратно в переменную. Чтобы понять, откуда в Си взялись эти операции, достаточно припомнить язык ассемблера и команды `ADD`, `SUB`, `AND`, `OR` и т. п.

Слегка особняком в списке операций стоят ***инкремент*** и ***декремент***, которые обозначаются соответственно двумя плюсами и двумя минусами. Это унарные операции, обычно применяемые к целочисленным переменным и предписывающие увеличение (уменьшение) значения переменной на единицу; иначе говоря, `++i` означает «занести в переменную `i` значение, на единицу большее, чем то, что там сейчас», а `--i` — то же самое, только значение заносится на единицу *меньшее*. В такой

форме эти операции означают то же самое, что и `i += 1` и `i -= 1`, но, как водится в языке Си, это только начало.

Продолжение состоит в данном случае в том, что у обеих операций наряду с обычной *префиксной* формой (то есть когда символ операции записывается перед переменной) присутствует ещё и *суффиксная форма*, то есть можно написать также `i++` и `i--`. В обоих случаях с переменной `i` происходит *ровно то же самое*, то есть она увеличивается или уменьшается на единицу; в этот момент у студентов обычно возникает вопрос, зачем же, в таком случае, вообще введены эти суффиксные формы.

Чтобы понять, в чём разница между префиксной и суффиксной формой операций инкремента и декремента, нам придётся в очередной раз вспомнить, что Си — не Паскаль и традиции здесь совершенно другие. В Паскале аналог операций инкремента/декремента присутствует в виде *встроенных процедур* `inc` и `dec`, но, будучи процедурами, они только меняют значение переменной и больше ничего не делают. Иное дело Си, где инкремент и декремент — это **операции**, то есть они могут встречаться в выражениях и, естественно, имеют некоторое значение. Различие заключается как раз в этом значении: если значением операции в префиксной форме оказывается *новое* (уже увеличенное) значение переменной, то постфиксная форма в качестве своего значения имеет *старое* значение переменной, то, которое переменная имела до выполнения операции.

Рассмотрим для примера два фрагмента:

```
x = 7;              p = 7;
y = ++x;            q = p++;
```

В результате выполнения первого фрагмента обе переменные `x` и `y` получат значение `8`, тогда как после выполнения второго фрагмента `p` тоже будет равна `8`, но `q` будет равна `7`. С декрементом ситуация полностью аналогична.

В системе команд PDP-11, с которой, как мы знаем, началась история ОС Unix и языка Си, присутствовала возможность при работе с косвенной адресацией произвести преинкремент, предекремент, постинкремент или постдекремент того регистра, из которого брался косвенный адрес. Это позволяет лучше понять, откуда взялись в Си операции `++` и `--`.

Ещё несколько операций

Теперь, пожалуй, настало время для настоящей экзотики. Начнём с замечания, что в Си арифметическими операциями считаются (и действительно являются) ***операция вызова функции***, обозначаемая круглыми скобками, и ***операция индексирования*** (то есть обращения к элементу массива), которая, как и в Паскале, обозначается скобками квадратными. Подробное рассмотрение операции индексирования отложим до параграфа, посвящённого массивам; про операцию вызова

функции отметим, что первым её операндом является *имя функции* либо *адрес функции*[27], а её «арность» (то есть количество операндов) может быть любым, но не меньше одного. Так, при вызове функции без параметров операция её вызова оказывается унарной, и выглядит это как имя функции, после которого указаны пустые скобки: `f()`. Подчеркнём, что, в отличие от Паскаля, в Си эти скобки обязательно указываются, даже когда параметров нет, ведь именно скобки обозначают для компилятора *вызов функции*; имя функции без скобок воспринимается компилятором как *адрес* функции, а не её вызов.

При вызове функции с параметрами арность операции вызова составит количество параметров функции плюс один; например, при вызове функции от трёх параметров арность операции вызова — четыре: для выражения `f(a, b, c)` можно сказать, что операндами операции `()` стали `f`, `a`, `b` и `c`.

Следующей экзотической операцией мы рассмотрим *тернарную* (то есть имеющую три аргумента) ***условную операцию***. Записывается она в виде трёх выражений, после первого ставится вопросительный знак, после второго — двоеточие, чтобы отделить его от третьего. Первое выражение вычисляется и рассматривается как логическое значение («истина» или «ложь»), и если получилась «истина», то вычисляется второе выражение, а если «ложь» — то третье. Например, мы могли бы написать:

```
x = cond(y) ? pro(y) : contra(y);
```

В этом случае сначала будет вызвана функция `cond(y)`, и если она вернёт «истину» (что угодно кроме нуля) — то в переменную `x` попадёт результат вычисления `pro(y)`, а `contra` вообще не будет вызываться; если же `cond(y)` вернёт «ложь», то есть ноль, то всё будет наоборот: `pro` вызываться не будет, а результат вычисления `contra(y)` будет занесён в `x`.

Более популярный пример условной операции выглядит так:

```
max = a > b ? a : b;
```

Здесь в переменную `max` заносится большее из двух значений — `a` и `b`. Этот пример лаконичнее, но он не даёт возможности пояснить, что из второго и третьего операндов выбирается один, а второй при этом не только не выбирается, но и, что очень важно, *не вычисляется*; на предыдущем примере это видно более наглядно.

[27] Есть точка зрения, что в Си имя функции и есть её адрес, но есть и противоположная точка зрения, и самое занятное, что семантика некоторых операций поддерживает *обе* точки зрения, не давая возможности понять, какая из них правильная. Подробное рассмотрение этого момента будет приведено вместе с описанием указателей на функции.

Чтобы понять, зачем такая операция нужна, следует сообразить, что она, естественно, может применяться не только в правой части присваивания, но и, например, в качестве параметра функции:

```
printf("Maximum value is %d\n", a > b ? a : b);
```

Более сложный пример может оказаться непонятным; в этом нет ничего страшного, его можно сейчас пропустить и вернуться к нему после изучения оператора `for`. Пусть у нас есть массив целых чисел `vec`, а его размер хранится в переменной `vsize`, и от нас требуется выдать его элементы на стандартный вывод, разделив их запятыми и пробелами, а в конце поставить перевод строки. Это можно, например, сделать так:

```
for(i = 0; i < vsize; i++)
    printf("%d%s", vec[i], (i < vsize-1) ? ", " : "\n");
```

Пожалуй, самая экзотическая операция языка Си — это операция «запятая». Выражение вида `a,b` вычисляется так: сначала вычисляется `a`, потом вычисляется `b`, а значением всего вместе становится результат вычисления `b`. К тому, как и где эта операция может оказаться полезной, мы ещё вернёмся; с другой стороны, этой операции в языке вполне могло не быть, поскольку во **всех** случаях, когда её используют, без неё можно легко обойтись. При этом сам символ «запятая», например, разделяет параметры в вызове функции, и там компилятор воспринимает этот символ не как символ операции, а как простой синтаксический разделитель. Остаётся только припомнить нашу мантру: нет, Си мы любим не за это.

Отдельного рассмотрения заслуживает ***операция преобразования типа выражения***. Если `expr` — это некоторое выражение, а `type` — некий тип, то конструкция вида `(type)expr` означает значение `expr`, *тип которого принудительно знаменён на* `type`. Пусть, например, у нас имеется целочисленная переменная `n`; выражение `(double)(n+1)/3.0` означает, что к значению `n` будет прибавлена единица, результат будет преобразован в число с плавающей точкой (типа `double`) и поделен на `3.0`, причём деление будет «настоящее» (не целочисленное), так как его операнды — числа с плавающей точкой.

Чаще всего операцию преобразования типа применяют для адресных выражений и указателей, но об этом у нас разговор ещё впереди.

Операции, связанные с адресами и указателями

В завершение обсуждения операций перечислим те из них, для которых время подробного описания ещё не пришло, поскольку мы пока не знакомы ни с указателями, ни со структурными типами. Ко всем этим операциям мы обязательно вернёмся, здесь же мы приводим их исключительно для полноты картины:

- операция взятия адреса обозначается амперсандом «&», а от побитового «и» отличается количеством операндов — она унарная, в отличие от бинарного побитового «и»; например, выражение `&x` означает *адрес* переменной `x`;
- обратная операция — операция разыменования[28], обозначается звёздочкой «*»; от операции умножения её тоже отличают по количеству аргументов; например, если `p` — это некий адрес, то `*p` — это то, что расположено в памяти по этому адресу;
- операция выборки поля из структуры или объединения обозначается точкой (что-нибудь вроде `item1.next`);
- операция выборки поля из структуры или объединения по имеющемуся *адресу* обозначается стрелкой `->`; например, `a->b` есть то же самое, что и `(*a).b`.

О приоритетах операций

Большинство книг, посвящённых языку Си, содержат полную таблицу приоритетов операций, и находятся люди, вызубривающие эту таблицу наизусть; в основном это студенты-троечники, которым обязательно надо убедить хотя бы самого себя, что усилий на учёбу потрачено «достаточно», а если ничего не получилось, то виноват предмет, а не студент; кроме того, в вызубривании таблицы приоритетов часто оказываются замечены классические «унылые отличники», у которых в школе выработалась вредная привычка зубрить всё, что попалось в учебнике. Разумеется, для практического программирования знание этой таблицы не только не обязательно, но и, пожалуй, бесполезно. Достаточно помнить всего несколько несложных правил;

- приоритет умножения и деления выше, чем сложения и вычитания; по той же причине приоритет конъюнкции выше приоритета дизъюнкции, ведь конъюнкция аналогична умножению, а дизъюнкция — сложению;
- приоритет сравнений ниже, чем приоритет арифметики, а приоритет логических связок ниже, чем приоритет сравнений; в частности, в выражении `c >= 'a' && c <= 'z'` скобки не нужны (в отличие, кстати, от Паскаля);
- приоритет любой унарной операции выше, чем приоритет любой бинарной;
- приоритет *суффиксных* операций выше, чем приоритет префиксных; например, в выражении `++a[7]` сначала к `a` применяется индексирование, а к его результату (то есть к полученному элементу массива) применяется `++`;
- приоритет тернарной условной операции ниже всего перечисленного, а приоритет всех операций присваивания — ещё ниже, и в самом низу таблицы гордо располагается операция «запятая»;

[28] Англ. *dereference*.

если не обращать внимания на «запятую», то в операндах тернарной операции можно использовать (без скобок) любые выражения, не содержащие присваиваний, а в правой части присваивания, опять же без скобок, поставить вообще любое выражение (кроме содержащих запятую; но выражения, в которых запятая всё-таки содержится, лучше не применять, ясности программе они не добавят).

4.3.6. Операторы языка Си

Количество *операторов* в языке Си гораздо меньше, чем в Паскале, потому что все возможности, которые только могут быть вытеснены из языка в библиотеку, реализованы именно в библиотеке. Как уже говорилось, в Си (самом по себе) отсутствуют какие бы то ни было средства ввода-вывода, а всевозможные `printf`'ы и `scanf`'ы — это библиотечные функции, которые сами написаны на Си. Это позволило создателям языка ограничиться всего *одиннадцатью* операторами, которые мы сейчас перечислим.

Читателям, привыкшим к Паскалю, ещё раз рекомендуется обратить внимание на то, что, в отличие от Паскаля, символ точки с запятой в Си считается не *разделителем* операторов, а *частью синтаксиса самих операторов* (притом не всех). Иначе говоря, если синтаксисом оператора предусмотрена точка с запятой, то она будет нужна там вне всякой зависимости от контекста; если же она не предусмотрена — то её там не будет (это значит, что компилятор без всякой точки с запятой знает, как распознать конец такого оператора).

Ещё одно общее отличие от Паскаля состоит в том, что ключевых слов, которые ставятся между условием и телом (подобно паскалевским `do`, `then` и `of`) в Си не предусмотрено; создателям Си пришлось придумать другой способ обозначения конца условного выражения: это делается путём заключения его в круглые скобки, которые здесь (в отличие от Паскаля) обязательны.

Вычисление выражения

Чаще других в программах на Си встречается ***оператор вычисления выражения ради побочного эффекта***. Его синтаксис выглядит очень просто: это произвольное арифметическое выражение, после которого стоит точка с запятой — она как раз и превращает *выражение* (которое могло бы быть частью другого выражения) в законченную конструкцию — оператор. Роль у этой точки с запятой двойная: во-первых, она сообщает компилятору, что конструкция закончилась и можно больше не ожидать появления знаков операций; во-вторых, она

показывает, что для данного выражения нас совершенно не интересует его значение.

В принципе, конструкции вида

```
x + 5;
```

или даже просто «`5;`» с точки зрения языка Си оказываются вполне легитимными, хотя и бессмысленными, ведь здесь нет никакого побочного эффекта. Оптимизатор просто выбросит соответствующий код, а компилятор, если его запустили с флагом `-Wall`, выдаст предупреждение, что данный оператор «не имеет эффекта» (*has no effect*) — он так поступает всегда, когда конструкция не нарушает формальных правил языка, но есть основания сомневаться, что программист имел в виду именно это. Конечно, оператор вычисления выражения придуман для совершенно иных ситуаций. Одна из них — это обыкновенное присваивание:

```
x = 17;
```

Из предыдущего параграфа мы узнали, что присваивание является операцией, так что `x = 17` — это не более чем выражение, которое могло бы встретиться в составе более сложных выражений и которое, кстати говоря, имеет значение; символ точки с запятой указывает, что нас не интересует это значение, а выражение являет собой завершённую конструкцию. Подчеркнём ещё раз, **это никакой не «оператор присваивания», такого в Си просто нет**. Это оператор вычисления выражения ради побочного эффекта, а само выражение построено на основе бинарной арифметической операции — присваивания.

Вторая часто встречающаяся ситуация — это вызов функции ради побочного эффекта. Такова, например, хорошо знакомая нам строка

```
printf("Hello, world\n");
```

Выглядит это совсем не похоже на вышеприведённое присваивание, и тем не менее это **тот же самый оператор** вычисления выражения. В качестве выражения здесь выступает бинарная версия вызова функции (первый аргумент — `printf`, второй аргумент — строковый литерал `"Hello, world\n"`). Мы даже знаем, что у этого выражения есть значение — функция `printf` в результате такого вызова вернёт число `13`, но нам это число не нужно, о чём мы и сообщаем компилятору, поставив точку с запятой.

Отметим, что выражение может быть пустым, в результате чего получится самый короткий оператор языка Си — «пустой оператор», состоящий из одной только точки с запятой. Его часто используют в качестве тела цикла, когда всё, что должен делать цикл, оказалось в заголовке (в Си такое происходит сплошь и рядом). Впрочем, уж если вам понадобился пустой оператор, гораздо лучше сделать его

в виде *пустого блока* «`{}`», чем в виде неприметной и сливающейся с окружающим текстом точки с запятой. Пустые фигурные скобки гораздо лучше видно в тексте программы, и не возникает никаких сомнений, что автор программы имел в виду именно пустой оператор, а не что-то другое.

Составной оператор (блок); локальные переменные

Как и в Паскале, в Си часто (и ровно по тем же причинам) возникает потребность объединить несколько операторов в один; для этого используется ***составной оператор***, который, как мы уже успели понять (и, возможно, даже привыкнуть к этому), обозначается фигурными скобками. Но в сравнении с Паскалем составной оператор языка Си — средство более мощное, поскольку в Си составной оператор является полноценным ***блоком*** — таким фрагментом программы, для которого возможны локальные описания[29].

Локальные переменные (и другие локальные имена, о которых мы узнаем позже) описываются в самом начале блока, сразу после открывающей фигурной скобки[30] и до появления в блоке первого оператора. Напомним, что синтаксис описания переменных в Си в простейшем случае[31] предполагает имя типа, вслед за которым идёт имя переменной, либо несколько таких имён, разделённых запятой. Описания локальных переменных мы уже видели в примере программы, решающей квадратное уравнение, но там переменные описывались в начале функции; теперь мы знаем, что это можно делать в начале *любого составного оператора*.

К примеру, при реализации сортировки массива «пузырьком» нам нужна переменная, через которую мы меняем местами значения в элементах массива, если эти элементы стоят не в том порядке. В Паскале мы эту переменную вынуждены были описывать как локальную переменную соответствующей процедуры. В Си это можно сделать «более локально», прямо в теле `if`'а, например:

```
if(a[i] > a[i+1]) {
    int t;
    t = a[i];
```

[29] Напомним, что в Паскале блоки встречаются только в процедурах и функциях; собственно говоря, паскалевская подпрограмма представляет собой заголовок и блок, а блок, в свою очередь, состоит из секции описаний и секции операторов, последняя обрамляется традиционными «операторными скобками».

[30] Начиная со стандарта C99, это ограничение было снято, а описывать переменные стало возможно в любом месте программы; настоятельно не рекомендуется этим пользоваться — эта возможность не стоит того, чтобы ради неё начать использовать C99 вместо традиционного языка Си.

[31] В общем виде синтаксис описания переменной в Си чрезвычайно сложен — настолько, что мы вообще не будем приводить его полностью, а отдельные его возможности будем вводить постепенно.

```
            a[i] = a[i+1];
            a[i+1] = t;
            flag = 1;
        }
```

Переменная `t` нигде больше не нужна, но в Паскале нам приходилось терпеть тот факт, что она видна во всём теле функции. В приведённом примере мы ограничили область видимости переменной `t` так, что её не видно нигде, кроме того фрагмента, в котором она непосредственно используется.

Ветвление

С оператором `if` мы уже неоднократно встречались в примерах, так что нам осталось сказать про него не так много. Его общий синтаксис таков:

`if (` *<условие>* `)` *<оператор 1>* [`else` *<оператор 2>*]

Как это обычно происходит в Си, условное выражение обязательно должно быть заключено в круглые скобки. Тип условного выражения должен быть либо целочисленным, либо адресным; арифметический ноль и нулевой указатель означают «ложь», всё остальное — «истину». Если результатом вычисления условного выражения стала «истина», выполняется *<оператор 1>*, в противном случае выполняется *<оператор 2>*, если таковой есть; его может не быть, поскольку ветка `else` необязательна.

Следует обратить внимание на то, что **синтаксис оператора `if` сам по себе не предусматривает точки с запятой**, но её могут предусматривать *<оператор 1>* и *<оператор 2>*. Наличие или отсутствие ветки `else` никак не влияет на наличие или отсутствие точки с запятой в *<операторе1>*, на это влияет только синтаксис самого этого оператора; после Паскаля некоторое время приходится привыкать к тому, что точка с запятой перед `else` — это не ошибка.

Циклы

Простейший вариант цикла — обыкновенный цикл с предусловием, который выглядит в Си почти так же, как в Паскале (даже ключевое слово используется то же самое — `while`), только условие приходится брать в круглые скобки и при этом нет слова `do` или каких-либо его аналогов. Точнее говоря, синтаксис цикла с предусловием в Си такой:

`while (` *<условие>* `)` *<оператор>*

Как и следовало ожидать, этот оператор предполагает вычисление условия, выход, если оно ложно, если же оно истинно — выполнение оператора и снова вычисление условия, и так пока условие не станет ложным.

Цикл с постусловием в Си, который называют обычно просто *do-while*, от паскалевского repeat-until отличается довольно заметно. Формально его синтаксис таков:

`do` *<оператор>* `while ( <условие> ) ;`

Пожалуй, самое заметное его отличие от паскалевского варианта — в семантике *<условия>*: здесь это *условие продолжения* работы цикла, тогда как в Паскале цикл с постусловием предполагает выражение для *условия завершения* цикла. Иначе говоря, цикл *do-while* выполняется так: сначала выполняется *<оператор>*, затем вычисляется *<условие>*, **и если оно истинно, выполнение продолжается с начала**, то есть снова выполняется *<оператор>*.

Второе отличие состоит в подходе к телу цикла. В Паскале используется тот факт, что слова, обозначающие начало и конец цикла, *уже есть*, и поэтому между ними можно заключить сколько угодно операторов, не используя операторные скобки. В Си на первый взгляд обозначение начала и конца тоже уже есть, но это не помогает, поскольку в конце используется слово `while`, которое может с тем же успехом обозначать начало другого (вложенного) цикла, на этот раз с предусловием; приходится предполагать, что телом, как и в других сложных операторах, будет *один* оператор. Иной вопрос, что в реальной жизни тело цикла *do-while* **всегда** записывают в виде составного оператора, некоторые программисты даже уверены, что это требование синтаксиса языка. Более того, единственная возможность исключить путаницу между двумя случаями употребления ключевого слова `while` состоит в том, что **при записи цикла *do-while* заключительное слово `while` оставляют на одной строке с закрывающей фигурной скобкой от тела**, примерно так:

```
do {
    get_event(&event);
    res = handle_event(&event);
} while (res != EV_QUIT_NOW);
```

Следует также обратить внимание на то, что оператор *do-while* оканчивается точкой с запятой, она является частью его синтаксиса.

Кроме обычных циклов с предусловием и постусловием в Си также присутствует цикл `for`, но, в отличие от Паскаля, где одноимённый цикл просто называют арифметическим, здесь эта конструкция *может, но не обязана* представлять арифметический цикл. Неподготовленному читателю формальное описание цикла `for` может показаться совершенно непонятным; мы пойдём традиционным для таких ситуаций путём — сначала приведём примеры и лишь затем дадим формальное описание.

Допустим, у нас имеется целочисленная переменная `i`. Следующий цикл напечатает квадраты целых чисел от 1 до 25:

```
for(i = 0; i <= 25; i++)
    printf("%d x %d\t = %d\n", i, i, i*i);
```

Это действительно выглядит как арифметический цикл, и, собственно говоря, это он и есть. Однако наваждение начинает развеиваться, если посмотреть на следующий пример:

```
for(a = x, b = y; a; a %= b)
    if(a < b) {
        int t;
        t = a;
        a = b;
        b = t;
    }
```

Как уже, возможно, догадался читатель, этот цикл вычисляет наибольший общий делитель чисел `x` и `y`, используя для этого переменные `a` и `b` и применяя алгоритм Евклида. Перед началом работы в переменные `a` и `b` заносятся значения `x` и `y`; цикл выполняется, пока выражение `a` *истинно* (то есть не ноль); если `a` оказалось меньше `b`, то в теле цикла производится обмен значений этих переменных через временную переменную `t`, и, наконец, в `a` заносится остаток от деления её старого значения на `b`. Результат — искомый наибольший общий делитель — оказывается в переменной `b`.

Окончательно забыть об арифметических циклах позволяет следующий пример:

```
for(n = 0, c = getchar(); c != EOF; c = getchar())
    if(c == '\n') {
        printf("%d\n", n);
        n = 0;
    } else {
        n++;
    }
```

На всякий случай отметим, что такой стиль программирования мы никоим образом не приветствуем и привели этот фрагмент лишь в качестве примера того, что можно сделать с циклом `for`, если не проявлять разборчивости в средствах. Поясним, что функция `getchar` прочитывает из стандартного потока один символ и возвращает его код, либо возвращает особое значение `EOF` (на самом деле, это просто `-1`), если прочитать символ не удалось. Цикл читает символы из потока стандартного ввода и каждый раз при получении символа конца строки печатает длину полученной строки.

Как видим, в заголовке цикла `for` имеются три секции, разделённые символом точки с запятой; в каждой из них можно записать выражение. Первое из выражений (*инициализация*) вычисляется перед началом

работы цикла; обычно здесь записывается присваивание начального значения переменной цикла, но вообще никто не мешает написать тут что угодно. Второе выражение представляет собой *условие* продолжения цикла (как в цикле `while`). Третье выражение (*итерация*) вычисляется каждый раз *после* выполнения тела цикла, то есть непосредственно перед очередным (но не первым!) вычислением условия. Формально синтаксис цикла `for` можно представить так:

`for (` [*<инициализация>*] `;` [*<условие>*] `;` [*<итерация>*] `)`
<оператор>

Заметим, что все три выражения, предусмотренные в заголовке `for`, являются необязательными и могут быть пропущены (но точки с запятой должны при этом остаться на месте). Если пропущены инициализация или итерация — считается, что в соответствующей ситуации ничего делать не надо; пропущенное условие означает, что цикл следует выполнять «бесконечно», то есть до тех пор, пока выход из него не будет произведён каким-либо альтернативным способом. В частности, если в заголовке `for` присутствует только условие, такой цикл эквивалентен циклу `while` с таким же условием. Бесконечный цикл программисты на Си традиционно записывают лаконичным `for(;;)` (читается как *forever*; читатели, знакомые с английским, могут оценить изящную игру слов).

Рассказ о цикле `for` будет неполным без упоминания операции «запятая». Описывая её (см. стр. 233), мы обещали рассказать, для чего её можно использовать, но «когда-нибудь потом». Так вот, благодаря наличию операции «запятая», а также тому, что части заголовка `for` разделяются точкой с запятой (отметим, что это *единственный* случай в Си, когда точка с запятой находится не в конце оператора, а в самой его сердцевине), в цикле `for` можно использовать *больше одной переменной цикла*. Например, следующий цикл «переворачивает» массив `arr` длины `arr_len`, состоящий из целочисленных элементов:

```
for(i = 0, j = arr_len - 1; i < j; i++, j++) {
    int t;
    t = arr[i];
    arr[i] = arr[j];
    arr[j] = t;
}
```

Наиболее примечателен здесь тот факт, что, судя по всему, *именно для этого запятую сделали операцией*. Некоторые энтузиасты находят ей другие применения, но всякий раз оказывается, что такие «находки» никакого выигрыша не дают. Иначе говоря, в язык была добавлена арифметическая операция, бесполезная и бессмысленная во всех контекстах, кроме одного (но и в этом одном использующаяся редко), к тому же превращающая синтаксис выражений в сумбур, поскольку во

многих случаях запятая используется не как символ операции, а как простой разделитель. Нет, Си мы любим не за это.

Любой из циклов в любой момент может быть досрочно прекращён уже знакомым нам по Паскалю[32] оператором «`break;`» (именно так, используется ключевое слово `break` и точка с запятой, хотя, как это очень легко видеть, точка с запятой тут избыточна — но синтаксис Си именно таков).

Предусмотрен также и второй хорошо знакомый нам оператор, «`continue;`», который досрочно завершает выполнение тела цикла, сам цикл при этом продолжается со следующей итерации. В частности, в циклах `while` и `do-while` выполнение продолжается с вычисления условия (предусловия или постусловия — это здесь неважно), а в цикле `for` выполнение продолжается очередным вычислением выражения *итерация*, после которого, опять-таки, вычисляется условие. Общее правило тут такое: оператор `continue` предписывает *проигнорировать остаток тела цикла*, но остальные элементы цикла продолжают выполняться в обычном порядке.

Возврат из функции

Как мы уже неоднократно видели в примерах, возврат управления из функции производится оператором `return`. Этот оператор несёт двойную нагрузку: во-первых, он определяет значение, которое вернёт функция; во-вторых, он завершает выполнение функции, в том числе досрочно.

В зависимости от того, предусмотрен ли возврат значения из данной функции, оператор `return` принимает одну из двух форм. Для обычных функций его синтаксис таков:

`return` <*выражение*> `;`

где <*выражение*>, будучи вычисленным, даёт как раз результат выполнения функции. Что же касается функций, для которых вместо типа возвращаемого значения указано слово `void`, то в них, разумеется, никакого выражения нет, ведь они не возвращают значений. В этом случае оператор принимает лаконичную вторую форму:

`return ;`

Следует отметить, что `void`-функция замечательно может завершиться без посторонней помощи, просто дойдя до конца, так что в таких функциях `return` следует использовать, только если из функции потребовалось выйти досрочно.

Иное дело «настоящие» функции. Поскольку в Си нет иного способа указать возвращаемое значение, а не вернуть никакого значения, когда

[32] На самом деле этих операторов не было в оригинальном Паскале; как несложно догадаться, они были заимствованы, причём как раз из языка Си, так что сейчас мы, если можно так выразиться, добрались до их первоисточника.

от нас его ожидают — это явная ошибка, оператор `return` в таких функциях совершенно необходим и неизбежен; в большинстве случаев он ставится в последней строке текста функции. Интересно, что если этого не сделать, или, точнее говоря, написать текст функции так, что компилятору *покажется*, что она может дойти до конца, так и не встретив оператор `return`, и при этом её тип значения не `void`, то будет выдано предупреждение, и такое предупреждение ни в коем случае не следует игнорировать. Пользуясь случаем, ещё раз напомним читателю про флажок `-Wall` в командной строке компилятора (не будет флажка — не будет и предупреждения, и сколько крови мы на этом испортим, остаётся только гадать).

Чтобы понять, почему определение возвращаемого значения намертво сцеплено с завершением функции, стоит обратиться к конвенции вызовов функций в Си. По традиции возврат значения из функций производится через регистр, причём среди всех регистров выбирается «самый главный»; в частности, на архитектуре i386 целые числа, умещающиеся в 32 бита, возвращаются через регистр `EAX`, а любые числа с плавающей точкой возвращаются через вершину регистрового стека сопроцессора. Вспомнив систему команд i386, мы легко убедимся в том, что после занесения финального значения функции в соответствующий регистр мы больше не сможем производить вычисления: слишком уж неудобно обходиться без «аккумулятора» (а если понадобится умножать и делить, так и попросту невозможно), ну а вычисления с плавающей точкой в обход вершины стека не получится делать вообще никак. Мало того, если вызвать какую-то ещё функцию, она тоже — в соответствии с той же самой конвенцией — вернёт своё значение через те же «самые главные регистры», тем самым испортив то, что в них лежит.

Вот так вот и получается, что занесение возвращаемого значения «куда следует» *должно быть последним действием перед возвратом управления из функции*.

Но почему, можно спросить, такого не происходило в Паскале? Дело в том, что конвенция вызовов подпрограмм, использовавшаяся в классических версиях Паскаля, принципиально иная, и, в частности, возврат значения там производился через специально для этого выделенную область памяти в стеке, причём выделял её вызывающий[33]. Ясно, что занести значение в такую область памяти можно в любой момент, и оно будет там лежать, дожидаясь своего часа и никоим образом не мешая нам производить дальнейшие вычисления. Стоит отметить, что для функций, возвращающих нечто, не помещающееся в `EAX` и при этом не являющееся числом с плавающей точкой, компиляторы Си производят передачу возвращаемого значения через область памяти, чей адрес передан в функцию неявным параметром, ну а в самых ранних версиях языка значения таких типов вообще нельзя было вернуть из функции.

[33] Отметим, что Free Pascal, который мы изучали, использует для возврата значения регистры — точно так же, как это делают компиляторы Си; если присваивание, фиксирующее возвращаемое значение, делается в функции раньше её завершения, это значение хранится в области локальных переменных, в перед возвратом переносится в `EAX` или в стек сопроцессора.

Оператор `goto`

Уже знакомый нам по Паскалю ***оператор безусловного перехода*** заслуживает отдельного рассмотрения. Единственное заметное отличие от Паскаля состоит в том, что метку, на которую будет делаться `goto`, не нужно заранее описывать: вы просто используете произвольный идентификатор, не занятый ни подо что другое. Областью видимости такого идентификатора всегда является функция, «прыгать» из одной функции в другую нельзя; но вот внутри функции прыгнуть можно в том числе и *внутрь сложного оператора*, в том смысле, что компилятор не станет вам мешать вытворять подобный идиотизм; разумеется, пользоваться этим ни в коем случае не следует.

Как и в Паскале, меткой можно пометить только *оператор*, так что, если возникает необходимость поставить метку в конце составного оператора, после неё необходимо предусмотреть пустой оператор (обычно просто ставят точку с запятой). Как и в большинстве языков, где присутствуют метки, собственно метка записывается непосредственно перед оператором и отделяется от него двоеточием. Синтаксис оператора `goto` и вовсе ничем на первый взгляд не отличается от паскалевского, только что точка с запятой по традиции входит в состав оператора:

`goto` <*метка*> `;`

Изучая Паскаль, мы уже отмечали (см. т. 1, §2.6.3), что в большинстве случаев `goto` лучше не использовать, но есть два (не один, не три, а ровно два) случая, когда использование `goto` не только не запутывает программу, но, напротив, делает её более ясной. Напомним, что эти ситуации — выход из нескольких вложенных друг в друга конструкций и освобождение локально захваченных ресурсов перед досрочным завершением подпрограммы; наше рассуждение про использовние `goto` в Паскале полностью применимо и к Си.

Оператор выбора

Оператор выбора `switch`, приблизительный аналог паскалевского `case`, пожалуй, самый запутанный в Си. Идея, как и в Паскале, состоит в том, чтобы в зависимости от значения некоторого выражения выполнить одну из нескольких ветвей кода, то есть это своего рода обобщение ветвления на случай N ветвей; однако подход к построению соответствующего оператора в Си кардинально отличается.

Общий синтаксис оператора `switch` таков:

`switch (` <*выражение*> `) {` <*метки и операторы*> `}`

При этом <*выражение*> должно иметь произвольный целочисленный тип, а <*метки и операторы*> представляют собой произвольную последовательность операторов языка Си, первый из которых, а также, возможно, *некоторые* другие снабжены так называемыми `case`-метками. Эти метки бывают двух видов; обыкновенные `case`-метки состоят из

ключевого слова `case`, *целочисленной константы времени компиляции* и двоеточия, то есть имеют следующий синтаксис:

`case` <*константа*> :

Метка второго вида выглядит проще:

`default` :

Этот вариант метки обозначает «все варианты, кроме явно перечисленных»; обычно её ставят в конце оператора `switch`, хотя это и не обязательно. Подчеркнём, что константы в `case`-метках должны быть таковы, чтобы компилятор мог вычислить их во время компиляции; переменные или тем более вызовы функций здесь не годятся[34].

А теперь нам необходимо осознать ключевое отличие оператора `switch` от привычных конструкций выбора из других языков. **Операторы в теле `switch` представляют собой единую секцию операторов**, которая выполняется от метки, соответствующей вычисленному значению выражения, и **до конца тела `switch`**, если только выполнение не будет прервано с помощью какого-нибудь из операторов, передающих управление куда-нибудь ещё. Например, оператор

```
switch(t) {
case 1:
    printf("First\n");
case 2:
    printf("Second\n");
case 3:
    printf("Third\n");
case 4:
    printf("Fourth\n");
case 5:
    printf("Fifth\n");
default:
    printf("More\n");
}
```

при `t`, равном `4`, напечатает не просто `Fourth`, как обычно ожидают начинающие, а гораздо больше:

```
Fourth
Fifth
More
```

Иначе говоря, очередная `case`-метка являет собой точку, откуда выполнение тела `switch` могло бы начаться, но **никоим образом не**

[34] Более того, в языке Си присутствуют константы, описываемые как переменные, но со словом `const`, запрещающим их изменение; так вот, они тоже не считаются константами времени компиляции и не могут использоваться в `case`-метках; это объясняется тем, что такая константа может, вообще говоря, располагаться в другой единице трансляции, так что компилятор не будет знать её значение во время компиляции.

предписывает завершить выполнение операторов, если это выполнение началось выше неё.

Такая экзотическая семантика, в принципе, вполне объяснима, если вспомнить, что Си создан как замена языку ассемблера; в некоторых (достаточно редких) случаях это даже оказывается удобно, но чаще нам в программах, наоборот, хотелось бы разбить код на непересекающиеся ветви и выбрать одну из них в зависимости от значения выражения, то есть, грубо говоря, паскалевский вариант оператора выбора нас в большинстве случаев устроил бы больше. Поэтому перед каждой очередной `case`-меткой нам приходится тем или иным способом явно прекращать выполнение тела `switch`. Обычно для этого используют `break`, который внутри `switch` означает примерно то же, что и внутри циклов: прекратить выполнение тела, передав управление в конец оператора `switch`; например:

```
switch(t) {
case 1:
    printf("First\n");
    break;
case 2:
    printf("Second\n");
    break;
case 3:
    printf("Third\n");
    break;
case 4:
    printf("Fourth\n");
    break;
case 5:
    printf("Fifth\n");
    break;
default:
    printf("More\n");
}
```

Кроме того, досрочно завершить `switch` можно, сделав `goto` «наружу», вернув управление из функции с помощью `return`, а при наличии объемлющего цикла — выполнив `continue` для досрочного прекращения его итерации. В реальной жизни чаще других из всех этих способов применяется как раз `return`.

Следует сказать несколько слов об оформлении фрагментов кода, содержащих `switch`. Прежде всего отметим, что в вышеприведённых примерах мы не сдвигали метки относительно самого оператора; в принципе, можно их и сдвигать, например так:

```
switch(t) {
```

```
        case 1:
            printf("First\n");
            break;
        case 2:
            printf("Second\n");
            break;
        default:
            printf("More\n");
    }
```

Но главное тут в другом. Оператор выбора, будучи применён в реальной практике, обычно имеет тенденцию распухать на несколько десятков строк, делая функцию, в которой он располагается, совершенно нечитаемой. С этим можно бороться несколькими способами:

- старайтесь выносить `switch` целиком в отдельную функцию или хотя бы оставлять вместе с ним в одной функции как можно меньше другого кода;
- если отдельные секции `switch` поддаются оформлению в виде обособленных функций, обязательно сделайте это, оставив в теле самого `switch` только метки, вызовы этих функций и `break`'и;
- **никогда не размещайте один `switch` внутри другого**.

4.3.7. Локальные «статические» переменные

Мы уже знакомы с *глобальными* и *локальными* переменными; напомним, что глобальные переменные описываются вне тел функций и видны во всём тексте исходного файла, начиная от точки их описания (или объявления), тогда как локальные переменные описываются в начале любого блока — тела функции или составного оператора — и видны от точки описания до конца того блока, в котором переменная описана, то есть до закрывающей фигурной скобки.

Кроме ***области видимости***, переменные характеризуются ещё ***временем существования***, то есть временны́м периодом от того момента, когда переменная создаётся (под неё выделяется область памяти), и до момента, когда ранее выделенная под переменную область памяти высвобождается. Для глобальных переменных время существования совпадает с временем выполнения программы, а локальные переменные, располагающиеся в стеке, начинают существовать при входе в блок, в котором они описаны, и исчезают при выходе из этого блока (точнее говоря, они возникают и исчезают в момент соответствующего изменения регистра «указатель стека»).

В языке Си предусмотрен довольно странный гибрид локальных переменных с глобальными — так называемые локальные статические переменные. Как и простые локальные переменные, статические переменные описываются в начале блока и видны в тексте программы от

точки описания до конца блока. С другой стороны, их время существования, как и у глобальных переменных, совпадает со временем работы программы, а размещаются они, как и глобальные переменные, в секции `.data` или `.bss` в зависимости от того, задан для них инициализатор или нет; если инициализатора нет, статическая переменная инициализируется нулём. Описание локальной статической переменной отличается от описания обычной локальной переменной наличием слова `static`, например:

```
int my_function(int x, int y)
{
    int tt = 25;
    static int zz = 75;

    /* ... */
}
```

В этом примере `tt` — обыкновенная локальная переменная, она заводится заново каждый раз, когда начинается исполнение функции `my_function`, и получает начальное значение `25`; если функция будет прямо или косвенно вызывать сама себя (напомним, это называется рекурсией), то переменная `tt` может одновременно существовать в нескольких экземплярах — по числу вызовов `my_function`, которые уже состоялись, но ещё не завершились. В момент завершения работы функции соответствующий экземпляр переменной `tt` исчезает вместе со всем стековым фреймом.

Совершенно иначе ведёт себя переменная `zz`: вне всякой зависимости от того, вызывается ли функция `my_function` или нет, переменная `zz` с самого начала работы программы уже существует, имеет значение `75` и готова к работе, причём по окончании работы функции она никуда не исчезает; иначе говоря, она **сохраняет своё значение между вызовами функции**. Например, функция

```
void print_next_number()
{
    static int n = 0;
    printf("%d\n", n);
    n++;
}
```

при каждом её вызове печатает следующее целое число, то есть при первом вызове печатает `0`, при втором вызове — `1` и так далее.

Следует отметить, что локальные статические переменные обладают практически всеми недостатками обычных глобальных переменных, так что перед их использованием следует хорошо подумать. Если вы не совсем уверены, что понимаете, для чего нужно применение локальных

статических переменных, то не применяйте их вовсе; ваша программа от этого хуже не станет.

На всякий случай отметим ещё один момент, связанный со словом `static`. Этот модификатор можно применять не только к локальным, но и к глобальным описаниям, как к переменным, так и к функциям. При этом значение слова `static` становится совершенно иным, не имеющим ничего общего с вышеописанными статическими локальными переменными: для глобальных описаний `static` означает, что они не будут видны из других модулей программы. Подробно мы этот момент обсудим в главе, посвящённой раздельной трансляции.

4.4. Указатели, массивы, строки

Работа с массивами в языке Си организована в виде ***арифметики адресов***; её часто называют арифметикой указателей, что не совсем верно. Так или иначе, прежде чем рассматривать массивы, необходимо освоить работу с указателями.

Напомним, что наш базовый вычислитель — это машина фон Неймана, одним из основных свойств которой является линейность и однородность оперативной памяти, а отдельные ячейки памяти идентифицируются *адресами*. То, что в языках высокого уровня[35] называют ***переменной***, на уровне машинного кода представляет собой не более чем *область памяти*, то есть несколько ячеек памяти, расположенных подряд, или, иначе говоря, имеющих последовательные адреса. Под ***адресом области памяти*** понимается наименьший из адресов ячеек, составляющих область. Для нас здесь важно то, что любая переменная имеет свой адрес.

Строго говоря, компилятор может расположить локальную переменную в регистре, тогда адреса у неё не будет; но он так поступает лишь в случае, если мы никак не можем этого обнаружить.

Во многих языках программирования, включая Паскаль и Си, адреса считаются информацией, которую можно хранить и обрабатывать; но если при работе на языке ассемблера адреса ничем не отличаются от обычных чисел, то языки высокого уровня вводят для адресов отдельные типы данных. Как и в Паскале, в языке Си адресный тип привязан к типу переменной, адрес которой имеется в виду.

Напомним два базовых принципа, которые мы уже формулировали в первом томе при рассмотрении указателей Паскаля:

Указатель — это переменная,
в которой хранится адрес.

Утверждение вида «A указывает на B»
означает «A содержит адрес B».

[35] Во всяком случае, в *императивных* языках, но других мы пока не рассматривали.

Собственно говоря, эти принципы общие, они справедливы не только для Си. А теперь приступим к делу.

4.4.1. Указатели и операции над ними

Адресный тип описывается в Си с помощью символа звёздочки, которая ставится после типа переменной, адрес которой описывается; например, «`int *`» означает *адрес* ***`int`***, а «`double *`» — *адрес* ***`double`***. Соответствующим образом описываются и переменные-указатели:

```
int *p;
double *q;
```

Отдельно стоит сказать о положении символа «`*`» в описании. Сама «звёздочка» относится к символам-разделителям, то есть выделять её пробелами не обязательно, и с точки зрения компилятора совершенно не важно, с какой стороны мы поставим пробелы и поставим ли их вообще. Собственно говоря, нам было бы совершенно всё равно, как писать объявления указателей, если бы не вариант описания, в котором перечисляются через запятую несколько имён переменных; и здесь внезапно оказывается, что звёздочка, хоть и имеет отношение к типу, воспринимается компилятором как атрибут описываемого имени, а не как часть типа. Например, если нам нужно описать три указателя на `char`, это выглядит так:

```
char *s1, *s2, *s3;
```

Начинающие программисты, присоединяя звёздочку к типу, а не к имени (что, вообще говоря, вполне логично), часто делают довольно характерную ошибку: пишут что-то вроде

```
char* s1, s2, s3;
```

предполагая, что все три переменные получат тип `char*`; на самом же деле при этом такой тип получит только `s1`, а остальные две окажутся типа `char`.

Если вы чувствуете себя неуверенно, просто описывайте каждый указатель на отдельной строке, не перечисляя их через запятую. Много места это не займёт, а текст станет яснее.

Основные операции, связанные с адресами — это ***операция взятия адреса***, обозначаемая символом «`&`», и обратная к ней ***операция разыменования***[36], обозначаемая всё той же «звёздочкой» «`*`». Так, если `t` — произвольная переменная, то `&t` — это её адрес; если `p` — указатель, то `*p` — это то, на что он указывает.

Например, если у нас описаны две переменные

```
int x;
int *p;
```

[36] Надо признать, что словоформа «разыменование» для русского языка выглядит несколько чужеродной и, судя по всему, противоречит традициям словообразования; но, с другой стороны, как адекватно перевести на русский оригинальное английское *dereference*?!

то мы можем занести адрес `x` в указатель `p`:

```
p = &x;
```

Теперь `*p` — это своего рода «синоним» для `x`, то есть, например,

```
*p = 27;
```

занесёт значение `27` в область памяти типа `int`, расположенную по адресу `p`, то есть в переменную `x`.

Начинающие часто путают ситуации, в которых указатель слева от присваивания снабжается или не снабжается звёздочкой; поэтому мы приведём ещё один пример. Пусть имеются следующие описания:

```
int x, y;
int *p, *q, *r;
```

Пусть теперь мы сделали присваивания:

```
x = 25;
y = 36;
p = &x;
q = &y;
```

Рассмотрим следующий оператор:

```
r = p;
```

Здесь мы *переменной `r` присваиваем значение переменной `p`*, а поскольку в переменной `p` находится адрес переменной `x`, после присваивания то же самое — адрес переменной `x` — будет находиться также и в переменной `r`, то есть она тоже будет указывать на `x`. Итак, сейчас у нас `p` и `r` указывают на `x`, а `q` указывает на `y`. Рассмотрим теперь оператор

```
*r = *q;
```

Здесь в левой части присваивание — «то, на что указывает `r`», а в правой — «то, на что указывает `q`», а поскольку они указывают (с учётом вышерассмотренного присваивания) на `x` и `y`, этот оператор занесёт в `x` текущее значение `y`, то есть число `36`. Сами указатели при этом не изменятся.

Казалось бы, в приведённом примере всё ясно; к сожалению, опыт показывает, что далеко не все, кто по тем или иным причинам изучает программирование, могут с ходу разобраться в этих присваиваниях. **Если у вас остались здесь хотя бы малейшие неясности, даже не думайте идти дальше!** В дальнейшем тексте вы попросту вообще ничего не поймёте и только зря потратите время. Попробуйте разобрать этот пример ещё раз, если же это не поможет — обратитесь с вопросами

к кому-то, кто сможет вам разъяснить происходящее. В языке Си на указателях основано буквально всё; операция взятия адреса потребовалась нам даже в программе, решающей квадратное уравнение. Если сейчас оставить «в тылу» малейшую неуверенность — уже при чтении следующего параграфа такая неуверенность превратится в фундаментальное непонимание происходящего.

Во многих источниках, посвящённых языку Си, можно встретить утверждение, что символ «*» в описании переменной-указателя имеет прямое отношение к операции разыменования. Некие зачатки логики здесь можно уловить, если заметить, что после описания

```
int x;
```

x имеет тип `int`, а после описания

```
int *p;
```

p имеет тип «указатель», но `*p` (то есть то, что занимает место переменной из предыдущего описания) — действительно имеет тип `int`.

Дело вкуса, но на личный взгляд автора этих строк такую логику никак нельзя считать безупречной. Позже, разбирая более сложные описания, мы убедимся, что звёздочка при прочтении описаний превращается в слова «указатель на», то есть она, будучи встреченной в описаниях, не указатель превращает в простую переменную, а, *напротив*, обычный тип превращает в адресный. Иначе говоря, роль звёздочки в описаниях *прямо противоположна* (!) её роли в выражениях.

Отметим, что в Си можно работать с адресами, не уточняя, на переменные какого типа эти адреса будут указывать. Соответствующий тип указателя называется `void*`; если описать указатель такого типа:

```
void *z;
```

то в переменную `z` можно будет занести совершенно любой адрес, и такое присваивание компилятор рассматривает как легитимное, не выдавая ни ошибок, ни предупреждений. Более того, разрешено также и присваивание в другую сторону, то есть любому типизированному указателю можно присвоить нетипизированный адрес.

Вспомнив указатели Паскаля, мы можем заметить, что Паскаль предусматривал специальное «адресное» значение, которое с гарантией не является адресом какой-либо ячейки или области памяти; это значение в Паскале обозначалось ключевым словом `nil`, его иногда называют «нулевым указателем», хотя это и не совсем верно: это адрес, а не указатель. В языке Си такое тоже есть, но в зависимости от компилятора и архитектуры конкретное значение такого указателя может быть разным; к счастью, стандартная библиотека предусматривает специальное обозначение для такого адресного значения: идентификатор

`NULL`[37], который всегда равен соответствующему значению для данного компилятора.

Интересно, что значение адреса можно использовать в качестве логического значения везде, где таковое требуется, в том числе в заголовках операторов ветвления и циклов; при этом «нулевой указатель» (то есть значение NULL) считается «ложью», а любой другой адрес — «истиной».

4.4.2. Массивы

Рассказ о массивах в языке Си мы начнём с парадоксального, на первый взгляд, утверждения: **в языке Си массивов нет**. Сразу после такого утверждения мы приведём пример описания массива[38]:

```
int m[20];
```

Здесь описан массив из 20 элементов типа `int`, причём эти элементы доступны с помощью операции индексирования под номерами (индексами) от `0` до `19`. Предвидя недоумение читателя (ведь только что мы заявили, что массивов нет), поясним: элементы массива доступны *каждый в отдельности*, тогда как к самому массиву как единому целому обратиться невозможно. Практически во всех ситуациях введённое в описании имя `m` обозначает не сам массив как таковой, а **адрес его первого элемента**. Поскольку элементы массива, в том числе первый из них, в данном случае имеют тип `int`, имя `m` представляет собой *константу* типа `int*`. Например, если мы опишем указатель

```
int *p;
```

присваивание `p = m;` будет не только возможно, но и не потребует от компилятора никакого преобразования типа: ведь здесь и слева, и справа от присваивания мы видим один и тот же тип `int*`. Больше того, после такого присваивания можно будет обращаться к элементам массива `m` через переменную `p` с помощью всё той же операции индексирования: `p[0]`, `p[1]`, ..., `p[19]` обозначают теперь ровно то же самое, что и `m[0]`, `m[1]`, ..., `m[19]`. С другой стороны, поскольку `m` — это адрес, к нему можно применить операцию разыменования, то есть обратиться к первому элементу массива `m` можно не только через индексирование (`m[0]`), но и иначе: `*m` — это абсолютно то же самое. Мы видим, что

[37] Этот идентификатор набирается заглавными буквами; почему это так, мы узнаем позднее из главы о макропроцессоре.

[38] Здесь и далее мы в примерах указываем размерности массивов и некоторые другие константы в явном виде, то есть в виде числа. Мы делаем так, чтобы не загромождать примеры описанием констант, что снизило бы их наглядность; тем не менее, следует помнить, что в тексте настоящей программы (в отличие от иллюстративного примера) числа в явном виде встречаться не должны, все их следует описывать константами.

индексирование в Си — это операция не над массивами (над массивами вообще нет никаких операций), а над *адресами*.

Чтобы понять, что́ на самом деле представляет собой операция индексирования, нужно пояснить ещё один важный момент: в языке Си к типизированному адресному выражению (то есть к адресному выражению любого типа, кроме `void*`) можно *прибавить целое число*, при этом адрес изменяется на размер элемента того типа, на который указывает исходный адрес; например, адрес типа `char*` при добавлении к нему единицы увеличится на единицу, а адрес типа `int*` в таком же случае — на четыре (ведь `int` занимает 4 байта). Таким образом, если в памяти размещён массив элементов одного типа, добавление единицы к адресу одного из этих элементов даст адрес следующего элемента массива. Иначе говоря, `m` — это адрес первого элемента, `m+1` — адрес второго элемента, и так далее; последний элемент нашего массива имеет адрес `m+19`. Коль скоро это адреса, то их можно разыменовывать. Получается, что к элементам массива мы можем обращаться через разыменование: `*m`, `*(m+1)`, `*(m+2)`, ..., `*(m+19)`, и это будет совершенно то же самое, что `m[0]`, `m[1]`, ..., `m[19]`.

Скажем больше: выражение `a[b]`, каковы бы ни были эти `a` и `b`, означает *то же самое*, что и `*(a+b)`. Осознать всю глубину экзотики позволяет замечание, что от перемены мест слагаемых сумма не меняется, так что вместо `m[17]` можно написать `17[m]`, и компилятор это благополучно проглотит. **Не надо так делать!** О наличии этой возможности мы рассказываем только для иллюстрации.

Выше упоминалось, что имя массива представляет адрес первого элемента *почти* во всех ситуациях. Единственная *осмысленная* ситуация, в которой имя массива соответствует всему массиву как единому объекту — это применение к имени массива псевдофункции `sizeof`: в этом случае выдаётся размер всего массива целиком. В частности, `sizeof(m)` для массива из нашего примера будет 80. Пресловутыми «стандартами» предусмотрены ещё две такие ситуации, но их достаточно сложно объяснить, а одна из них к тому же совершенно бессмысленна. Можете предполагать, что `sizeof` — единственное исключение из правил; такое предположение вам ничем не грозит.

Одним из самых заметных (и тяжёлых) следствий недоступности массива как единого объекта является невозможность присваивания массивов, которой часто не хватает, особенно при переходе с Паскаля. Как мы увидим чуть позже, строки представляют собой частный случай массивов — и тоже, разумеется, не присваиваются. Для начинающих, привыкших к высокоуровневой работе со строками в других языках, отсутствие возможности присвоить строку превращается в ночной кошмар, по крайней мере на первых порах.

Отметим ещё один интересный момент. В языке Си определена операция ***вычитания типизированных адресов***, результатом которой является целое число. Эта операция имеет смысл только при применении к адресам элементов одного массива; её результат — *расстояние*

между двумя элементами, то есть разность их индексов. Например, если мы работаем всё с тем же массивом `m` и присвоили указателю `p` адрес элемента `m[13]` (то есть выполнили `p = &m[13]` или просто `p = m + 13`), то значением выражения `p - m` будет число 13. Как можно заметить, эта операция *обратна* операции прибавления целого числа к адресу.

4.4.3. Динамическая память

С ***динамической памятью*** мы уже знакомы по Паскалю, но в Си подход к её использованию существенно отличается. Прежде всего отметим, что *в самом языке*, как водится, средств работы с динамической памятью нет: они вытеснены в библиотеку. Основными функциями считаются `malloc`, выделяющая память, и `free`, освобождающая память. Заголовки этих функций выглядят так:

```
void *malloc(int size);
void free(void *p);
```

Функция `malloc` принимает в качестве параметра размер необходимой области памяти, выделяет такую область из «кучи» и возвращает адрес начала выделенной области. Поскольку функция `malloc` не знает и не может знать ничего о том, для каких целей мы затребовали память, она возвращает нетипизированный адрес, который мы сами можем превратить в адрес того, что нам нужно. Например:

```
double *k;
k = malloc(360*sizeof(double));
```

После выполнения такого присваивания указатель `k` будет указывать на свежевыделенную область памяти, размер которой достаточен для размещения 360 элементов типа `double`. Вспомнив, что индексирование есть операция над адресами, мы обнаруживаем, что с этой областью можно работать как с обычным массивом, обращаясь к его элементам по индексам. Например, следующий цикл заполнит элементы этого массива значениями синуса с интервалом один градус:

```
for(i = 0; i < 360; i++)
    k[i] = sin((2*M_PI/360.0) * (double)i);
```

Функция `free` освобождает ранее выделенную память, делая её вновь доступной для выделения. В качестве параметра эта функция принимает **адрес, ранее возвращённый функцией** `malloc`. Например, освободить наш массив можно так:

```
free(k);
```

Размер области памяти библиотека хранит где-то у себя, так что указывать этот размер при освобождении не требуется. С другой стороны, если функции `free` случайно дать параметром что-то отличное от адреса, возвращённого `malloc`'ом, авария вам практически гарантирована, причём если программа «свалится» непосредственно при вызове `free`, можете считать, что вам крупно повезло; скорее всего, программа успеет ещё некоторое время поработать и только потом завершится с аварией, так что определить, где сделана ошибка, может оказаться весьма непросто.

Стандартная библиотека предусматривает ещё две функции для выделения динамической памяти — `calloc` и `realloc`; мы отложим их рассмотрение до главы 4.14. Отметим, что лучше избегать использования `realloc`, пока вы не научитесь уверенно менять размеры динамических массивов без её помощи.

Заголовки стандартных функций для работы с динамической памятью расположены в заголовочном файле `stdlib.h`.

4.4.4. Модификатор `const`

Ключевое слово `const`, пришедшее из языка Си++[39], используется в Си (и в Си++), чтобы *запретить изменение некоторой области памяти*. В отличие от Паскаля, в Си для *описания констант*, то есть именования неких значений, слово `const` не годится: переменные, описанные с этим модификатором, всё же остаются переменными, их просто запрещено изменять.

Обычно константные переменные описывают с инициализатором, хотя компилятор этого и не требует; например:

```
const int iteration_count = 78;
```

Подчеркнём, что введённое таким образом имя — не константа в том смысле, что `iteration_count` нельзя использовать там, где требуется *значение времени компиляции* — например, в `case`-метках оператора `switch`, при указании размерностей массивов[40] и т. п. Всё это делает подобные описания не слишком полезными.

[39] В частности, это слово принципиально не используется даже в последнем издании книги Кернигана и Ритчи.

[40] Начиная с C99, стандарты разрешают такое использование благодаря тому, что размер массива вообще «стало можно» задавать произвольным выражением, вычисляемым во время исполнения. Эта возможность называется *variable length arrays* (VLA). **Никогда, ни при каких условиях не используйте это!** Если до введения VLA имя локальной переменной всегда представляло собой константное смещение в стековом фрейме, то после их введения имена локальных переменных при переводе в машинный код могут превратиться в сколь угодно сложные арифметические выражения, для вычисления которых может не хватить регистров. В итоге VLA оказываются столь сложны в реализации, что многие компиляторы прибегают к услугам runtime-библиотеки, уничтожая таким образом главное и едва ли не единственное достоинство Си — zero runtime. Кроме того, размеры стека обычно ограничены, и бесконтрольное применение VLA резко увеличивает риск аварии,

Совсем другое дело получается при использовании модификатора `const` в описаниях (и объявлениях) указателей. Для начала отметим, что

```
const int *p;
```

описывает не *константный указатель* (то есть указатель, который нельзя изменять; сам указатель в данном случае изменять как раз очень даже можно), а *указатель на константную область памяти*. Иначе говоря, слово `const` относится здесь не к переменной `p`, а к той области памяти, на которую `p` будет указывать.

Описать неизменяемый указатель тоже можно:

```
int * const q;
```

Если вам это понадобилось, свяжитесь с автором книги и расскажите, как вы дошли до такой жизни. Отметим, что едва ли не большинство профессиональных программистов, пишущих на Си, об этой конструкции не знают, поскольку она практически никогда не нужна.

В этой своей ипостаси модификатор `const` находит очень активное применение, в особенности в описаниях формальных параметров функций. К примеру, если вы видите заголовок функции

```
void suspicious_func(int *a);
```

то вы вынуждены подозревать, что такая функция что-то сделает с переданной ей областью памяти, тогда как если параметр снабжён модификатором `const`:

```
void trustworthy_func(const int *a);
```

то вы можете считать, что тем самым автор функции заявил вам, пользователю, что изменять память, на которую указывает `a`, его функция не собирается.

По правде говоря, слово `const` ничего не гарантирует, ведь всегда можно изменить тип адресного выражения на любой другой, в том числе не предполагающий константности. При работе с низкоуровневыми языками вообще трудно говорить о каких-либо гарантиях. Так или иначе, слово `const` можно воспринимать как *обещание* не менять соответствующую область памяти, а нарушать собственные обещания, как известно, нехорошо; к тому же за соблюдением такого обещания всё-таки худо-бедно следит компилятор, так что нарушить его можно, лишь применив явно «некрасивый» приём.

связанной с переполнением стека. Сама возможность VLA превращает Си в высокоуровневый язык, в качестве которого Си не имеет смысла — в мире есть много более простых, приятных и логичных высокоуровневых языков, не обладающих недостатками Си.

4.4.5. Инициализаторы для массивов

При описании массива язык Си позволяет задать начальные значения его элементов. Массив при этом изображается в виде перечисления в фигурных скобках, а сами элементы разделяются запятыми. Например:

```
int m[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 21, 23 };
```

Обратите внимание на точку с запятой после закрывающей фигурной скобки, она здесь обязательна — это часть синтаксиса описания. Кроме того, крайне важно понимать ещё один момент: **все элементы инициализатора обязаны быть константами времени компиляции**, то есть среди перечисленных через запятую элементов не должно быть ни переменных, ни вызовов функций, ни таких выражений, которые компилятор не сможет вычислить прямо во время компиляции.

Если для массива предусмотрен инициализатор, то можно не указывать в явном виде его размерность, компилятор вычислит её сам:

```
int m[] = { 2, 3, 5, 7, 11, 13, 17, 19, 21, 23 };
```

Здесь размерность массива будет, как и в прошлом случае, составлять десять элементов, поскольку именно столько элементов предусмотрено в инициализаторе. Так сделано, чтобы избавить программиста от необходимости подсчитывать количество элементов вручную; узнать, какой размер получился, можно, поделив размер массива на размер его элемента, то есть написав `sizeof(m)/sizeof(m[0])` или `sizeof(m)/sizeof(*m)`. Например, следующий цикл распечатает все элементы массива:

```
for(i = 0; i < sizeof(m)/sizeof(*m); i++)
    printf("[%d] = %d\n", i, m[i]);
```

Отметим, что на примере инициализаторов для массивов наглядно видно: **инициализация и присваивание — это совершенно разные вещи**. Массиву *нельзя* таким способом ничего присвоить, когда он уже существует, то есть конструкция из фигурных скобок и списка значений не является в Си выражением и не может встречаться в других выражениях, в том числе и в присваивании; она может встречаться только в *описании* массива — в роли инициализатора.

Отметим, что язык допускает использование инициализаторов для локальных массивов, что-то вроде

```
int f(int x)
{
    int m[] = { 2, 3, 5, 7, 11, 13, 17, 19, 21, 23 };

    /* ... */

}
```

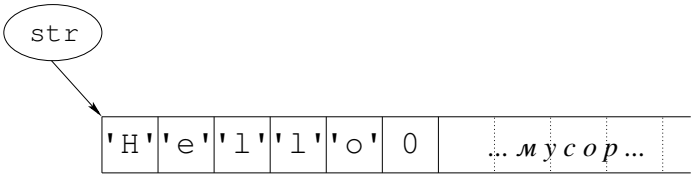

Рис. 4.1. Размещение в памяти строки `"Hello"`

Прежде чем воспользоваться этой возможностью, следует осознать, что всё это отнюдь не бесплатно: *данные будут копироваться в область память в стековом фрейме каждый раз в начале выполнения такой функции*. В некоторых случаях это не страшно, но чаще всего так делать не стоит.

4.4.6. Строки

Как мы уже знаем, ***строки*** (или, говоря шире, ***данные в текстовом представлении***) знамениты непредсказуемостью своего размера: в большинстве случаев во время написания программы мы можем лишь приблизительно прикинуть, какого количества памяти «уж точно хватит» для размещения строки, и в итоге с хорошей вероятностью обнаружим, что ошиблись. Память обычно выделяется с некоторым запасом, а во время исполнения оказывается, что строка занимает меньше памяти, чем под неё выделено; как следствие, при обработке строк необходимо соглашение, позволяющее указать, какова в настоящий момент длина строки.

В частности, при работе со строками на Паскале под хранение длины строки выделяется особый байт; если рассматривать паскалевскую строку как массив символов, то её длина хранится в элементе этого массива, имеющем индекс `0`. Как следствие, паскалевские строки не могут превышать в длину 255 символов.

Для языка Си характерен другой подход. Строки здесь тоже представляются массивами символов, то есть массивами элементов типа `char`, при этом *корректной строкой* считается только такой массив типа `char`, в котором хотя бы один элемент равен нулю; он рассматривается как *ограничитель*, то есть все элементы, предшествующие нулевому, считаются составляющими строку, а всё, что находится в массиве после нулевого элемента, игнорируется (см. рис. 4.1). Конечно, в массиве может оказаться больше одного элемента, равного нулю; в этом случае в качестве ограничителя рассматривается тот из них, индекс которого меньше, чем у других, или, иначе говоря, ближний к началу массива. Длина строки, таким образом, есть разница между положением ограничителя (нулевого элемента) и начала строки.

Если сравнивать этот подход с принятым в Паскале, в качестве его несомненного достоинства следует отметить отсутствие априорного

ограничения на длину строк: при необходимости можно обрабатывать строки длиной хоть в миллион символов (а если очень надо, то и больше, лишь бы хватило оперативной памяти). С другой стороны, очевидны и недостатки этого подхода: во-первых, для определения длины строки её приходится просматривать всю в поисках нулевого элемента, тогда как для паскалевской строки достаточно было извлечь её элемент в нулевой позиции; во-вторых, сама строка не может, очевидно, содержать символ с кодом `0`. Впрочем, считается, что «символ» с таким кодом не может встречаться в текстовых данных: в частности, если в файле встретился нулевой байт, то этот файл заведомо не текстовый.

Рассмотрим несколько примеров. Для начала напишем функцию, которая получает на вход строку и определяет её длину. Начнём с заголовка. С типом возвращаемого значения всё более-менее понятно, это ведь длина строки в символах, то есть обыкновенное целое число. Интереснее обстоят дела с параметром. На вход по условию задачи необходимо получить строку, но как это сделать? Строка — это частный случай массива символов, а мы уже знаем, что никакого способа обратиться к массиву как единому объекту язык Си не предусматривает; мы ведь по этой причине даже заявляли, что массивов в языке Си вообще нет. Тем не менее, с массивами можно работать благодаря адресной арифметике, частным проявлением которой является операция индексирования, что же до самого массива, то доступ к нему осуществляется через *адрес первого элемента*. Строка у нас состоит из элементов типа `char`, так что параметр, через который она будет передаваться в функцию, должен иметь тип «адрес char'а». Давайте, однако, не спешить с описанием параметра типа `char*`; есть ещё одно соображение, влияющее на этот тип. Очевидно, что для вычисления длины строки не нужно вносить в эту строку какие-либо изменения. Это может оказаться важным, если строка, длину которой нам надо посчитать, по тем или иным причинам относится к числу объектов, изменение которых запрещено; если не объяснить компилятору, что наша функция не собирается ничего менять, то при попытке её вызова для неизменяемого объекта будет выдано предупреждение и с этим придётся что-то делать.

Итак, нам нужно отразить в заголовке функции, что она возвращает целое число, а на вход принимает адрес `char`'а, и притом не собирается изменять область памяти по этому адресу. Назвав нашу функцию `string_length`[41], мы сможем начать её описание примерно так:

[41] В стандартной библиотеке языка Си присутствует функция `strlen`, решающая ровно эту задачу. Объявление (заголовок) этой функции находится в заголовочном файле `string.h`. Если определение длины строки потребуется вам в «настоящей» (не учебной) программе, лучше использовать функцию из библиотеки, просто потому, что так ваша программа будет понятнее стороннему читателю; но пока вы только учитесь, от использования функций из `string.h` вам настоятельно рекомендуется воздержаться, иначе вы так и не поймёте, что такое строка в Си.

```
int string_length(const char *str)
{
```

Теперь нам нужно просмотреть область памяти, начало которой располагается по адресу `str`, в поисках нулевого элемента типа `char`, который, заметим, обязан там присутствовать по условию задачи, иначе это была бы не строка. Наш первый вариант решения будет выглядеть совершенно по-паскалевски: мы просто вспомним, что, коль скоро указатель `str` указывает на массив, то с этим указателем можно и работать как с массивом, то есть применять к нему операцию индексирования; напомним ещё раз, что индексирование в Си — это операция над адресом, а не над массивом. Выглядеть это всё может примерно так:

```
int string_length(const char *str)
{
    int i;
    i = 0;
    while(str[i] != '\0')
        i++;
    return i;
}
```

Текущая рассматриваемая позиция здесь хранится в переменной `i`, мы начинаем рассмотрение с нулевой позиции, если в текущей позиции в массиве оказался ноль — возвращаем номер этой позиции в качестве результата: нетрудно видеть, что, поскольку нумерация позиций начинается с нуля, длина строки в точности равна первой позиции, которая в строку не входит (например, если ограничитель встретился в нулевой позиции, то строка пустая и её длина, соответственно, равна нулю).

Впрочем, опытные программисты на Си напишут то же самое совсем иначе. Во-первых, выражение `str[i]` никто не мешает использовать в качестве логического, а не сравнивать его зачем-то с нулём. Во-вторых, если внимательно посмотреть на цикл `while`, можно заметить, что непосредственно перед ним записано присваивание переменной цикла начального значения, а последним (и единственным) оператором в теле цикла является изменение значения переменной цикла на то, которое будет использовано в следующей итерации. В подобных ситуациях в Си используется цикл `for`. Исправив оба указанных недочёта, мы получим следующее:

```
int string_length(const char *str)
{
    int i;
    for(i = 0; str[i]; i++)
        {}
    return i;
}
```

Тело цикла здесь получилось пустым, что для Си вполне типично. Однако наверняка найдутся и те, кто заявит, что здесь слишком много операций сложения: на каждой итерации, во-первых, увеличивается на единицу переменная `i`, а во-вторых, её значение прибавляется к указателю `str`: в самом деле, `str[i]` есть, как мы уже знаем, строго то же самое, что и `*(str+i)`. Сэкономить одно сложение можно, если в качестве переменной цикла использовать не индекс текущего элемента строки, а указатель на него; вычислить результат в конце мы сможем благодаря операции вычитания адресов. Наша функция теперь примет следующий вид:

```
int string_length(const char *str)
{
    const char *p;
    p = str;
    while(*p)
        p++;
    return p - str;
}
```

или лучше вот так:

```
int string_length(const char *str)
{
    const char *p;
    for(p = str; *p; p++)
        {}
    return p - str;
}
```

Среди «прожжёных сишников» есть и те, кто предпочтёт следующий вариант:

```
int string_length(const char *str)
{
    const char *p = str;
    while(*p++);
    return p - str;
}
```

Никогда так не делайте! А чтобы было понятно, почему так делать не следует, попробуйте этот вариант функции тщательно разобрать на тему возможных ошибок, и засеките, сколько времени это у вас займёт.

В качестве следующего примера рассмотрим функцию копирования строки в заранее отведённую область памяти; будем предполагать, что памяти отведено достаточно, то есть за это несёт ответственность

вызывающий[42]. Мы назовём её `string_copy`. Возвращать этой функции вроде бы нечего, поэтому в качестве типа возвращаемого значения укажем `void`. В качестве параметров функция будет получать адрес области памяти, *куда* следует скопировать строку, а также, собственно, адрес строки; параметры мы по традиции назовём `dest` и `src` от слов *destination* и *source*. Порядок параметров выберем по аналогии с операцией присваивания: сначала *куда*, потом *откуда*. Наконец, обратим внимание на то, что оригинал строки функция не меняет, а вот область памяти, куда производится копирование — очевидно, меняет; с учётом этого заголовок получается таким:

```
void string_copy(char *dest, const char *src)
{
```

Начнём, как и в предыдущем примере, с варианта «по-паскалевски»; объявим переменную для текущей позиции и применим к обоим массивам операцию индексирования, но вариант с `while` пропустим и перейдём сразу к использованию `for`:

```
void string_copy(char *dest, const char *src)
{
    int i;
    for(i = 0; src[i]; i++)
        dest[i] = src[i];
    dest[i] = 0;
}
```

Обратите внимание на последнюю срочку; она нужна, чтобы превратить массив `dest` в корректную строку. Дело в том, что для значения `i`, соответствующего позиции ограничивающего нуля в исходной строке, тело цикла выполнено уже не будет, так что ноль не будет скопирован.

Следующая версия той же функции будет использовать указатели, чтобы сэкономить на сложениях; при этом мы воспользуемся тем, что параметры функции внутри неё представляют собой простые локальные переменные[43], которые вполне можно менять. В результате мы сможем обойтись без введения дополнительных локальных переменных:

```
void string_copy(char *dest, const char *src)
{
    while(*src) {
        *dest = *src;
        dest++;
        src++;
```

[42] Стандартная функция, выполняющая это действие, называется `strcpy`; см. также сноску 41 на стр. 260.

[43] Вспомните структуру стекового фрейма, которую мы подробно рассматривали при изучении программирования на языке ассемблера.

```
    }
    *dest = 0;
}
```

Или даже так:

```
void string_copy(char *dest, const char *src)
{
    for(; *src; dest++, src++)
        *dest = *src;
    *dest = 0;
}
```

Рассмотрение задачи копирования строки не будет полным без примера, представляющего подлинный образчик «истинно сишного» программирования. Оставляем сей пример для самостоятельного анализа в надежде, что время, потраченное на распутывание этого ребуса, позволит читателю в будущем не увлекаться подобными трюками:

```
void string_copy(char *dest, const char *src)
{
    while((*dest++ = *src++));
}
```

Отметим, что вторые круглые скобки мы здесь поставили, чтобы компилятор не выдавал нам предупреждение об использовании присваивания в качестве логического значения, что в большинстве случаев (но не в этом) свидетельствует об ошибочном использовании знака присваивания вместо знака сравнения (двойного равенства).

Следует заметить, что все вышеприведённые версии `string_copy` написаны в предположении, что области памяти `dest` и `src` не пересекаются. Для пересекающихся областей памяти функция побайтового копирования будет чуть сложнее: в зависимости от их взаимного расположения копирование может потребоваться в прямом или обратном порядке.

4.4.7. Строковые литералы

Разобравшись, что представляют собой строки в языке Си, мы сможем теперь понять, что конкретно делает компилятор при виде строки, заключённой в двойные кавычки. Прежде всего компилятор отводит где-то в *неизменяемой области памяти* (чаще всего — в сегменте кода, то есть в секции `.text`) нужное количество памяти, по одному байту на символ и один байт на завершающий ноль, и соответствующим образом эту область памяти заполняет. Например, видя литерал `"Hello"`, компилятор отведёт *шесть* ячеек памяти, в первую занесёт ASCII-код буквы `H`, во вторую — код буквы `e`, а в последнюю — ноль.

Неизменяемая область памяти выбирается из соображений эффективности. Как правило, изменение строкового литерала во время работы программы не нужно, а при одновременном запуске нескольких экземпляров одной и той же программы современные операционные системы создают в памяти только один экземпляр неизменяемой части программы (в том числе сегмента кода). В системах семейства Unix это особенно актуально, ведь новые процессы в этих системах создаются путём копирования существующего процесса. Как следствие, размещение строковых литералов в неизменяемой области памяти позволяет при работе нескольких копий одной программы иметь в памяти один экземпляр всех её строковых констант.

Разместив содержимое строкового литерала в генерируемом объектном модуле, компилятор вместо него подставляет, как можно догадаться, *адрес первого элемента*, причём этот адрес снабжается модификатором `const`, поскольку попытки изменения внутренностей строкового литерала ни к чему хорошему не приведут. Иначе говоря, в контексте выражения, где встречен строковый литерал, он имеет тип `const char *` и представляет адрес той области (неизменяемой) памяти, где расположено его содержимое.

Между прочим, возможно даже такое выражение: `"Abrakadabra"[4]`; оно будет иметь тип `char`, а его значением будет код буквы `k`. В самом деле, литерал `"Abrakadabra"` есть адрес, а к адресу можно применить операцию индексирования. Некоторые программисты пытаются применять выражения подобного рода (конечно, с переменной величиной в качестве индекса, иначе это было бы вовсе лишено смысла), но лучше так не делать, поскольку ясности вашей программе подобные трюки не добавят. Также возможно и выражение `*"Abrakadabra"`, которое равно коду буквы `A`; его применение может преследовать только одну цель: специально запутать программу.

Из всего сказанного про строковые литералы есть одно важное исключение. **Строковый литерал может применяться в качестве инициализатора массива** элементов типа `char`, `signed char` и `unsigned char`, и при этом, естественно, компилятор не станет размещать его в неизменяемой памяти (кстати, даже в том случае, если массив будет объявлен со словом `const`). Например, описание

```
char str[6] = "Hello";
```

создаст массив из шести элементов — точно так же, как если бы мы написали

```
char str[6] = { 'H', 'e', 'l', 'l', 'o', 0 };
```

Число, обозначающее размерность массива, можно, как обычно, опустить:

```
char str[] = "Hello";
```

Здесь размерность массива составит шесть элементов, так как именно столько подразумевается строковым литералом (пять — длина строки, плюс один на ограничительный ноль). Подчеркнём, что это совершенно **не** то же самое, что написать

```
char *ptr = "Hello";
```

В этом случае описывается не массив, а указатель, который инициализируется адресом строкового литерала, тогда как сам литерал при этом воспринимается компилятором согласно общим правилам, и всё сказанное про неизменяемые области памяти снова обретает силу.

Самое очевидное проявление различия между вышеописанными `str` и `ptr` состоит в том, что `str` — не переменная в том смысле, что ей нельзя ничего присваивать, тогда как `ptr` — обыкновенная переменная типа «указатель». Кроме того, `sizeof(ptr)` будет равен размеру указателя (то есть 4 на 32-битных системах и 8 на 64-битных), тогда как `sizeof(str)` будет равен размеру области памяти, занятой массивом, в данном случае `6`. Наконец, массив `str` можно изменять, то есть, например, присваивание `str[4]=0` превратит слово `Hello` в слово `Hell`; если же попытаться сделать `ptr[4]=0`, это приведёт к попытке записи внутрь строкового литерала, так что ваша программа завершится аварийно. Настоятельно рекомендуется попробовать так сделать, компьютер от такой ошибки не взорвётся, а вы будете знать, как всё это выглядит на практике.

4.5. Обработка аргументов командной строки

При работе в операционной среде ОС Unix мы, как правило, запускаем программы, указывая кроме их имён ещё и определённые параметры — имена файлов, опции и т. п. Так, при запуске компилятора `gcc` мы можем написать что-то вроде

```
gcc -Wall -g prog.c -o prog
```

Слова, указанные после имени программы, называются ***параметрами командной строки***. В данном случае этих аргументов пять: ключи «`-Wall`» и «`-g`», имя файла «`prog.c`», ключ «`-o`» и имя исполняемого файла «`prog`». Отметим, что и само имя программы, в данном случае «`gcc`», считается элементом командной строки. Иначе говоря, командная строка представляет собой массив строк, состоящий в данном случае из шести элементов: «`gcc`», «`-Wall`», «`-g`», «`prog.c`», «`-o`» и «`prog`».

Естественно, мы и сами можем написать программу, получающую те или иные сведения через командную строку. При запуске программы

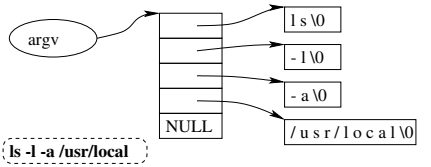

Рис. 4.2. Структура данных командной строки

операционная система отводит в её адресном пространстве специальную область памяти, в которой располагает строки, составляющие командную строку. Получить доступ к этой области памяти можно, описав функцию `main` как *имеющую параметры*, причём первый из этих параметров должен иметь тип `int` — он задаёт количество элементов командной строки. Второй параметр представляет собой адрес начала массива из указателей на сами строки (слова), составляющие командную строку; ясно, что каждый элемент этого массива имеет тип `char*`, ну а адрес первого (как и любого) элемента этого массива будет уже типа `char**`. Обычно эти аргументы функции `main` называются `argc` и `argv` от слов *argument count* и *argument vector*. Заголовок функции `main` с учётом этого принимает следующий вид:

```
int main(int argc, char **argv);
```

На рис. 4.2 показана структура данных, доступная через указатель `argv`, на примере команды `ls -l -a /usr/local`. Параметр `argc` в этом случае будет равен четырём по количеству значащих элементов командной строки, но в массиве, как можно заметить, элементов на один больше: последний из них (в данном случае пятый) равен «нулевому адресу» `NULL`.

Например, следующая программа напечатает свои аргументы командной строки, взяв каждый из них для наглядности в квадратные скобки:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int i;
    for(i = 1; i < argc; i++)
        printf("[%s]\n", argv[i]);
    return 0;
}
```

Комбинация `%s` в форматной строке `printf` (s от слова *string*) означает печать строки, расположенной в памяти по адресу, который берётся из очередного аргумента. Результат работы программы будет выглядеть, например, так:

```
avst@host:~$ ./cmdprint abra kadabra shvabra
[abra]
[kadabra]
[shvabra]
avst@host:~$ ./cmdprint 1       2    " 3   4 " 5
[1]
[2]
[ 3   4 ]
[5]
avst@host:~$
```

Написать программу `cmdprint` можно и иначе, не используя `argc`, а вместо этого воспользовавшись наличием `NULL` в конце массива указателей, например:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    argv++;
    while(*argv) {
        printf("[%s]\n", *argv);
        argv++;
    }
    return 0;
}
```

или, что то же самое:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    while(*++argv)
        printf("[%s]\n", *argv);
    return 0;
}
```

Следующая программа напечатает имя, по которому её вызвали:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("My name is %s\n", argv[0]);
    return 0;
}
```

Например:

```
avst@host:~$ ./printname
My name is ./printname
avst@host:~$
```

4.6. Стандартные функции ввода-вывода

Отвлечёмся на некоторое время от возможностей языка Си и попробуем освоить небольшую часть стандартной библиотеки, которую обычно называют ***средствами высокоуровневого ввода-вывода***. Все возможности, которые мы рассмотрим в этом параграфе, покрываются всё тем же заголовочным файлом `stdio.h`.

4.6.1. Посимвольный ввод-вывод

Напомним, что постоянно попадающиеся нам «ввод с клавиатуры» и «вывод на экран» правильнее было бы называть вводом *из стандартного потока ввода* и выводом *в стандартный поток вывода*, ведь никто не может нам гарантировать, что по ту сторону стандартных потоков находятся действительно клавиатура и экран; в общем случае мы даже не можем с уверенностью это определить[44]. Именно так мы и будем говорить; «вывод на экран и ввод с клавиатуры» оставим тем, кто не собирается становиться программистом. Больше того, поскольку этот и следующий параграфы посвящены только стандартным потокам, мы не будем этот момент уточнять, пока не придёт время работы с произвольными потоками, а не только стандартными; пока же под словами «ввод» и «вывод», а равно «чтение» и «запись» мы будем понимать соответствующие операции над стандартными потоками ввода-вывода.

Начнём с *ввода символа* или, точнее говоря, ввода очередной порции данных, соответствующей минимальному адресуемому на данной машине; в реальной жизни это всегда один байт. Функция, которая выполняет эту операцию, называется `getchar`; её профиль выглядит так:

```
int getchar();
```

Как видим, она не получает параметров, а возвращает значение типа `int`, то есть целое число. Но почему не `char`, или `signed char`, или `unsigned char`? По смыслу ведь они, как кажется, подходят лучше? Действительно, если читается один байт[45], разрядности любого из `char`'ов хватило бы для представления прочитанного значения, но только при

[44] В принципе, в ОС Unix мы можем задать системе вопрос, связан ли данный файловый дескриптор с ***терминальным устройством*** или нет; однако терминальное устройство может быть (а в современных условиях практически всегда является) результатом программной эмуляции, так что экрана и клавиатуры по ту сторону дескриптора может не оказаться, даже если на вопрос о терминальном устройстве система ответит утвердительно; хотя, конечно, в большинстве случаев «терминальное устройство» предполагает, что «с той стороны» присутствует живой пользователь.

[45] Не будем больше повторять поднадоевшее заклинание про минимальное адресуемое.

условии, что **чтение прошло успешно** и очередное значение действительно прочитано. И проблема здесь не только и не столько в том, что может произойти ошибка: на стандартном вводе ошибки происходят редко. Мы знаем одну *абсолютно штатную* ситуацию, когда никаких ошибок не происходит, но очередного байта в потоке ввода нет и прочитать оттуда нечего: это ***ситуация конца файла***. Если пользователь, запустивший нашу программу, перенаправил нам стандартный ввод, «подсунув» вместо клавиатуры обыкновенный файл, ситуация конца файла возникнет (спасибо Капитану Очевидность), когда файл будет весь прочитан, то есть он кончится; но даже если на ввод нашей программе поступают данные, которые живой пользователь набивает на обычной клавиатуре, ситуация конца файла всё равно может случиться: драйвер терминала *имитирует* её при нажатии комбинации клавиш `Ctrl-D`.

Если функция `getchar` успешно прочитала один байт, она именно его и вернёт, причём прочитанный байт будет проинтерпретирован как беззнаковое число соответствующей разрядности; иначе говоря, возвращено будет значение от `0` до `255`. Если бы так происходило всегда, функция действительно могла бы возвращать значение типа `unsigned char`, но есть ведь ещё ситуация конца файла. Обозначить эту ситуацию одним из возможных значений типа `char` нельзя, поскольку в потоке ввода может встретиться абсолютно любой байт — никто ведь не гарантирует нам, что эти данные текстовые, бинарный файл тоже может быть прочитан из стандартного потока ввода[46]; каково бы ни было значение из диапазона `0..255`, избранное нами для обозначения ситуации конца файла, это привело бы к неразличимости ситуации «файл кончился» и ситуации «из файла прочитан байт, равный значению, обозначающему ситуацию конца файла». Ясно, что это абсолютно разные ситуации, так что специальное значение, возвращаемое функцией, если очередной байт прочитан не был, не должно находиться в диапазоне значений прочитанных байтов, то есть должно лежать *за пределами диапазона* `0..255`.

В качестве такого значения авторы стандартной библиотеки Си избрали обыкновенную минус-единицу (-1), но для большей ясности её обозначают идентификатором `EOF` от слов *end of file*, причём именно так, заглавными буквами. Функция, таким образом, возвращает одно из 257 возможных значений, а для этого разрядности `char`'ов уже не хватает; поэтому её тип возвращаемого значения должен иметь бóльшую разрядность, а в таких случаях в языке Си обычно используют `int`.

Для примера рассмотрим программу, которая читает текст и на каждую прочитанную строку отвечает лаконичным `OK`. Это можно сделать, например, так:

[46] Более того, целый ряд утилит командной строки именно так и делает; в среде ОС Unix так поступают, например, практически все архиваторы.

```
#include <stdio.h>

int main()
{
    int c;
    c = getchar();
    while(c != EOF) {
        if(c == '\n')
            printf("OK\n");
        c = getchar();
    }
    return 0;
}
```

Надо отметить, что профессиональные программисты, пишущие на Си, так никогда не делают, поскольку считают, что нехорошо дублировать строчку «`c = getchar();`» перед циклом и в конце тела цикла. В отличие от Паскаля, где такого дублирования можно избежать разве что с помощью оператора `break`, в Си можно воспользоваться тем, что присваивание является операцией, и «загнать» вызов функции `getchar` в заголовок цикла, а результат присваивания сравнить с константой `EOF`. Здесь необходимо вспомнить, что приоритет присваивания ниже, чем приоритет сравнения, так что операцию присваивания придётся взять в скобки. Всё вместе будет выглядеть так:

```
#include <stdio.h>

int main()
{
    int c;
    while((c = getchar()) != EOF) {
        if(c == '\n')
            printf("OK\n");
    }
    return 0;
}
```

Теперь давайте слегка усложним нашу программу: пусть она вместо `OK` печатает длину только что прочитанной строки. Для этого мы введём счётчик, то есть целочисленную переменную, в которой будет накапливаться количество прочитанных символов, а когда строка кончится, программа напечатает значение этой переменной и обнулит её, чтобы подсчёт символов следующей строки начался с нуля. Всё вместе будет выглядеть так:

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

```
int main()
{
    int c, n;
    n = 0;
    while((c = getchar()) != EOF) {
        if(c == '\n') {
            printf("%d\n", n);
            n = 0;
        } else {
            n++;
        }
    }
    return 0;
}
```

Подчеркнём, что в обоих примерах переменная, в которую заносится значение, возвращённое функцией `getchar`, имеет тип `int`. Начинающие программисты, не учитывая вышеприведённых рассуждений о типе значения `getchar`, часто делают характерную ошибку, описывая эту переменную как `char`, в результате чего их программа не различает ситуации, когда `getchar` вернула `EOF` (т. е. -1) и когда она вернула число `255`, прочитав байт с таким значением. При обработке текстовой информации видимый результат такой ошибки зависит от применяемой кодировки; к примеру, если наша программа работает в ОС Windows и обрабатывает текст в кодировке cp1251, она «свалится» по концу файла, прочитав маленькую букву «я», поскольку её код в cp1251 равен как раз `255`; при использовании кодировки `koi8-r`, которая до сих пор достаточно часто встречается в системах семейства Unix, тот же эффект будет наблюдаться при прочтении заглавного твёрдого знака. Если текст представлен в кодировке `utf8`, то байт со значением `255` там встретиться не должен, но если он в потоке данных всё-таки случайно окажется, то программа, опять-таки, примет его за конец файла. Если же на стандартный поток ввода поданы бинарные данные, то байт со значением `255` в них может встречаться довольно часто: достаточно записать в файл в двоичном виде небольшое по модулю отрицательное целое число, и его представление будет состоять из одного значащего байта и всех остальных, равных как раз `255`.

На всякий случай повторим ещё раз: **значение, возвращённое функцией `getchar`, нельзя сразу присваивать переменной типа `char`**, это приведёт к потере информации; такое присваивание можно сделать лишь после того, как мы убедились, что возвращено значение, отличное от `EOF`.

Операцию вывода символа в стандартный поток вывода производит функция `putchar`; её заголовок выглядит так:

```
int putchar(int c);
```

Параметр задаёт код выводимого символа; формально он имеет тип `int`, но на самом деле числа за пределами диапазона значений `unsigned char` (в реальной жизни это всегда диапазон `0..255`) будут превращены в числа этого диапазона путём отбрасывания «лишних» битов, так что давать их этой функции ни к чему.

Функция возвращает код выведенного символа, если всё в порядке, а если произошла ошибка, то уже знакомую нам константу `EOF`. Впрочем, обычно в программах это значение не проверяется, точно так же, как не проверяется значение `printf`: ошибки при выводе в поток стандартного вывода встречаются довольно редко.

Например, следующая программа принимает на ввод произвольный текст, а выводит начало каждой строки до первого пробела. Для этого предусматривается флажок `pr`, который равен «истине», если в текущей строке ещё не было пробелов, и «лжи», если пробелы уже встречались.

```
/* untilspace.c */
#include <stdio.h>

int main()
{
    int c, pr;
    pr = 1;   /* true */
    while((c = getchar()) != EOF) {
        switch(c) {
        case '\n':
            putchar('\n');
            pr = 1;
            break;
        case ' ':
            pr = 0;
            break;
        default:
            if(pr)
                putchar(c);
        }
    }
    return 0;
}
```

4.6.2. Форматированный ввод-вывод

Функции ***форматированного ввода-вывода*** `printf` и `scanf` мы уже активно использовали в примерах, каждый раз поясняя очередную их возможность. Настало время навести в этом деле порядок, и начнём мы с функции `printf`. Формальный её прототип выглядит так:

```
int printf(const char *format, ...);
```

Многоточие означает, что в этом месте может быть *ещё сколько угодно параметров*; такие функции называются обычно ***вариадическими***. Здесь присутствуют некоторые ограничения на типы таких параметров, которые мы будем рассматривать, когда дело дойдёт до *написания* вариадических функций, а пока просто примем как данность, что параметры всех типов, которые умеет печатать `printf`, через загадочное многоточие передать можно.

Работа функции `printf` *управляется* форматной строкой; можно сказать, что функция представляет собой *интерпретатор* форматной строки. Формально говоря, форматная строка состоит из ***форматных директив***, которые делятся на *обычные символы* (любые байты, кроме нулевого и символа «%») и *директивы преобразования*, которые начинаются с символа «%» и заканчиваются *спецификатором преобразования*, в роли которого могут выступать символы `diouxXeEfFgGcsp%`; некоторые реализации поддерживают и другие символы в дополнение к этому набору. Спецификатор преобразования указывает, какого типа значение следует извлечь из списка фактических параметров функции (то есть, попросту говоря, из очередной позиции стека) и в каком виде представить это значение на печати. Например, мы уже встречали спецификатор «`d`» (от слова *decimal*) и знаем, что «`%d`» означает целый тип и его десятичное представление; спецификаторы «`o`» и «`x`» тоже означают целый тип, но требуют представить его в восьмеричной и шестнадцатеричной системах соответственно. Описание всех перечисленных спецификаторов приведено в табл. 4.1.

Внутри директивы преобразования, то есть между символом `%` и символом спецификации формата, могут быть указаны дополнительные параметры форматирования (каждый из них не обязателен, то есть может присутствовать, а может и отсутствовать:

- флаги: набор (возможно, пустой) из символов «`-`», «`+`», «`0`», «`#`» и пробел;
- целое число, задающее ширину поля, то есть количество знакомест, отведённых на вывод данного параметра;
- десятичная точка и целое число, задающее так называемую ***точность представления***;
- так называемый ***модификатор разрядности***.

Начнём с ширины и флагов. Число, задающее ширину, означает минимальное количество знакомест, которое будет использовано для выдачи представления очередного параметра; если представление занимает меньше знаков, оно будет (в простейшем случае) дополнено пробелами слева. Например, `printf("[%5d]", 12)` напечатает «`[   12]`». Флаги позволяют изменить это поведение:

- знак «`-`» означает, что выравнивание печатаемой информации следует проводить по левому краю, а не по правому; например, `printf("[%-5d]", 12)` напечатает «`[12   ]`»;

Таблица 4.1. Спецификаторы преобразований функции `printf`

символ	значение
`d`, `i`	знаковое целое число представляется в десятичной системе счисления
`o`	беззнаковое целое число представляется в восьмеричной системе счисления
`u`	беззнаковое целое число представляется в десятичной системе счисления
`x`, `X`	беззнаковое целое число представляется в шестнадцатеричной системе счисления, причём `x` использует буквы `abcdef`, а `X` использует буквы `ABCDEF`
`f`, `F`	число с плавающей точкой представляется в виде десятичной дроби
`e`, `E`	число с плавающей точкой представляется в экспоненциальной форме, например, `6.6234e-34` (буква `e` или `E`, отделяющая порядок от мантиссы, выбирается та же, что и в спецификаторе)
`g`, `G`	число с плавающей точкой представляется в экспоненциальной форме, если значение порядка меньше `-4` или больше либо равен точности, заданной перед спецификатором (по умолчанию 6); в противном случае число представляется в виде обычной десятичной дроби, причём если число имеет нулевую дробную часть, то десятичная точка не печатается
`c`	параметр является целым числом, это число воспринимается как однобайтный код символа, печатается в итоге символ
`s`	параметр воспринимается как адрес строки; строка печатается
`p`	параметр воспринимается как адрес типа `void*` и печатается в шестнадцатеричном виде
`%`	печатается символ `%`, никакие параметры из стека не извлекаются

- цифра «`0`» означает, что число следует дополнять слева не пробелами, а нулями; например, `printf("[%05d]", 12)` напечатает «`[00012]`»;
- знак «`+`» означает, что число должно обязательно печататься со знаком, даже если оно положительное; например, `printf("%+d,%+d", 12, -3)` напечатает «`+12,-3`»; без этого знака отрицательные числа всё равно печатаются с минусом, но положительные знаком не снабжаются;
- символ «пробел» означает, что для знака нужно оставить место, но печатать его только для отрицательных чисел, а положительные предварять пробелом; например, `printf("[% d],[% d]", 12, -3)` напечатает «`[ 12],[-3]`».

Флаг «`#`», хотя и входит в число флагов, задаёт не дополнительное требование к ширине поля, а требование модифицировать сам фор-

мат. Для восьмеричных и шестнадцатеричных чисел этот флаг требует вывести соответственно лидирующий `0` или комбинацию `0x`/`0X` (в зависимости от того, используется ли `%x` или `%X`); для спецификаторов `e`, `E`, `f` и `F` флаг требует обязательной выдачи десятичной точки, даже если дробная часть равна нулю; для `g` и `G` флаг запрещает удалять незначащие нули в дробной части.

Отметим, что ширина задаёт *минимальное*, но отнюдь не *максимальное* количество используемых знакомест. Например, `printf("%3d", 1234)` напечатает «1234», никакие цифры отброшены не будут, несмотря на то, что число `1234` в указанные три позиции не помещается.

После ширины (или вместо неё, если её нет) можно задать *точность*, записанную с точки. Проще всего с точностью обстоят дела для чисел с плавающей точкой: точность задаёт количество цифр после запятой (для `g` и `G` — количество значащих цифр в мантиссе). Однако работает эта часть директивы преобразования не только для чисел с плавающей точкой, но и для целых и даже для строк, и в этих случаях её семантика не столь очевидна. Для строк «точность» означает *максимальное* количество символов строки, остальные будут просто отброшены. Например, `printf("[%7.5s]", "abrakadabra")` напечатает «`[  abrak]`»: от строки будет взято только пять символов в соответствии с заданной «точностью», при этом на всю директиву будет отведено семь позиций, то есть произойдёт дополнение слева двумя пробелами; `printf("[%-7.5s]", "abrakadabra")` напечатает «`[abrak  ]`», и т. д.

Для целого числа «точность» означает *минимальное количество цифр*; обычно это применяется совместно с «шириной», то есть число сначала дополняется незначащими нулями до требуемой «точности», а затем — пробелами до нужной «ширины». Например, `printf("[%6.4d]", 12)` напечатает «`[  0012]`».

Наконец, последняя часть директивы преобразования, записываемая непосредственно перед символом-спецификатором, представляет собой модификатор разрядности, который может быть:

- буквой «`h`», означающей, что целочисленный параметр имеет тип `short` или `unsigned short`;
- буквой «`l`» («эл»), означающей, что целочисленный параметр имеет тип `long` или `unsigned long`;
- двумя буквами «`ll`» («эл-эл»), означающими, что целочисленный параметр имеет тип `long long` или `unsigned long long`;
- заглавной буквой «`L`», означающей, что параметр с плавающей точкой имеет тип `long double`.

Различать `float` и `double` не требуется, поскольку при вызове любой функции значение выражения типа `float`, указанное в списке фактических параметров, всегда преобразуется к типу `double`.

Например, «`% #7.4Lg`» — это пример директивы преобразования, в которой присутствуют все возможные части: два флага (пробел и «`#`»), ширина `7`, точность `4`, модификатор разрядности `L` и, наконец, символ-спецификатор преобразования `g`.

Отметим ещё одну интересную возможность. Как ширину, так и точность можно не писать в явном виде, а взять из очередного целочисленного параметра. Для этого в соответствующем месте директивы вместо числа пишется звёздочка «`*`». Например. `printf("%*.*d", w, p, n)` напечатает число `n` с использованием ширины, взятой из `w`, и точности, взятой из `p`.

Для форматированного *ввода* из стандартного потока ввода используется функция `scanf`; её заголовок несколько напоминает заголовок функции `printf`:

```
int scanf(const char *format, ...);
```

Эта функция, как и функция `printf`, *интерпретирует форматную строку*, однако неверно было бы предполагать, что, запомнив правила записи форматной строки `printf`, мы сможем грамотно воспользоваться также и `scanf`'ом; правила интерпретации их форматных строк весьма существенно различаются.

Отметим прежде всего, что видов форматных директив тут не два, а три. Во-первых, есть *директивы преобразования*, формируемые, как и для `printf`, из символа процента и символа-спецификатора, между которыми могут быть необязательные дополнительные настройки в виде числа, обозначающего максимальную ширину поля (здесь надо отметить, что у `scanf` отсутствует аналог задания «точности», а сама «ширина» имеет совершенно иную семантику), и, возможно, буквы, задающей/изменяющей разрядность читаемого числа. Каждая такая директива означает, что нужно прочитать из потока значение в определённом формате и занести его в область памяти, *адрес которой передан очередным параметром*. Кроме перечисленных настроек, в директиве сразу после символа `%` можно указать единственный *флаг*, понимаемый `scanf` — символ «`*`», который *запрещает присваивание*, то есть значение читается, но никакой параметр из стека не берётся и никакая информация ни в какую память не заносится: например, «`%*i`» предписывает прочитать целое число, но никуда его не заносить.

Во-вторых, для `scanf` в качестве директив особого рода рассматриваются последовательности *пробельных символов* произвольной длины, причём пробельными считаются собственно пробел, а также табуляция, перевод строки и т. д.; любая такая последовательность, встреченная в форматной строке (чаще всего это просто один пробел, во всяком случае, нет никаких оснований использовать что-то другое) рассматривается как указание выбирать из потока ввода пробельные символы и игнорировать их, и так до тех пор, пока очередной символ не окажется непробельным.

Наконец, как и `printf`, `scanf` рассматривает *все остальные символы* как директивы особого рода, но если `printf` их просто печатал, то `scanf` требует, чтобы, если он дошел в интерпретации форматной строки до обычного символа, то в потоке ввода в этот момент встретился *точно такой же символ*; если символы в потоке и в форматной строке не совпадают, `scanf` прекращает работу, оставив «неправильный» символ в потоке. То же самое происходит, если `scanf` не смогла трактовать последовательность символов как текстовую запись значения, которое от неё ожидается (целое число, число с плавающей точкой и т. п.)

Ещё одно ключевое отличие состоит в следующем. Для `printf` можно не различать, например, `float` и `double` (поскольку все значения типа `float` при передаче в качестве параметров приводятся к `double`), а также можно, в принципе, не отличать числа типа `char` и `short` от обычных `int`'ов. При использовании `scanf` такой номер не проходит, ведь в функцию передаются не значения, а *адреса* областей памяти, и не принимать во внимание их разрядность — заведомая ошибка.

Набор используемых символов-спецификаторов и их роли для `scanf` несколько напоминают спецификаторы для `printf`, но было бы ошибкой считать эти наборы одинаковыми. Например, для `printf` «`%d`» и «`%i`» означают одно и то же, тогда как поведение `scanf` для этих спецификаторов различается: если «`%d`» предписывает прочитать из потока целое число, записанное в десятичной системе, то «`%i`» предписывает прочитать целое число, которое может быть записано в десятичной, восьмеричной или шестнадцатеричной системах в соответствии с правилами языка Си: лидирующий ноль означает восьмеричную систему, а префикс «`0x`» или «`0X`» — шестнадцатеричную. С другой стороны, при работе с `printf` мы не различаем числа типа `float` и `double`, тогда как для `scanf` мы вынуждены обозначать их по-разному: например, «`%f`» одначает прочтение числа типа `float` (иначе говоря, прочтение числа с плавающей точкой и занесение результата в память по адресу, заданному в параметре, в предположении, что по этому адресу расположена переменная типа `float`), тогда как для `double` мы используем комбинацию «`%lf`»; функция `printf` такого спецификатора вообще не знает.

Спецификаторы, используемые `scanf`, перечислены в табл. 4.2. При чтении целых чисел `scanf` предполагает, что очередной параметр имеет тип `int*` (для «`%i`» и «`%d`») или `unsigned int*` (для «`%o`», «`%x`» и «`%u`»); это можно изменить, добавив непосредственно перед спецификатором модификатор разрядности: `h` для `short` (или `unsigned short`), `l` для `long` (соответственно, `unsigned long`), `L` (а не `ll`, как можно было бы ожидать) для `long long` или `unsigned long long`. При чтении чисел с плавающей точкой предполагается тип `float`, для `double` нужно указать модификатор `l`, для `long double` — модификатор `L`. Ширина поля для чисел обычно не используется, но в принципе её тоже можно

Таблица 4.2. Спецификаторы преобразований функции `scanf`

символ	значение
`d`	целое число в десятичной системе счисления
`i`	целое число в восьмеричной, десятичной или шестнадцатеричной системе счисления (с префиксами в соответствии с правилами Си)
`o`	целое число в восьмеричной системе счисления
`u`	беззнаковое целое число в десятичной системе счисления
`x`	целое число в шестнадцатеричной системе счисления (допускается префикс `0x` или `0X`, но он не обязателен)
`e`, `f`, `g`	число с плавающей точкой (по умолчанию `float`)
`p`	адресное выражение в том виде, в котором его печатает `printf`
`c`	символ (записывается код символа); по умолчанию читается один символ, но если задана ширина поля, то будет прочитано соответствующее количество символов в элементы массива, на начало которого указывает очередной параметр (массив не будет строкой, поскольку «нулевой» символ в его конец не добавляется)
`s`	строка; читается от текущей позиции максимально длинная цепочка, не содержащая пробельных символов (и не превышающая заданную ширину поля)
`[...]`	строка, состоящая из заданных символов; читается от текущей позиции максимально длинная цепочка, содержащая только символы из перечисленных между скобками (и не превышающая заданную ширину поля); если нужно включить в набор символ `]`, его указывают первым, например `[]abc]`
`[^...]`	строка, состоящая из любых символов, кроме заданных; читается от текущей позиции максимально длинная цепочка, содержащая любые символы, кроме перечисленных между скобками (и не превышающая заданную ширину поля); если нужно включить в набор символ `]`, его указывают первым, например `[^]abc]`

указать; тогда для формирования числа будет использовано не более чем заданное количество символов из потока ввода.

Совершенно иначе обстоят дела при чтении строк, то есть при использовании «`%s`», «`%[...]`» и «`%[^...]`». **Если не указать ширину поля при чтении строки с помощью `scanf`, то каков бы ни был размер массива, который вы создали для размещения этой строки, этого размера может не хватить и произойдёт переполнение**. При этом если ваша программа просто «упадёт» (то есть аварийно завершится), вы можете рассматривать это как невероятное везение; чем на самом деле опасны переполнения массивов данными, читаемыми извне, подробно рассказывается на стр. 291. Подчеркнём, что **за такое увольняют с работы**, так что игнорировать это замечание

крайне не рекомендуется. Короче говоря, **указание ширины поля при чтении строк абсолютно обязательно**, причём следует учитывать, что массив, в который будет прочитана строка, должен быть хотя бы на один элемент больше, чтобы осталось место для оканчивающего строку «нулевого» символа.

Отметим, что возможности, как для `printf`, взять параметр ширины поля из списка аргументов у `scanf` нет, что резко сужает нашу свободу манёвра: либо мы фиксируем ширину поля и размер массива во время написания программы, либо, как вариант, можем *сгенерировать форматную строку* во время исполнения, но это уже оказывается сложнее, чем устроить обыкновенное посимвольное чтение без всякого `scanf`. Большинство программистов сходятся во мнении, что осмысленное применение `scanf` возможно только в маленьких вспомогательных программах, которые пишутся ради одного-двух запусков и не передаются конечным пользователям, то есть единственным пользователем которых является их автор. Во всех остальных случаях возможностей `scanf` недостаточно для организации полноценной диагностики ошибок пользовательского ввода и т. п.

Так или иначе, полезно знать, что `scanf` возвращает количество успешно преобразованных и помещённых в память значений, то есть, иначе говоря, количество успешно обслуженных директив преобразования, предполагающих запись по адресам, заданным в списке параметров, либо `-1`, если достигнут конец файла или произошла ошибка при чтении. Например, `scanf("%d %d %d", &a, &b, &c)` может вернуть `-1`, если, не успев прочитать ни одного числа, она «упёрлась» в конец файла; она вернёт `0`, если при попытке проанализировать первое же число она встретила ошибочный ввод (буквы, знаки препинания или любую другую белиберду, которая не может быть истолкована как число); значение `1` будет возвращено, если первое число было успешно прочитано, а белиберда встретилась при чтении второго, и так далее; в случае полного успеха (то есть когда все три предполагаемых числа успешно прочитаны) будет возвращено значение `3`. Учтите, что программа, использующая `scanf`, заведомо мало на что годится, но если она ещё и не анализирует значение, которое `scanf` возвращает, то такую программу можно разве что выбросить.

4.6.3. Работа с текстовыми файлами

Ясно, что кроме стандартных потоков ввода-вывода нам могут понадобиться и другие потоки, к которым относятся, кроме всего прочего, обыкновенные файлы; точнее говоря, для работы с файлом его, как известно, требуется *открыть*, в результате чего как раз и образуется новый поток ввода-вывода.

Для идентификации потоков ввода-вывода библиотека предусматривает своеобразный аналог паскалевских файловых переменных — тип `FILE*`. Судя по наличию звёздочки, это некий *адрес*, и можно даже сказать более определённо — это адрес чего-то, что имеет тип `FILE`. Больше на эту тему сказать ничего нельзя, разные версии библиотеки могут описывать это «нечто» совершенно по-разному, и как оно на самом деле устроено, нас не волнует; то есть мы можем, разумеется, заглянуть как в заголовочный файл (в данном случае `stdio.h`), так и в исходные тексты стандартной библиотеки и в результате узнать, что за сущность в действительности скрывается за словом `FILE`, но делать это стоит разве что из любопытства. В самом деле, если мы попытаемся как-то воспользоваться полученным знанием, то наша программа с хорошей вероятностью не откомпилируется при смене версии стандартной библиотеки из-за того, что там внутренности типа `FILE` могут оказаться совершенно иными.

Можно заметить, что подход к *файловым переменным* в Си радикально отличается от принятого в Паскале. Там переменная соответствующего типа содержит всё необходимое для работы с файлом, в результате чего, например, переменные файлового типа в Паскале нельзя передавать в подпрограммы по значению. Библиотека языка Си, напротив, *сама* заботится о хранении всей нужной для работы с файлом информации, а пользователю (программисту) показывает лишь некий *адрес*, служащий для целей идентификации потока ввода-вывода (то есть отличия конкретного потока от всех остальных), но не более того. Указатели типа `FILE*` можно присваивать, можно передавать их в функции по значению, можно возвращать из функций, ведь передаётся/копируется в каждом случае не информация, связанная с файлом, а простое адресное значение.

Второе весьма заметное отличие от Паскаля состоит в том, что для работы с «нестандартными» потоками ввода-вывода предусмотрен отдельный набор функций, тогда как в Паскале мы использовали всё те же операторы (псевдо-процедуры) `read` и `write`, что и для стандартных потоков, только первым параметром указывали файловую переменную.

Операция открытия файла для высокоуровневого ввода-вывода выполняется функцией `fopen`, которая имеет следующий заголовок:

```
FILE* fopen(const char *name, const char *mode);
```

Как можно заметить, оба параметра — это *строки*, причём неизменяемые, то есть функция `fopen` ничего не пытается делать с их содержимым; это позволяет при необходимости использовать в качестве одного или обоих параметров строковые литералы. В качестве первого параметра выступает ***имя файла***, про которое нужно только знать, что если оно начинается с символа «`/`», то это ***абсолютный путь***, начинающийся

от корневой директории и не зависящий от того, в какой директории мы сейчас находимся; в противном случае имя считается заданным относительно текущей директории. В частности, если в имени вообще нет ни одного слэша, имеется в виду файл в текущей директории.

Второй параметр задаёт режим открытия файла, то есть то, что мы собираемся с этим файлом делать. Вариантов здесь не так много: `"r"` означает, что файл следует открыть только для чтения; `"r+"` показывает, что файл нужно открыть на чтение и запись, при этом работа начнётся с начала файла; `"w"` предписывает открытие файла на запись, при этом если файла не было, он создаётся (пустым), а если он уже существовал, его старое содержимое уничтожается; `"w+"` делает практически то же самое, только файл при этом открывается на запись и на чтение, то есть можно, записав что-то в файл, после этого принудительно спозиционироваться на уже записанные данные и прочитать их; `"a"` открывает файл на добавление информации в конец, то есть файл открывается на запись, текущее содержимое сохраняется, а запись начинается с первой позиции после старого конца файла; если файла не существовало, он создаётся; `"a+"` работает довольно хитро: файл открывается на чтение и запись, причём чтение начинается с начала файла, а запись происходит в его конец.

Реализация функции `fopen` в системах семейства Unix допускает ещё одну букву — «`b`» от слова *binary*, которая по идее должна означать, что открываемый файл следует рассматривать как двоичный, то есть не текстовый. Эту букву можно добавить в конец строчки `mode` или поставить перед символом `+` (что-то вроде `"rb"`, `"a+b"`, `"wb+"` и т. п.). В Unix-системах эта буква игнорируется, то есть совершенно ничего не делает и ни на что не влияет; однако если вам в голову придёт перенести вашу программу под Windows, вы можете с удивлением обнаружить, что работает ваша программа «как-то не так». Автор книги в своё время потратил часа полтора на поиск «ошибки», пока не разобрался, что вся проблема состоит в злосчастной буковке.

Функция `fopen` возвращает адрес типа `FILE*`, который потом можно будет использовать при вызове других функций, если только этот адрес — не `NULL`; если же `fopen` вернула `NULL` — это свидетельствует о произошедшей ошибке. Поводов ошибиться у `fopen` достаточно: файла может не существовать, у нас может не хватать прав на его чтение и/или модификацию, либо на создание файла в указанной директории, и так далее; поэтому проверка на `NULL` после `fopen` строго обязательна.

Итак, мы добрались до такой функции, вызов которой не просто «теоретически» может кончиться ошибкой, но и в действительности настолько часто кончается ошибкой, что не проверять на ошибку просто нельзя. Коль скоро это так, самое время узнать, что же делать, если ошибка произошла, и, прежде всего, как узнать, в чём причина ошибки.

Ответ на вопрос «что делать» зависит от того, какую программу мы пишем и как далеко находимся от функции `main`. Ясно, что надо бы сообщить пользователю о произошедшем, но не во всякой программе

мы можем просто взять и напечатать сообщение. Кроме того, возникает неизбежный вопрос «что дальше». Если ошибка проявилась прямо в функции `main` или в какой-то из функций, написанных в качестве вспомогательных для `main`, можно просто завершить программу с ошибочным кодом, тогда как если ошибка возникла в недрах какого-нибудь периферийного модуля или вообще в написанной нами библиотеке, про которую мы даже не знаем, кто и как её будет использовать — то выход из программы здесь будет, разумеется, неуместен; в такой ситуации надо проинформировать вызывающего об ошибке и дать ему возможность самому решить что делать.

Так или иначе, представляется необходимой возможность узнать, в чём причина неуспешного вызова. В языке Си (точнее, в его стандартной библиотеке) этот вопрос для функций, предполагающих обращение к операционной системе[47], решается единообразно: код ошибки в виде обычного целого числа заносится в глобальную переменную `errno`. Чтобы получить доступ к этой переменной, необходимо подключить заголовочный файл `errno.h`. Возможные значения переменной `errno` имеют символические имена; например, константа `ENOENT` означает, что ошибка произошла из-за отсутствия либо самого файла, который вы пытались открыть (на чтение), либо директории, в которой этот файл должен был быть расположен; `EACCES` означает, что для открытия файла вашей программе не хватило полномочий; `EROFS` означает, что файл находится на таком диске, который можно только читать (например, на CDROM'е), а вы пытаетесь открыть его на запись, и так далее.

Для каждой библиотечной функции или системного вызова все возможные коды ошибок, которые могут произойти, перечислены в соответствующих описаниях, доступных по команде `man`; однако в большинстве случаев этих кодов оказывается слишком много, чтобы по отдельности отрабатывать их все. В частности, для `fopen` список возможных ошибок содержит 21 различный код, большинство из которых вам никогда на практике не встретится. Обычно в программах либо вообще не анализируют значение `errno`, либо проверяют её на одно-два особых значения, а все остальные ошибки обрабатывают одним махом, используя предназначенные для этого средства. Так, функция `strerror` принимает на вход код ошибки, а возвращает адрес строки (то есть выражение типа `char*`), содержащей соответствующее этому коду диагностическое сообщение; на вход этой функции мы можем подать прямо саму переменную `errno`, а полученную строку, например, напечатать.

Впрочем, если мы всерьёз собрались выдать диагностическое сообщение, то лучше выдавать его не на стандартный вывод, а в стандартный поток диагностики (`stderr`), и проще всего это сделать с помощью

[47] Сама функция `fopen` не является системным вызовом, но открыть файл без участия ОС она, разумеется, не может; свою задачу она решает, применяя системный вызов `open`, знакомый нам из части, посвящённой языку ассемблера.

функции `perror`. Эта функция имеет один параметр — строку, описывающую то, с чем возникла проблема (для файлов обычно указывают имя файла, если же проблема не с файлом, указывают, например, имя функции, вызов которой привёл к ошибке); в поток диагностики выдаётся параметр `perror`, затем двоеточие и диагностическое сообщение, выбираемое в зависимости от значения `perror`. Например, открытие файла, имеющего имя `file.txt`, мы могли бы оформить так:

```
FILE *f;
f = fopen("file.txt", "r");
if(!f) {
    perror("file.txt");
    exit(1);
}
```

Если файла с таким именем в текущей директории не окажется, в диагностический поток будет выдано:

```
file.txt: no such file or directory
```

Если в вашей системе сконфигурирована локализация, выданное сообщение может оказаться русскоязычным или, в общем случае, может быть сформулировано на том языке, который установлен в вашей системе. Не пугайтесь. Отключить непрошеную русификацию в текущем сеансе работы вы можете командой

```
export LC_ALL= LANG=
```

К сожалению, совсем «открутить» русскоязычные сообщения может оказаться несколько сложнее — это потребует редактирования системных конфигурационных файлов, конкретные имена и содержание которых зависят от используемого вами дистрибутива.

Так или иначе, попытки вашей системы общаться с вами на языке, отличном от английского, полезно сразу же пресечь. Русификации редко бывают удачными, при этом если конечный пользователь может себе позволить не знать английского, программист себе такого позволить не может: если программист не владеет английским, он заведомо профессионально непригоден. Если же вы английским владеете, русификация может доставить вам только неудобства.

Забегая вперёд, отметим, что в стандартный поток диагностики можно, разумеется, выдать произвольное сообщение, например, так (функция `fprintf` и глобальная переменная `stderr` будут рассмотрены позже):

```
fprintf(stderr, "I think there's an error\n");
```

Операция, обратная открытию файла, то есть его *закрытие*, выполняется функцией `fclose`:

```
int fclose(FILE *f);
```

На вход эта функция получает поток ввода-вывода и, собственно говоря, закрывает его, делая дальнейшие операции с ним невозможными; при этом высвобождаются ресурсы операционной системы, обеспечивавшие работу данного потока ввода-вывода[48]. Функция возвращает ноль в случае успеха и `-1`, если произошла ошибка; впрочем, поток закрывается в любом случае, остаться открытым после применения `fclose` поток не может.

Как мы увидим чуть позже, высокоуровневый ввод-вывод работает с ***буферизацией***, что, в частности, подразумевает временное хранение данных, записанных в поток, в буфере, организованном библиотекой, то есть в памяти вашей программы. Вполне возможна, например, такая ситуация: вызванная вами функция записи возвращает управление с признаком успеха (то есть не заявляет ни о каких ошибках), но данные при этом не отданы операционной системе, а всего лишь помещены в буфер. Естественно, никто не гарантирует, что позже, когда библиотечные функции всё же попытаются «слить» системе полученную информацию, не произойдёт ошибка. Если на момент вызова `fclose` в буфере имеются непереданные данные, функция, естественно, попытается их передать, прежде чем окончательно закрыть поток, и при этом вполне возможно, что произойдёт ошибка.

Любопытно будет отметить, забегая вперёд, что ровно такая же ситуация складывается при работе с вводом-выводом низкого уровня, то есть когда ввод-вывод осуществляется непосредственно системными вызовами; в этом случае свою роль играют внутренние буфера операционной системы, а ошибку может выдать системный вызов `close`.

Коль скоро поток ввода-вывода открыт, над ним можно выполнять тот же набор действий, что и над стандартными потоками ввода-вывода, то есть стандартная библиотека предусматривает для потоков типа `FILE*` набор функций, аналогичный уже рассмотренному нами для стандартных потоков. В частности, для ввода и вывода отдельных символов (`char`'ов) имеются функции `fgetc` и `fputc`:

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

совершенно аналогичные ранее рассмотренным `getchar` и `putchar`, но принимающие на один аргумент больше; этот аргумент задаёт поток, через который следует работать. Например, следующая программа, получив через аргументы командной строки имена двух файлов, откроет первый на чтение, второй на запись и запишет во второй первые десять строк из первого (если строк окажется меньше, файл окажется просто скопирован):

[48] Как мы узнаем из следующего тома, возможна ситуация, когда с одним потоком ввода-вывода связано больше одного ***дескриптора***; ресурсы ОС в этом случае освобождаются только после закрытия последнего из них. К языку Си это прямого отношения не имеет, это особенности модели ввода-вывода в системах семейства Unix.

```
/* fgetcputc.c */
#include <stdio.h>

int main(int argc, char **argv)
{
    FILE *from, *to;
    int c, lnum;
    lnum = 1;
    if(argc < 3) {
        fprintf(stderr, "Too few arguments\n");
        return 1;
    }
    from = fopen(argv[1], "r");
    if(!from) {
        perror(argv[1]);
        return 2;
    }
    to = fopen(argv[2], "w");
    if(!to) {
        perror(argv[2]);
        return 3;
    }
    lnum = 1;
    while(lnum <= 10 && (c = fgetc(from)) != EOF) {
        fputc(c, to);
        if(c == '\n')
            lnum++;
    }
    return 0;
}
```

К посимвольному вводу-выводу относится ещё одна функция, которая оказывается неожиданно полезной при анализе текстов, хотя без неё всегда можно обойтись. Это функция ungetc, которая позволяет *вернуть только что прочитанный символ обратно в поток*. Её заголовок выглядит так:

```
int ungetc(int c, FILE *stream);
```

Следующая операция чтения получит этот символ первым, как если бы он ещё не был извлечён из потока. Впрочем, тут действует целый ряд ограничений. Во-первых, вернуть можно только один символ; во-вторых, это реально должен быть символ, только что прочитанный из этого же потока. При попытке сделать что-то иное результат оказывается неопределён, то есть, возможно, где-то оно даже сработает, а где-то — нет.

Эту функцию иногда применяют при ***лексическом анализе***, когда невозможно понять, не прочитав очередной символ, является он частью текущей лексемы или нет. Например, читая из потока выражение 756+27, мы поймём, что число 756 кончилось, лишь *прочитав* символ плюс. Вполне возможно, что обработка числа у нас выделена в отдельную функцию, в которой совершенно

не хочется устраивать ещё и обработку всех доступных разделителей, тем более что разделители обрабатывать приходится, естественно, не только после чисел.

Ясно, что возврат значения в поток — не более чем хак, тем более что при грамотной реализации лексического анализатора он вообще не должен знать, откуда берутся символы для анализа. Просто в данном конкретном случае этот хак часто экономит значительные усилия, в результате чего функция `ungetc` даже вошла, как мы видим, в состав стандартной библиотеки. Использовать её или нет — решайте сами.

Над потоками возможен также и форматированный ввод-вывод. Соответствующие функции называются `fprintf` и `fscanf`, и от рассмотренных выше функций `printf` и `scanf` они отличаются только наличием дополнительного параметра типа `FILE*`, который указывается первым (то есть перед форматной строкой). Например, следующая программа запишет в файл `sincos.txt` значения синуса и косинуса для углов от 0 до 359 градусов (не забудьте подключить математическую библиотеку при компиляции! Напомним, это делается флагом `-lm`):

```
/* sin360.c */
#include <stdio.h>
#include <math.h>

int main()
{
    FILE *f;
    int grad;
    f = fopen("sincos.txt", "w");
    if(!f) {
        perror("sincos.txt");
        return 1;
    }
    for(grad = 0; grad < 360; grad++) {
        double rads = (double)grad * M_PI / 180.0;
        double s = sin(rads);
        double c = cos(rads);
        fprintf(f, "%03d % 7.5f % 7.5f\n", grad, s, c);
    }
    fclose(f);
    return 0;
}
```

В качестве следующего примера рассмотрим функцию, которая получает на вход уже открытый поток ввода и, рассматривая его как последовательность представлений целых чисел, подсчитывает, во-первых, сумму этих чисел, и, во-вторых, их количество; чтение выполняется до тех пор, пока попытка чтения очередного числа по тем или иным причинам не оканчивается неудачно. Поскольку вернуть нужно два

числа, возврат производится через параметры, а сама функция имеет тип возвращаемого значения `void`:

```
void intfilesum(FILE *f, int *sum, int *count)
{
    int n;
    *count = 0;
    *sum = 0;
    while(fscanf(f, "%d", &n) == 1) {
        *sum += n;
        *count++;
    }
}
```

Отметим ещё один момент: *любую функцию (как библиотечную, так и вашу собственную), которая рассчитана на работу с потоком типа `FILE*`, можно применить к одному из стандартных потоков ввода-вывода.* Для этого в стандартной библиотеке описаны, а в заголовочном файле `stdio.h` объявлены три глобальные переменные типа `FILE*`: `stdin` (поток стандартного ввода), `stdout` (поток стандартного вывода) и `stderr` (диагностический поток).

Например, для функции `ungetc` нет аналога, работающего со стандартным вводом, но благодаря наличию переменной `stdin` вернуть символ в поток стандартного ввода можно, написав что-то вроде `ungetc(c, stdin)`.

4.6.4. Ввод-вывод отдельных строк

Как мы уже знаем из предыдущих частей книги, при обработке текстовых данных одной из основных единиц информации оказывается *строка*, причём текстовый файл состоит из строк, разделённых ***символом перевода строки*** (символ с кодом 10). Язык Паскаль, как мы помним, предлагает даже специальные операторы (псевдопроцедуры) для записи и чтения строк целиком. Имеются аналогичные возможности и в стандартной библиотеке языка Си.

Начнём с функций, требующих явного указания потока ввода-вывода; они называются `fputs` и `fgets`, и, как можно догадаться, первая из них выдаёт заданную строку в заданный поток вывода, а вторая прочитывает строку из заданного потока ввода и размещает прочитанное в указанной области памяти:

```
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

С функцией `fputs` всё более-менее понятно; в частности, мы могли бы вывести надпись `"Hello, world\n"` не с помощью `printf`, как мы делали до сих пор, а с помощью `fputs`:

```
fputs("Hello, world\n", stdout);
```

Фундаментальное отличие `fputs` от `printf` и `fprintf` состоит в том, что содержимое строки *не интерпретируется*, `fputs` никак не обрабатывает никакие специальные символы (вроде символа «`%`», который специальным образом обрабатывается функциями форматного ввода-вывода); заданная строка просто выводится как она есть, символ за символом, пока в очередном элементе массива не окажется ноль, означающий, что строка закончилась. В принципе, если стоит задача просто вывести строку, не вставляя в вывод никакие сконвертированные в текст значения выражений, использование `fputs` оказывается эффективнее, то есть, попросту говоря, она работает быстрее, ведь ей не нужно анализировать строку; впрочем, на практике выигрыш совершенно не заметен. Возвращает функция неотрицательное число при успехе и `-1` в случае ошибки; обычно число, возвращаемое при успехе, равно длине выданной строки, но в описаниях этой функции (включая стандарты) это не является требованием.

Функция `fgets` заслуживает более тщательного обсуждения. Параметр `s` задаёт адрес области памяти, в которую вы хотите поместить прочитанную строку. Обычно в качестве такой области памяти используют обыкновенный массив `char`'ов, причём он может быть как описан статически (то есть именно в виде массива), так и выделен динамически, это не важно. Основное, что тут необходимо заметить — это что **достоверно предсказать длину строки, которая будет введена, *невозможно***, так что, сколь бы большой массив мы ни описали, его всё равно может не хватить для размещения прочитанного; если бы функция не знала, *сколько* памяти имеется в её распоряжении, то, получив слишком длинную строку, она продолжила бы записывать её символы в память, находящуюся за пределами массива, и в лучшем случае что-нибудь бы испортила (*худший* случай будет рассмотрен чуть позже). Именно для решения этой проблемы предназначен второй параметр функции, который называется `size`: через него мы сообщаем функции `fgets`, сколько элементов в том массиве, адрес начала которого передан первым параметром.

Получив свои параметры, `fgets` читает из заданного потока ввода *не более чем* `size-1` символов. Чтение прекращается, если очередной символ оказался символом перевода строки (`'\n'`, он же символ с кодом `10`), либо если очередной символ прочитать не удалось (то есть произошла ошибка или возникла ситуация «конец файла»), либо если уже прочитано максимально возможное (т. е. `size-1`) количество символов. Если причиной окончания чтения стал перевод строки, он тоже заносится в массив как обыкновенный символ. Если хотя бы один символ был прочитан, функция заносит ноль в элемент массива, непосредственно следующий за тем, в который был помещён последний из

успешно прочитанных символов, превращая, таким образом, массив в корректную строку; при этом функция в качестве своего значения возвращает тот же адрес, который она получила через параметр `s`. Если ни одного символа прочитать не удалось, то функция никак не изменяет содержимое массива `s`, а возвращает `NULL`.

Например, следующая функция читает строки из потока `f1`, после чего записывает в поток `f2` эти же строки, обрамлённые квадратными скобками; при этом строкам позволяется иметь длину не более 50 символов, если же очередная строка оказывается длиннее, в поток `f2` записывается пустая строка (без скобок).

```
void string50(FILE *f1, FILE *f2)
{
    char buf[51];
    while(fgets(buf, sizeof(buf), f1)) {
        int i;
        int nlpos = -1;
        for(i = 0; i < sizeof(buf) && buf[i]; i++)
            if(buf[i] == '\n') {
                nlpos = i;
                break;
            }
        if(nlpos == -1) {
            int c;
            fputc('\n', f2);
            while((c = fgetc(f1)) != EOF && c != '\n')
            {}
        } else {
            buf[nlpos] = '\0';
            fprintf(f2, "[%s]\n", buf);
        }
    }
}
```

Поясним, что после успешного прочтения очередной строки мы здесь пытаемся в прочитанной строке отыскать символ перевода строки `'\n'`, наличие которого свидетельствует о том, что строка была прочитана полностью. Если этого символа в строке нет (то есть переменная `nlpos` так и осталась равна `-1`) — это значит, что произошло одно из двух: либо очередная строка оказалась слишком длинной, либо ситуация «конец файла» возникла в процессе чтения строки, то есть последняя строка в потоке `f1` оказалась неполной. В приведённом примере эти две ситуации не различаются, хотя одну от другой легко отличить, например, по позиции, в которой кончается строка (то есть встречен «нулевой символ»), а ещё лучше — по наличию или отсутствию ситуации «конца файла».

В случае, если прочитанная строка не заканчивается символом перевода строки, мы предполагаем, что это была слишком длинная строка, и пытаемся выбрать из потока остаток этой строки (цикл `while` с пустым телом). Если причиной отсутствия `'\n'` стала неполная последняя строка в потоке, этот цикл просто завершится на первой же итерации из-за ситуации «конец файла». Так или иначе, при этом мы по условию задачи выводим в поток `f2` пустую строку, то есть просто символ перевода строки.

В противном случае (то есть если символ `'\n'` найден) мы выводим в поток `f2` прочитанную строку, заключённую в квадратные скобки, для чего нам сначала необходимо отсечь от неё символ перевода строки; это делается занесением в ту позицию, где до этого располагался символ перевода строки, нулевого символа.

Для обеих рассмотренных библиотечных функций имеются их «аналоги», не требующие указания потока и работающие со стандартным вводом-выводом, но эти «аналоги» на поверку оказываются весьма сомнительными. Вывести строку в стандартный поток вывода можно с помощью функции `puts`:

```
int puts(const char *s);
```

Совершенно неожиданно оказывается, что эта функция работает не так, как её «файловый» аналог: после вывода строки она зачем-то выдаёт ещё и символ перевода строки, то есть если бы нам пришло в голову выдать всё то же `"Hello, world"` с её помощью, написать надо было бы так:

```
puts("Hello, world");
```

Обратите внимание на отсутствие в строке комбинации «`\n`».

Если про `puts` можно сказать, что она несколько *странная*, но всё же иногда может быть полезна, то **за использование `gets` увольняют с работы** (!), и это ничуть не преувеличение — автор своими глазами наблюдал такое увольнение на одном из своих коммерческих мест работы.

Дело тут вот в чём. Единственный параметр, который `gets` получает на вход, задаёт адрес массива для размещения строки; не предусмотрено никакого способа сообщить, каков же у этого массива размер. Как мы уже отмечали, предсказать длину строки **невозможно**, как невозможно (никакими способами!) и гарантировать, что при очередном обращении к `gets` в потоке ввода не окажется строка, превышающая размер отведённой памяти во сколько угодно раз.

В самом лучшем случае программа, «напоровшись» на такую строку, просто «свалится»; если так получилось — считайте, вам очень крупно

повезло. Чтобы понять, каков может быть *наихудший* сценарий, придётся вспомнить структуру стекового фрейма, знакомую нам тех пор, когда мы изучали программирование на языке ассемблера. Локальные переменные размещаются в стеке, а сам стек растёт в направлении уменьшения адресов, то есть в сторону, противоположную той, куда мы перемещаемся, увеличивая, например, индекс в массиве. Иначе говоря, если мы будем перемещаться вдоль массива от его начала к концу, но не будем знать, где находится конец, и в результате продолжим перемещаться по памяти уже за границей массива, то рано или поздно мы доберёмся до места, где в стеке располагается адрес возврата из текущей функции.

Дальнейшее — дело техники, пусть и филигранной. Допустим, наша программа вводит в одной из функций локальный массив и выполняет в него чтение с помощью `gets`. Некий злоумышленник, зная об этом, подсовывает нашей программе на вход строку такой длины, что адреса возврата сразу из нескольких функций оказываются содержимым этой строки затёрты, причём не просто так, а с тщательно рассчитанным эффектом. При завершении выполнения текущей функции вместо возврата управления вызывающему произойдёт передача управления туда, куда захотел злоумышленник, что позволяет ему вызвать в нашей программе любую функцию с такими значениями параметров, с какими он пожелает. Больше того, если в нашей программе не найдётся функции, подходящей для исполнения коварных замыслов злоумышленника, он может поступить ещё хитрее: вставить в подсунутую `gets`'у строку произвольный машинный код (главное, чтобы в этом коде не встречался байт `10`, иначе функция прекратит чтение; но обойтись без такого байта проще простого), и путём затирания адреса возврата обеспечить передачу управления именно на этот код.

Иначе говоря, программу, в которой имеется вызов `gets`, можно заставить делать практически что угодно, в том числе не имеющее ничего общего с её исходным предназначением. Осталось представить, что ваша программа работает на каком-нибудь сервере или, скажем, управляет банкоматом.

Описанная техника называется ***buffer overflow exploit*** и представляет собой один из простейших методов взлома компьютерных систем; обычно именно эту технику рассказывают «для начала» начинающим «хакерам[49]», и её же описывают в учебниках по информационной без-

[49] Вообще говоря, применение слова *хакер* для обозначения компьютерных взломщиков исходно некорректно. Хакерами называют себя высококвалифицированные программисты определённого класса, в том числе Линус Торвальдс, Ричард Столлман, Эрик Реймонд и многие другие знаменитости. Журналистское понимание слова «хакер» являет собой не более чем следствие некомпетентности подавляющего большинства журналистов практически в любом предмете, о котором они пытаются писать.

опасности, если только автор такого учебника сам хоть что-то понимает в этой самой безопасности[50].

Так или иначе, **использовать функцию `gets` категорически недопустимо — никогда, ни в каких программах, ни для каких целей**. В библиотеку языка Си эта функция попала, когда ещё не было ни компьютерных сетей, ни компьютерных взломщиков, ни самого понятия компьютерной безопасности. Те времена давно прошли.

Почему-то, даже услышав всё сказанное, некоторые студенты пытаются использовать `gets`, надеясь, по-видимому, на чудо; часто можно увидеть программу, где на вход `gets` подаётся указатель, который автор программы даже не потрудился инициализировать. Разумеется, такая программа не работает, просто падает; некоторых любителей чудес даже это не останавливает. При инициализации массива нужно указать размерность, но ведь мы не знаем, какой длины будет строка. При вызове `malloc` нужно указать размер выделяемой памяти, но ведь мы не знаем, какова будет строка. При вызове `fgets` требуется указать размер буфера, но ведь мы не знаем, какой длины будет строка! И только `gets` не требует указывать никаких размеров.

Разумеется, на самом деле массив в любом случае необходимо расположить в памяти и при этом, разумеется, придётся указать его размер — **до** того, как строка будет считана. Указания размера избежать попросту *нельзя*, то есть *невозможно*, в стандартной библиотеке Си нет средств, которые бы это позволяли. Всё, чем отличается `gets` — это тем, что она совершенно спокойно допускает выход за границы массива, каков бы он ни был, и никак этому не препятствует, то есть позволяет программе завершаться аварийно, позволяет злоумышленникам затереть адрес возврата в стеке и всё такое. *Помочь* эта функция не может *ничем*; кстати, если вдруг это утверждение оставило у вас ощущение какой-то недосказанности или подозрение, что «всё на самом деле не совсем так» — значит, вы до сих пор не понимаете, о чём идёт речь, и вам срочно необходима посторонняя помощь для приведения мыслей в порядок. Не поленитесь найти кого-нибудь, к кому можно за такой помощью обратиться.

Подчеркнём, что **штатного способа прочитать строку, никак не ограничивая её длину, в библиотеке Си нет ни в каком виде**. Если необходимо такое средство, его придётся изготовить самостоятельно — например, с помощью посимвольного чтения в элементы динамически выделенного массива, с увеличением (ручным!) размера этого массива по мере необходимости. Также можно использовать другие динамические структуры данных, например, списки; что заведомо

[50] Большинство книг, в заглавии которых упоминается инфрмационная безопасность, на деле целиком посвящены криптографии, а другие аспекты безопасности полностью игнорируют. Криптография, конечно, вещь полезная, но ни от взлома, ни от угроз, связанных с человеческим фактором, она нас не спасёт.

никак не получится — так это найти что-нибудь «волшебное», которое выполнит эту работу за вас. Здесь таких чудес не предусмотрено.

4.6.5. О буферизации ввода-вывода

Как уже говорилось, общая особенность библиотечных функций высокоуровневого ввода-вывода состоит в том, что они, помимо прочего, обеспечивают *буферизацию* данных. Так, если мы вызываем функцию `getchar`, то реально прочитан будет не один байт, как мы могли бы предположить, а столько, сколько прямо сейчас можно прочитать из потока стандартного ввода, но не больше определённого количества (например, одного килобайта). В частности, если чтение реально выполняется с клавиатуры, за которой сидит живой пользователь, и этот пользователь успел уже набрать некоторое количество данных, то с хорошей степенью вероятности эти данные будут прочитаны все; иначе говоря, когда мы вызовем `getchar`, она прочитает всё, что пользователь уже набрал, при этом нам она отдаст только один байт, а остальное поместит в буфер; при следующем вызове `getchar` она вовсе не станет обращаться к операционной системе, вместо этого она отдаст нам очередной байт из буфера.

То же самое происходит при записи в поток вывода. Данные, которые основная программа передаёт на вывод через высокоуровневые функции — `fputc`, `printf` и т. п. — сохраняются в буфере до наступления одного из условий, при которых буфер должен быть вытеснен. Например, если мы решим сформировать более-менее крупный файл с помощью `fputc`, то операционной системе данные будут передаваться не по одному символу, а сравнительно большими порциями (чаще всего — по 4 Kb), что позволяет заметно ускорить выполнение программы, ведь каждое обращение к операционной системе — это изрядная потеря процессорного времени на переключении контекстов и т. п.

Заметим, что буферизация *ввода* в большинстве случаев ни в каком управлении не нуждается, и хотя некоторые возможности такого управления существуют, они почти не применяются. Дело в том, что буферизация ввода в большинстве случаев просто экономит время, не давая никакого видимого эффекта.

Совершенно иначе обстоят дела с буферизацией *вывода*. Обнаружить её присутствие — проще пареной репы; начинающие программисты при отладке программ очень часто допускают весьма характерный просчёт: вставив в программу отладочную печать, забывают при этом перевести строку. В результате их отладочные надписи, естественно, не достигают экрана, а скапливаются в буфере; если программа при этом завершается аварийно, вытеснить буфер оказывается некому, и ожидавшаяся надпись на экране так и не появляется, из чего незадачливый программист делает вывод, что до оператора с отладочной печатью управление

не дошло; вывод, понятное дело, оказывается целиком и полностью ошибочным.

Наиболее надёжный способ заставить библиотеку отдать, наконец, содержимое буфера операционной системе — это потребовать вытеснения буфера в явном виде; для этого предназначена функция `fflush`:

```
int fflush(FILE *stream);
```

Применив эту функцию к потоку *вывода*, вы тем самым потребуете немедленно очистить буфер вывода, связанный с этим потоком; данные, естественно, будут отправлены по назначению.

Интересно, что `fflush` можно применить и к потоку ввода; при этом *в большинстве реализаций* информация, хранящаяся в буфере, будет попросту сброшена. Стандартами такое поведение никак не подкреплено, так что вполне возможны реализации, не делающие этого.

Естественно, буфер вывода будет очищен, а данные переданы системе, если этот буфер заполнен и больше в него ничего не помещается. Кроме того, очистка всех буферов вывода происходит при *корректном* завершении вашей программы.

Существуют ещё две ситуации, когда буфер вывода очищается, хотя об этом никто не просил; но чтобы описать эти ситуации, нам понадобится припомнить, что поток вывода может быть связан с различными сущностями: он может идти в дисковый файл, или в канал (обычно на ввод другой программе), или в сокет для отправки по компьютерной сети, или в псевдоустройство, которое может, например, отдавать данные на принтер или ещё в какое-нибудь периферийное устройство. Среди всех этих случаев следует особо выделить один, когда вывод идёт на ***терминал*** — устройство (возможно, виртуальное), предполагающее, что по ту сторону его находится живой пользователь. Операционная система позволяет узнать, так ли это, с помощью системного вызова `isatty`. Конечно, это не даёт стопроцентной гарантии наличия «по ту сторону» живого пользователя, но это и не требуется: если какая-либо программа эмулирует работу терминала, то, следовательно, по каким-то причинам ей требуется, чтобы её партнёры по взаимодействию вели себя так, как будто работают с пользователем.

Так или иначе, *если вывод происходит на терминал*, буфер вывода вытесняется ещё в двух важных случаях: при выводе символа перевода строки и при выполнении операции ввода, причём тоже на терминале. Для потоков, не связанных с терминалом, этого не происходит.

4.6.6. «Вывод» в строку и «ввод» из строки

Возможности описанных ранее функций форматного ввода-вывода можно задействовать для формирования отформатированного текстового представления в строковом буфере в памяти, не выводя его никуда, и наоборот, для анализа информации, в таком буфере содержащейся.

Функция `sprintf`, как и `printf`, анализирует форматную строку, по мере необходимости извлекая значения из списка своих параметров (то есть из стека), но результат преобразования в текстовое представление никуда не выводится, а вместо этого складывается в предоставленный строковый буфер. Профиль этой функции такой:

```
int sprintf(char *buf, const char *format, ...);
```

Буфер, то есть массив, в котором следует сформировать результирующую строку, необходимо передать функции через первый параметр; остальные параметры такие же, как у `printf`: форматная строка, которая анализируется в точности по тем же правилам, что и для обычного `printf`, и дополнительные параметры, которые должны соответствовать присутствующим в строке директивам преобразования. Следует, разумеется, помнить, что этой функции неоткуда знать размер буфера, так что никаких проверок на эту тему она не производит; обеспечить буфер нужного размера — обязанность вызывающего. К примеру, если вы применяете директиву «`%s`» для вставки в ваш результат некой строки, относительно которой точно не знаете, какого размера она может оказаться, следует обязательно ограничить пространство, которое она может занять в вашем результате, указав перед спецификатором «точность»; например, использовать что-нибудь вроде «`%.20s`»: это гарантирует, что больше 20 элементов из буфера использовано не будет.

В современных версиях библиотеки, как правило, присутствует функция `snprintf`, принимающая ещё один параметр — размер буфера. К сожалению, эта функция в библиотеках Си появилась достаточно поздно; в стандартах она закреплена только начиная с C99. В некоторых старых версиях библиотеки эта функция присутствовала, но игнорировала свой второй параметр.

Если параметрами командной строки компилятора зафиксировать версию языка Си, имевшую место до принятия C99 (например, указать в командной строке `-ansi -pedantic`), то функция `snprintf` окажется недоступна: системные заголовочные файлы написаны так, чтобы опции компилятора влияли не только на сам язык, но и на возможности библиотеки. Это означает, что в проектах, в которых использование ANSI C является требованием, применение `snprintf` может оказаться невозможным.

Функцию `sprintf` часто используют, чтобы перевести число того или иного типа в его текстовое представление, например:

```
char str[32];
int n;
/* ... */
sprintf(str, "%d", n);
```

Здесь вызов функции `sprintf` занесёт в массив `str` текстовое представление числа, хранящегося в переменной `n`. Очень полезна функция `sprintf`, когда возникает необходимость передать некие текстовые

данные с использованием ввода-вывода *низкого уровня*, то есть непосредственно через системные вызовы и дескрипторы потоков, минуя высокоуровневую библиотеку; в этом случае данные сначала формируют в массиве типа `char` при помощи `sprintf`, а затем уже передают операционной системе.

Полезно знать, что функция `sprintf`, как и `printf`, и `fprintf`, возвращает количество «выведенных» символов, то есть, попросту говоря, длину строки, сформированной в буфере. «Нулевой» байт, помещённый в конец строки, здесь не учитывается.

Функция `sscanf` проделывает противоположную операцию, то есть анализирует строку в соответствии с заданным форматом и раскладывает полученные результаты по адресам из списка параметров. Её профиль таков:

```
int sscanf(const char *buf, const char *format, ...);
```

Правила интерпретации форматной строки в точности совпадают с таковыми для `scanf`; роль ситуации конца файла здесь играет преждевременное окончание строки `buf`, то есть если эта строка кончилась раньше, чем функция успешно проанализировала хотя бы одну директиву преобразования, возвращается `-1`.

По мнению автора этих строк, пользы от `sscanf` ещё меньше, чем от `scanf`; впрочем, автор не претендует на истину в последней инстанции.

4.6.7. Блочный ввод-вывод

Ещё во вводной части (см. т. 1, §1.6.6) мы объяснили различия между текстовым и бинарным представлением данных, а позже, обсуждая Паскаль, сами научились работать с файлами, имеющими как текстовый, так и бинарный формат. Естественно, программы, написанные на Си, тоже могут работать не только с текстовыми файлами.

Отметим, что средства посимвольного ввода и вывода (рассмотренные ранее функции `fgetc` и `fputc`) прекрасно справляются не только с потоками текста, но и с произвольными потоками байтов, в том числе не имеющими ничего общего с текстом. Определяющей причиной, требующей применения для работы с бинарными файлами специальных средств, отличных от рассмотренных «потоковых» функций, оказывается скорее не само нетекстовое представление данных, а то, что такие данные отнюдь не всегда рассматриваются как *поток*. Работа с бинарными файлами часто организуется с использованием попеременно операций чтения и записи в разные места файла; с текстовым файлом такая работа невозможна из-за непостоянства размеров строк и текстовых фрагментов, представляющих числа и другие данные, но файл, который мы не считаем текстовым, никто не мешает рассматривать как

последовательность неких записей, которые никогда не изменяют свой размер. В этом случае для каждой записи известен не только размер, но и её положение в файле (ведь размеры всех предыдущих записей тоже известны), так что можно в любой момент любую из записей как прочитать, так и перезаписать. Очевидно, что при такой работе мы рассматриваем файл не как поток или последовательность (байтов или чего бы там ни было), а скорее как некий аналог массива, только расположенный на диске.

Рассказ о стандартных функциях ввода-вывода был бы неполным без упоминания средств работы с бинарными файлами, и именно им мы посвятим этот параграф; но прежде чем продолжить, отметим, что эти средства среди программистов, пишущих на Си, не слишком популярны. **Чаще всего для работы с файлами, имеющими тот или иной нетекстовый формат, используются функции низкоуровневого ввода-вывода**, которые мы рассмотрим в следующем параграфе; вы можете сразу перейти к нему, ничего не потеряв.

Заголовочный файл `stdio.h` содержит прототипы нескольких функций, позволяющих работать с файлом как с массивом произвольных (нетекстовых) данных. Открыть файл следует, как обычно, с помощью функции `fopen`; для чтения и записи предназначены функции `fread` и `fwrite`, имеющие следующие прототипы:

```
int fread(void *ptr, int size, int n, FILE *stream);
int fwrite(const void *ptr, int size, int n, FILE *stream);
```

Для обеих функций параметр `stream` задаёт открытый файл, с которым нужно работать; параметр `n` указывает *количество элементов*, которые нужно прочитать или записать, параметр `size` задаёт *размер каждого такого элемента*. Через параметр `ptr` в функции передаётся адрес области памяти: для `fwrite` — *содержащей информацию*, которая должна быть записана в файл, а для `fread` — той, *куда следует поместить* прочитанную информацию. В обоих случаях эта область памяти обязана иметь размер не меньше чем `size`·`n` байт.

Функции возвращают количество успешно прочитанных или записанных *элементов* (не байтов!). Если произошла ошибка, функции возвращают ноль. То же самое происходит, если функция `fread`, не прочитав ни одного элемента (это важно), обнаружила ситуацию «конец файла». Ситуация «конец файла» может возникнуть, когда несколько элементов уже прочитано; в этом случае `fread` возвращает число, мéньшее, чем указанный ей параметр `n` (равное количеству успешно прочитанных элементов), а следующее обращение к `fread` уже вернёт ноль.

Отличить ошибочную ситуацию от штатной (то есть вполне нормальной, не ошибочной) ситуации «конец файла» можно с помощью функций `feof` и `ferror`:

```
int feof(FILE *stream);
int ferror(FILE *stream);
```

Функции возвращают ненулевое значение («истину») в случае, если в заданном потоке возникла ситуация «конец файла» (`feof`) или ошибка (`ferror`), в противном случае возвращается ноль («ложь»). Например, если функция `fread` вернула ноль, можно сразу же вызвать `feof`, и если она вернёт истину — то всё в порядке, просто файл кончился, если же она вернёт ложь — то что-то пошло фундаментально не так.

Изменить текущую позицию в файле можно с помощью функции `fseek`:

```
int fseek(FILE *stream, long offset, int whence);
```

Параметр `stream` задаёт открытый файл, с которым нужно работать. Параметр `offset` указывает, на сколько байтов следует сместиться, а параметр `whence` определяет, от какого места эти байты следует отсчитывать. При значении `whence`, равном константе `SEEK_SET`, отсчёт пойдёт от начала файла (значение `offset` при этом должно быть неотрицательным), при значении `SEEK_CUR` — от текущей позиции (`offset` может быть как положительным, так и отрицательным, и нулевым), при значении `SEEK_END` — от конца файла (самое забавное, что для этого случая значение `offset` тоже может быть любым, как отрицательным или нулевым, так и положительным). Функция возвращает новое значение текущей позиции, считая от начала файла.

Возвращаясь к функциям `fread` и `fwrite`, отметим, что наиболее универсальный размер «элемента» — один байт, и чаще всего именно так с этими функциями и работают, передавая параметром `size` единицу.

4.6.8. Ввод-вывод низкого уровня

Высокоуровневые средства ввода-вывода, работающие с файловыми переменными типа `FILE*`, имеют два несомненных достоинства. Во-первых, они позволяют переводить числа из внутреннего представления в текстовое (при записи в потоки ввода-вывода) и обратно (при чтении из потоков). Во-вторых, ввод-вывод через высокоуровневые потоки, как уже говорилось, *буферизуется*, что позволяет экономить на количестве дорогостоящих обращений к ядру операционной системы. Первое оказывается ценно при работе с текстовыми файлами, второе — при работе с потоками (неважно, текстовыми или бинарными); но если мы рассматриваем файл как массив неких (бинарных) записей, чередуя при этом операции чтения, записи и позиционирования, то нам оказывается не нужен ни перевод в текстовое представление (ведь формат у нас бинарный), ни буферизация, которая рассчитана на работу с потоками и не приносит пользы, если прыгать по файлу туда-сюда.

С учётом этого программисты, как уже упоминалось в предыдущем параграфе, предпочитают для «блочного» ввода-вывода использовать ***низкоуровневые средства ввода-вывода*** — функции `open`, `read`, `write`, `lseek` и `close`. В операционных системах семейства Unix эти функции, как мы знаем из §3.6.7, представляют собой *системные вызовы*, то есть они реализованы прямо в ядре операционной системы; в

других системах эти функции могут не быть системными вызовами, но работать будут точно так же.

Функции низкоуровневого ввода-вывода не используют никаких специальных типов для файловых переменных; вместо этого они различают потоки ввода-вывода *по номерам*, называемым ***файловыми дескрипторами***. Это небольшие неотрицательные целые числа: `0`, `1`, `2`, `3` и так далее. Как мы уже знаем (см. §3.6.7, стр. 145), дескрипторы `0`, `1` и `2` соответствуют стандартным потокам (соответственно ввода, вывода и диагностики).

Открывается файл вызовом `open`:

```
int open(const char *name, int mode);
int open(const char *name, int mode, int perms);
```

Параметр `name` задает имя файла, который мы хотим открыть. Как обычно, это может быть короткое имя файла, находящегося в текущем каталоге, или же путь к файлу (как полный, так и относительный). Параметр `mode` задаёт режим, в котором мы намерены работать с файлом. Режим задаётся в виде целого числа, отдельные биты которого обозначают те или иные особенности предстоящей работы с файлом; каждый такой бит представлен предопределённой целочисленной константой, а комбинации из нескольких таких битов мы получаем, применяя операцию побитового «или» («`|`»). Основными режимами являются `O_RDONLY` (только чтение), `O_WRONLY` (только запись) и `O_RDWR` (чтение и запись). Из этих трёх констант нужно указать одну и только одну.

Кроме перечисленных основных констант, существуют еще и *модифицирующие* константы, которые при необходимости можно добавить к основным, используя операцию побитового «или». К этим константам относятся:

- `O_APPEND` — открыть файл на добавление в конец: каждая операция записи будет осуществлять запись в конец файла;
- `O_CREAT` — если файла не существует, разрешить операционной системе его создание;
- `O_TRUNC` — если файл существует, перед началом работы уничтожить его старое содержимое, в результате чего длина файла станет нулевой; запись начнется с начала файла;
- `O_EXCL` (используется только в сочетании с `O_CREAT`) — потребовать от операционной системы создания нового файла; если файл уже существует, не открывать его и выдать ошибку.

Поддерживаются и другие модификаторы, но рассматривать их мы не будем.

Третий параметр (`perms`) следует задавать только в случае, если значение второго параметра предполагает возможность создания файла, то есть если используется `O_CREAT`. Этот параметр задаёт права

доступа[51] для создаваемого файла. Отметим, что из значения этого параметра система побитово вычитает число, называемое `umask`, которое контролируется пользователем и может быть установлено в удобное пользователю значение. Так, в системе, предназначенной «для своих», пользователи чаще всего устанавливают `umask` равным `022`, так что для создаваемых файлов права на запись для всех, кроме владельца, сбрасываются, а возможность читать и выполнять предоставляется всем желающим. Более осторожные пользователи могут установить `umask` равным `027`, запретив любой доступ к новосоздаваемым файлам для всех пользователей, не входящих в ту же группу, или вообще `077`, что соответствует сбрасыванию всех прав доступа для кого-либо кроме владельца.

Так или иначе, пользователь может, устанавливая параметр `umask`, сам выбрать, какой режим защиты для его файлов будет ему наиболее удобен; поэтому **при создании обычного файла данных (то есть при условии, что мы не собираемся его исполнять как программу), следует задавать значение прав доступа** `0666` — права на чтение и запись (но не на исполнение) для владельца, группы и всех остальных. Обычно права на запись для группы и остальных пользователей автоматически исчезают после вычитания `umask`, если же этого не происходит — значит, так хотел пользователь. Наряду со значением `0666` достаточно часто используется `0600`, запрещающее (независимо от значения `umask`) любой доступ к создаваемому файлу для кого-либо кроме его владельца. Так следует поступать, если создаваемый файл предназначен для хранения конфиденциальной информации — паролей, номеров банковских карточек и т. п.

Вызов `open` возвращает значение `-1`, если произошла та или иная ошибка. Если же файл успешно открылся, возвращается небольшое целое неотрицательное число — номер файлового дескриптора.

Чтение из файла (или, говоря более широко, из любого потока ввода) производится системным вызовом `read`:

```
int read(int fd, void *mem, int len);
```

Параметр `fd` задаёт файловый дескриптор; `mem` указывает на область памяти, в которой следует разместить прочитанные данные; `len` сообщает вызову размер этой области памяти, чтобы избежать её переполнения. Вызов возвращает `-1` в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество прочитанных байтов. Естественно, это число не может быть больше `len`. Особым случаем является возвращаемое значение `0`: **если `read` вернул ноль, это означает, что в заданном потоке ввода наступила ситуация «конец файла»**.

[51] Права доступа к файлам в ОС Unix мы подробно описали во вводной части, см. т. 1, §1.4.11.

Сделаем одно важное замечание. Какова бы ни была программа, использующая `read`, что бы она ни делала, анализ значения, возвращаемого `read` как функцией, строго обязателен. **Программа, вызывающая `read` ради побочного эффекта, заведомо неправильна** вне всякой зависимости от того, для чего и как она написана. Так, если в программе содержится что-то вроде

```
read(fd, buf, sizeof(buf));
```

то такая программа годится разве что на выброс. Правильный вызов функции `read` должен выглядеть примерно так:

```
count = read(fd, buf, sizeof(buf));
```

с обязательным последующим анализом значения переменной `count`.

Для **записи** в файл (или другой поток вывода) предусмотрен вызов `write`:

```
int write(int fd, const void *data, int len);
```

Параметр `fd` задаёт файловый дескриптор; `data` указывает на область памяти, содержащую данные, которые нужно записать в файл или другой поток вывода; `len` задаёт количество этих данных. Вызов возвращает `-1` в случае ошибки, а в случае успеха возвращается положительное число, означающее количество записанных байтов, естественно, не превышающее `len`. В большинстве случаев число записанных байт в точности равно значению `len`, однако полагаться на это опасно. В корректно написанной программе значение, возвращаемое вызовом `write`, обязательно должно проверяться.

После окончания работы с файлом его следует закрыть вызовом `close`:

```
int close(int fd);
```

где `fd` — дескриптор закрываемого файла. Вызов возвращает `0` в случае успеха, `-1` в случае ошибки. Напомним, что при завершении процесса все открытые файлы закрываются автоматически.

Многие программисты привыкли игнорировать значение, возвращаемое вызовом `close`; такая практика имеет под собой определённые основания, ведь, согласно описанию этого вызова, *файловый дескриптор не может после вызова `close` остаться открытым*, то есть он закрывается даже в том случае, если вызов вернул ошибку. Несмотря на это, игнорировать значение, возвращаемое вызовом `close`, не следует, особенно если мы *записывали* данные в поток, открытый на вывод или на ввод и вывод одновременно. Дело тут в том, что операционная система может ради повышения производительности применять свою собственную буферизацию: получив от нашей программы данные, которые нужно записать в файл, она сразу же вернёт управление, делая вид, что запись уже произведена,

тогда как на самом деле запись на диск произойдёт гораздо позже. Такой режим вывода называется ***асинхронным***; наша программа о применении этого режима может не знать. Если теперь какие-то проблемы с реальной записью данных произойдут уже *после последнего вызова* `write`, то момент закрытия потока (вызов `close`) останется для операционной системы последней возможностью нам об этих проблемах сообщить.

Для изменения *текущей позиции* в открытом файле используется вызов `lseek`, очень похожий на функцию `fseek`, которую мы обсуждали на стр. 299.

```
int lseek(int fd, int offset, int whence);
```

Параметр `fd` здесь задаёт номер файлового дескриптора; как и для `fseek`, параметр `offset` указывает, на сколько байтов следует сместиться, а параметр `whence` определяет, от какого места эти байты следует отсчитывать: от начала файла (`SEEK_SET`), от текущей позиции (`SEEK_CUR`) или от конца файла (`SEEK_END`). Вызов возвращает, как и `fseek`, новое значение текущей позиции, считая от начала файла. Например, `lseek(fd, 0, SEEK_SET)` установит текущую позицию на начало файла, `lseek(fd, 0, SEEK_END)` — на конец файла. Вызов `lseek(fd, 0, SEEK_CUR)` никак позицию не изменит, но его можно использовать, чтобы узнать текущее значение позиции. Прочитать последние 100 байт файла можно так:

```
int rc;
char buf[100];
/* ... */
lseek(fd, -100, SEEK_END);
rc = read(fd, buf, 100);
```

Отметим, что при смене позиции можно зайти за конец файла. Само по себе это не приводит к изменению размера файла, но если после этого произвести запись, размер файла увеличится (конечно, файл при этом должен быть открыт в режиме, допускающем запись). При этом возможно образование «дырки» между последними данными перед старым концом файла и первыми данными, записанными с новой позиции. Таким образом, например, можно создать на мегабайтной дискете файл размером в гигабайт. Это корректно, поскольку в ОС Unix такие «дырки» не заполняются реальными данными и не занимают места на диске, пока кто-нибудь не произведет операцию записи.

4.7. Избранные примеры программ

Прежде чем идти дальше, рассмотрим два интересных примера. Начнём с программы, которую мы обещали написать ещё во вводной части (см. т. 1, §1.5.5).

4.7.1. Ханойские башни

Подробно мы эту задачу уже исследовали при изучении Паскаля (см. т. 1, §2.14.2) и убедились в том, что её лучше не пытаться решать без применения рекурсии — получается сложно и некрасиво. Здесь мы ограничимся только рекурсивным решением, которое будет практически дословно повторять решение на Паскале, приведённое в первом томе на стр. 391. Выглядеть программа будет так:

```
/* hanoi.c */
#include <stdio.h>
#include <stdlib.h> /* for atoi */

static void solve(int source, int target, int interm, int n)
{
    if(n == 0)
        return;
    solve(source, interm, target, n-1);
    printf("%d: %d -> %d\n", n, source, target);
    solve(interm, target, source, n-1);
}

int main(int argc, char **argv)
{
    int n;
    if(argc < 2) {
        fprintf(stderr, "No parameter given\n");
        return 1;
    }
    n = atoi(argv[1]);
    if(n < 1) {
        fprintf(stderr, "Incorrect token count\n");
        return 2;
    }
    solve(1, 3, 2, n);
    return 0;
}
```

Подробные пояснения мы опускаем, поскольку уже привели их для программы на Паскале; обратите внимание, насколько две программы получились схожими.

4.7.2. Сопоставление строки с образцом

С задачей сопоставления строки с образцом мы уже сталкивались дважды: в §2.14.3 (см. т. 1) мы решили эту задачу на Паскале, в §3.3.9 решение той же задачи на языке ассемблера использовалось в качестве

примера организации рекурсии на уровне команд процессора. Решение на Си благодаря использованию арифметики указателей оказывается, пожалуй, несколько элегантнее, чем на Паскале; что касается ассемблерного решения, то следующий код на Си повторяет его практически слово в слово:

```
/* match_c.c */
int match(const char *str, const char *pat)
{
    int i;
    for(;; str++, pat++) {
        switch(*pat) {
            case 0:
                return *str == 0;
            case '*':
                for(i=0; ; i++) {
                    if(match(str+i, pat+1))
                        return 1;
                    if(!str[i])
                        return 0;
                }
            case '?':
                if(!*str)
                    return 0;
                break;
            default:
                if(*str != *pat)
                    return 0;
        }
    }
}
```

Впрочем, для большей ясности решения его можно разбить на две взаимно рекурсивные функции. Основная по-прежнему будет называться `match`, а цикл, организуемый при нахождении в образце звёздочки, мы выделим в отдельную функцию, которую назовём `starmatch`. Поскольку функции вызывают друг друга, для одной из них придётся сделать прототип; мы напишем прототип для `starmatch`, затем реализуем `match`, и лишь после этого напишем, наконец, тело `starmatch`:

```
/* match_c2.c */
int starmatch(const char *str, const char *pat);
int match(const char *str, const char *pat)
{
    switch(*pat) {
        case 0:
            return *str == 0;
        case '?':
```

```
            if(!*str)
                return 0;
            return match(str+1, pat+1);
        case '*':
            return starmatch(str, pat+1);
        default:
            if(*str != *pat)
                return 0;
            return
                match(str+1, pat+1);
    }
}
int starmatch(const char *str, const char *pat)
{
    int i;
    for(i=0; ; i++){
        int res = match(str+i, pat);
        if(res)
            return 1;
        if(!str[i])
            return 0;
    }
}
```

4.8. Перечислимый тип

4.8.1. Правила описания и основные возможности

Как мы уже знаем, языки программирования позволяют ввести тип данных, множество значений которого конечно и явным образом задаётся программистом в виде набора *идентификаторов*; эти идентификаторы, собственно говоря, как раз и являются возможными значениями. Такая сущность получила название «***перечислимый тип***», и при изучении Паскаля мы ним уже сталкивались. В языке Си перечислимый тип тоже присутствует и обозначается ключевым словом `enum` (от слова *enumeration* — «перечисление»), но при внимательном рассмотрении оказывается явлением совершенно иным, нежели перечислимый тип Паскаля.

Приведём для начала пример описания перечислимого типа:

```
enum colors { red, orange, yellow, green, blue, violet };
```

Здесь идентификаторы `red`, `orange` и т. д. представляют набор значений нового типа, `colors` — это ***имя перечисления***, но, как ни странно, *именем типа* этот идентификатор сам по себе не является, хотя, несомненно, новый тип здесь введён. По правилам языка Си имя только что

введённого типа состоит из *двух* слов: `enum colors`, то есть мы можем, например, описать переменную такого типа:

```
enum colors one_color;
```

Если при этом забыть слово `enum`, компилятор выдаст ошибку, поскольку типа `colors` он не знает, а знает только тип `enum colors`[52]. По идее, переменная `one_color` теперь должна была бы быть способна принимать только значения, явным образом указанные в описании типа, а сами эти значения — `red`, `green` и прочие — можно было бы заносить только в такие переменные; но не тут-то было, ведь Си не Паскаль.

В отличие от Паскаля, в Си идентификаторы, введённые в качестве возможных значений в описании перечислимого типа, **считаются целочисленными константами**, то есть имеют тип `int`. Если не предпринять специальных мер, то самая первая константа из описания перечисления, в данном случае `red`, будет иметь значение `0`, а каждая следующая окажется на единицу больше: `orange` будет равна `1`, `yellow` — `2` и т. д., `violet` получит значение `5`. Важно понимать, что эти идентификаторы действительно становятся «честными синонимами» соответствующих чисел, то есть их можно присваивать целочисленным переменным, можно использовать в арифметических выражениях и т. д.: например, выражение `violet * green - blue` компилятор проглотит без звука, получится то же самое, как если бы мы написали `5 * 3 - 4`, то есть `11` (это, разумеется, не значит, что так действительно стоит делать; наоборот, так делать не надо). Больше того, и в переменную `one_color` можно занести любое целое число, поскольку с точки зрения разрядности и знаковости она представляет собой обыкновенный `int`, но здесь компилятор хотя бы выдаст предупреждение.

Многие программисты, пишущие на Си, в особенности те, кто никогда не писал на более строгих языках вроде того же Паскаля, предпочитают вообще не вводить переменные и функции типа `enum`, обходясь обычными `int`'ами, а описания перечислимых типов используют только для задания констант. Такая практика вряд ли заслуживает одобрения, ведь тем самым мы лишаем компилятор возможности выдать нам пусть не ошибку, но хотя бы предупреждение в сомнительной ситуации. Тем не менее этот подход оказался столь популярен, что при описании типа `enum` в Си даже разрешили не указывать имя перечисления, то есть писать что-то вроде

```
enum { aplha, beta, gamma };
```

[52] Такое же точно именование из двух слов — ключевого, указывающего разновидность типа, и идентификатора самого типа — применяется в Си также для составных типов, которые мы будем рассматривать в следующей главе. Почему и зачем создатели языка Си пошли этим довольно неочевидным путём, сказать трудно; в частности, в языке Си++, появившемся на десяток лет позже, идентификаторы перечислений, структур и объединений, а также классов (фирменной особенности Си++) стали полноценными именами типов, но «чистый» Си от такого «двухсловного» именования пользовательских типов так и не отказался.

пропустив при этом идентификатор, относящийся к самому типу. Ясно, что способа описать переменную или функцию такого типа у нас не будет.

Значения констант, вводимых в описании перечислимого типа, можно задать явно; так, при описании цветов мы могли бы поступить хитрее, чем в рассмотренном примере:

```
enum colors {
    red      = 0xff0000,
    green    = 0x00ff00,
    blue     = 0x0000ff,
    yellow   = 0xffff00,
    cyan     = 0x00ffff,
    magenta  = 0xff00ff,
    black    = 0,
    white    = 0xffffff,
    grey     = 0x808080,
    violet   = 0xee82ee
};
```

Здесь мы не просто ввели константы для обозначения цветов, но и снабдили их значениями, соответствующими представлению этих цветов в модели RGB; разумеется, это существенно расширяет область применения введённых констант.

Две константы из одного и того же перечисления могут иметь одинаковые значения, Си это разрешает. Например, мы могли бы добавить в наше перечисление ещё одну константу:

```
    aqua     = 0x00ffff,
```

которая, как можно заметить, имеет такое же значение, как и `cyan`.

В одном описании можно смешивать константы с указанием значения и без такового; константа, значение которой не указано, будет на единицу больше предыдущей. Например, если написать

```
enum numbers { one = 1, two, three, first = 1, second, third };
```

то идентификаторы `two` и `second` будут иметь значение 2, а `three` и `third` — значение 3. При указании значения справа от знака равенства можно использовать любую константу времени компиляции, в том числе арифметическое выражение, которое компилятор может вычислить, например:

```
enum example_enum {
    example_first = 100,
    example_second = example_first * 20,
    example_third = example_first + 1000,
    example_last = example_third
};
```

Здесь введённые константы получат значения 100, 2000, 3000 и 3000.

Отдельного замечания заслуживает **точка с запятой после закрывающей фигурной скобки** в описаниях перечислимых типов. Для Си подобное сочетание не вполне характерно и встречается всего в двух случаях: при описании пользовательских типов и при описании переменных со сложными инициализаторами; мы уже видели эту ситуацию в примерах инициализаторов массивов.

На самом деле это не две различные ситуации, а одна: точкой с запятой в Си должно заканчиваться любое объявление или описание, за исключением *описания функции*, в котором вместо точки с запятой фигурирует тело функции. Больше того, в Си вообще не различаются конструкции описания типа и описания переменной; коль скоро появился тип, можно указать и имена переменных, например:

```
enum states {
    running, blocked, ready
} new_state, last_state;
```

Здесь одно описание вводит сразу тип `enum states`, три целочисленные константы `running`, `blocked` и `ready`, а также две *переменные* `new_state` и `last_state`, обе имеющие тип `enum states`. Точку с запятой, которую мы ставили раньше в описаниях `enum`'ов сразу после закрывающей фигурной скобки, можно рассматривать как наш отказ от возможности немедленно описать переменные для только что введённого типа, но на самом деле она просто обозначает конец конструкции *описания*, которая в общем случае состоит из спецификатора типа и списка описываемых переменных (возможно, пустого). Формальный синтаксис Си позволяет оставить список переменных пустым в любом описании, в том числе и таком, которое никаких новых типов не вводит; например, строка

```
int ;
```

не является ошибочной с формально-синтаксической точки зрения, это обычное *описание* с пустым списком переменных. Конечно, такое описание *бессмысленно*, и большинство компиляторов выдаст здесь предупреждение — но не ошибку.

Так или иначе, если забыть поставить точку с запятой после описания типа, диагностика, которую вы получите от компилятора, скорее всего, будет настолько невразумительной, что у начинающего практически нет шансов с ходу понять, в чём на самом деле состоит проблема. Здесь можно дать сразу два совета. Во-первых, **будьте внимательны и не забывайте точку с запятой в конце описания нового типа**, отнеситесь к этому моменту чуть более серьёзно, чем к остальным вывертам языка Си, требующим внимательности. Во-вторых, попробуйте

(можно прямо сейчас) сделать эту ошибку преднамеренно и посмотрите, какова будет диагностика, выданная вашим компилятором. Есть шанс, что после этого, когда то же самое произойдёт непреднамеренно, вы вспомните, что такую диагностику уже встречали, и потратите меньше времени на выявление источника проблемы.

4.8.2. Перечислимый тип как средство описания констант

При изучении Паскаля мы отмечали, что наличие в программе явным образом указанных чисел крайне нежелательно, за исключением совершенно очевидных случаев с использованием `0`, `1`, очень редко `2`, иногда ещё `-1`. Все остальные числовые константы должны быть снабжены именами, и в программе следует использовать имена, а не числа. Паскаль для этого предусматривает секцию описания констант.

В ранних версиях Си для именования константных значений приходилось использовать макропроцессор, никаких иных средств для введения констант не было. Подробный разговор о макросах Си у нас ещё впереди, пока же отметим, что макропроцессор отрабатывает после лексического анализа, но раньше синтаксического, в результате чего идентификаторы, введённые как макросы, не подчиняются областям видимости, в частности, не локализуются в функциях и т. п., и вообще они во многом крайне неудобны.

Введение в язык перечислимого типа, разумеется, не имело самостоятельной целью создание средства для описания констант; факт, однако, состоит в том, что в Си это единственный способ описать корректный идентификатор, подчиняющийся соглашениям об областях видимости, но при этом представляющий собой *константу времени компиляции*. Вскоре после появления в языке перечислимого типа этот факт был осознан и им начали активно пользоваться. В современных программах на Си можно часто встретить что-то вроде

```
enum { max_buffer_size = 1024 };
```

Очевидно, что в качестве «перечислимого типа» такая конструкция не имеет ни малейшего смысла: никому не нужен тип, выражения которого могут принимать всего одно значение. Цель такого описания — совершенно в другом: создать *имя* (в данном случае `max_buffer_size`), которое будет в программе использоваться для обозначения некой константной величины, избавляя программиста от необходимости явно упоминать число `1024`, причём, возможно, в нескольких местах, что может обернуться феерическими проблемами в случае принятия решения об изменении константы (в данном случае — об изменении максимального размера какого-то буфера), например, с `1024` на `2048`. Читателя программы применение константы избавляет от необходимости каждый

раз при виде числа 1024 судорожно пытаться понять, что же имелось в виду под числом 1024 и почему используется именно такое число, а не какое-то другое; при применении имени вместо числа становится более-менее понятно, что здесь имеется в виду какой-то размер буфера, а если непонятно, что это за буфер и почему у него именно такой размер, а не иной, можно найти (обычным текстовым поиском) то место, где этот идентификатор введён; скорее всего, там окажется подобающий комментарий.

Этот способ введения констант имеет одно весьма существенное ограничение: так можно ввести только *целочисленную* константу, и никакую другую; для других приходится по-прежнему использовать макропроцессор. Всё-таки средство (перечислимые типы) здесь применяется не по его исходному назначению, так что тут и жаловаться, в общем, не на что. Но даже с учётом этого ограничения и при наличии чёткого понимания, что сие есть откровенный *хак*, такой способ введения констант имеет несомненные достоинства в сравнении с применением макропроцессора и, естественно, широко применяется.

Отдельно стоит заметить, что в большинстве программ, а равно и во многих книгах, посвящённых Си, включая книгу Кернигана и Ритчи, имена таких констант записывают в верхнем регистре, то есть всеми большими буквами — что-то вроде MAX_BUFFER_SIZE. В главе, посвящённой макропроцессору, мы подробно обсудим, откуда взялась такая странная традиция и по каким причинам ей совсем не стоит следовать, когда речь идёт о константах, вводимых через enum.

4.8.3. Перечислимый тип и оператор выбора

Поскольку константы-значения перечислимого типа представляют собой, как уже говорилось, *константы времени компиляции*, их можно (и нужно) использовать в составе case-меток оператора switch. При этом выражение в заголовке оператора может, конечно, иметь обычный целочисленный тип, но если всё-таки постараться и сделать так, чтобы это выражение было типа enum — использовать переменную типа enum или функцию, возвращающую значение этого типа, — то мы получим дополнительную поддержку со стороны компилятора: он возьмёт на себя работу по проверке, *все ли возможные значения мы обработали*.

Говоря формально, компилятор предполагает, что если выражение, по которому проводится выбор в операторе switch, имеет перечислимый тип, то в теле switch должны присутствовать case-метки для всех *различных* значений, предусмотренных для данного перечислимого типа, либо, по крайней мере, должна присутствовать метка default; если это не так, компилятор выдаёт предупреждение. Пусть, например, у нас описан перечислимый тип

```
enum greek { alpha, beta, gamma = beta, delta };
```

и где-то в программе встречается функция

```
void greek_print(enum greek x)
{
    switch(x) {
    case alpha:
        printf("Alpha\n");
        break;
    case beta:
        printf("Beta\n");
        break;
    }
}
```

При компиляции этой функции компилятор выдаст предупреждение о том, что значение перечислимого типа `delta` не обработано в операторе `switch`; заметим, про значение `gamma` компилятор ничего не скажет, поскольку оно совпадает с `beta`, а значение `beta` в операторе обработано.

Избавиться от этого предупреждения можно очень просто: поставить метку `default` в самом конце оператора `switch`, а после неё, чтобы удовлетворить синтаксическим требованиями, предусмотреть пустой оператор (например, просто точку с запятой):

```
      /*   ...    */
    case beta:
        printf("Beta\n");
        break;
    default:
        ;
    }
}
```

Однако гораздо лучше и правильнее будет не «избавляться от предупреждения», а внимательно изучить ситуацию, чтобы понять, почему же так получилось, что в операторе есть альтернативы для всех вариантов, предусмотренных нашим перечислимым типом, кроме одного. Очень часто это происходит потому, что в описание типа был добавлен новый вариант значения, но при этом в соответствующий оператор `switch` его обработку добавить *забыли*. Предупреждение, выдаваемое компилятором, призвано оповестить программиста об этой ошибке, возможно, избавив от долгой и мучительной отладки в будущем; отмахиваться от таких предупреждений попросту глупо, ведь выйдет в итоге, скорее всего, себе дороже. Больше того, в ситуациях, когда выражение в заголовке `switch` имеет перечислимый тип, рекомендуется вообще воздерживаться от применения метки `default`, перечисляя все возможные альтернативы (которых обычно не так много) в явном виде; если для части альтернатив, к примеру, делать ничего не нужно, их следует сгруппировать в

конце тела `switch`, там, где вы в противном случае поставили бы метку `default`. Сгенерированный машинный код ничем отличаться не будет, но читателю вашей программы вы при этом недвусмысленно покажете, что ничего не забыли.

Отметим ещё один важный момент. В программах начинающих программистов часто можно встретить что-то похожее на

```
switch(n) {
case 1:
    /* ... */
case 2:
    /* ... */

    /* ... */

case 13:
    /* ... */

}
```

Ни в одном серьёзном проекте такой код не имеет шансов пройти контроль качества; больше того, как только такое увидит ваш более опытный коллега, эффект вам, скорее всего, не понравится. С работы, пожалуй, не уволят (хотя случается всякое), но широкого обсуждения ваших профессиональных качеств, в просторечии именуемого «публичной поркой», вам точно не избежать, и, конечно же, такой код неизбежно придётся переделать. В подобных случаях использование `enum` вместо чисел **обязательно**, более того, для таких вещей `enum` как раз и предназначен.

Чтобы понять, почему это так, задайте себе простой вопрос: а что такое, простите, `13`, какую ситуацию это число обозначает, чему соответствует? Будучи автором кода, вы, скорее всего, ответ найдёте почти сразу — даже если вы его не помните, то, вероятно, хорошо помните, куда надо посмотреть, чтобы вспомнить. А теперь поставьте себя на место стороннего читателя вашей программы и прикиньте, сколько у него займёт времени поиск в вашем тексте соответствущей информации. Больше того, вы и сами можете легко перепутать, например, число `13` и число `12`, использовав их в одном месте одним способом, а в другом — другим. Если дело дойдёт до анализа вашего кода на предмет его правильности, придётся в каждом из мест, где встречается загадочная нумерация, проверять, не перепутал ли автор свои собственные номера, на что может уйти масса времени и сил. А если **вместо загадочного числа `13` в программе использовать имя константы**, описывающее то, чему оно соответствует, все перечисленные проблемы вообще не возникают.

4.9. Составной тип данных и динамические структуры

При изучении Паскаля мы уже сталкивались со связными динамическими структурами данных, такими как односвязный и двусвязный список, двоичное дерево поиска и хеш-таблица. Для их построения, кроме указателей, необходим ***составной тип данных***, который в Паскале называется «запись» (`record`); в этой главе мы введём аналог паскалевской записи, который в языке Си называется «структурой» и обозначается ключевым словом `struct`.

4.9.1. Структуры

Как мы уже знаем, ***составной тип данных*** предполагает, что *переменная* этого типа состоит из нескольких переменных других типов, причём в общем случае различных — в этом (безотносительно используемого языка программирования) состоит ключевое отличие переменных составного типа от массивов. Элементы составного типа, то есть те переменные, из которых состоит переменная составного типа, называются ***полями***[53].

В языке Си для создания классических составных типов применяется понятие ***структуры***, вводимое ключевым словом `struct`. Общий подход к синтаксису описания здесь тот же, что и для уже знакомых нам перечислимых типов: сначала ключевое слово (здесь `struct`, для перечислимых типов это было слово `enum`) извещает компилятор, какого рода тип будет описываться, затем следует *необязательное* имя — идентификатор, который вместе с предшествующим ключевым словом составит имя типа; после этого в фигурных скобках следует информация о типе, для структурного типа это перечисление полей, составляющих структуру; завершает описание список вводимых переменных, возможно, пустой:

`struct` [<*имя*>] `{` <*список_ полей*> `}` [<*список_ перем.*>] `;`

Как и в Паскале, для доступа к полям структуры в Си используется точка, то есть если `s1` — это имя переменной структурного типа и у этой структуры есть поле `f`, то `s1.f` обозначает соответствующую часть переменной `s1`.

Например, создавая базу данных для учебного заведения, мы могли бы заметить, что относительно каждого *студента* нас интересует его имя, пол, год рождения, код специальности, номер курса, номер группы (который на самом деле лучше хранить в виде небольшой строки, а не числа, потому что такие номера часто включают в себя дополнительные буквы), а также его средний балл, который можно хранить в виде

[53] Соответствующий английский термин — *fields*; как видим, перевод в данном случае совершенно точный.

числа с плавающей точкой. Структура, хранящая всю перечисленную информацию, могла бы быть описана, например, так:

```
enum { max_name_len = 64, max_group_len = 8 };
/* ... */
struct student {
    char name[max_name_len];
    char sex;   /* 'm' or 'f' */
    int year_of_birth;
    int major_code;
    int year;
    char group[max_group_len];
    float average;
};
```

Подчеркнём, что `student` здесь — так называемое ***имя структуры***, которое само по себе не является именем типа; введённый тип имеет имя, состоящее из двух слов `struct student`. Например, описать переменную этого типа мы можем так:

```
struct student st1;
```

Теперь `st1` — это переменная, которая *состоит* из переменных меньшего размера, называемых ***полями***. Доступ к полям осуществляется через точку; например, воспользовавшись одной из версий функции `string_copy`, описанных ранее (см. стр. 263), мы могли бы заполнить эту переменную информацией:

```
string_copy(st1.name, "Otlichnikov Vasiliy Sergeevich");
st1.sex = 'm';
st1.year_of_birth = 1995;
st1.major_code = 51311;
st1.year = 3;
string_copy(st1.group, "312");
st1.average = 4.792;
```

Переменные структурного типа можно инициализировать при их описании, например:

```
struct student st1 = {
    "Otlichnikov Vasiliy Sergeevich",
    'm', 1995, 51311, 3, "312", 4.792
};
```

Отметим, что подобные конструкции допустимы при *инициализации*, но не при *присваивании* (см. рассуждение на стр. 258); с другой стороны, присваивать структурные переменные друг другу, в отличие от массивов, можно. Более того, структуры можно передавать в функции в

качестве параметров, а также возвращать из функций; это, впрочем, не означает, что так действительно нужно делать, за исключением случаев совсем небольших структур; если ваша структура имеет более-менее существенный объём, передавать в функцию лучше всё-таки её адрес. Что касается возврата из функции, то обычно делают иначе: в функцию передают адрес структурной переменной, и функция заполняет её поля нужной информацией.

Интересно, что итоговый размер переменной структурного типа может оказаться больше, чем суммарный размер всех её элементов. Дело тут в том, что переменные (т. е. области памяти), занимающие больше одного байта (в общем случае — больше одной ячейки памяти) на многих аппаратных архитектурах могут начинаться в памяти только с определённых адресов, чаще всего — чётных. Например, процессоры Sparc вообще не поддерживают операции извлечения из памяти двухбайтных и четырёхбайтных чисел по нечётным адресам; процессоры интеловской архитектуры, с которыми мы чаще всего работаем, такое извлечение допускают, но оно работает заметно медленнее, нежели при использовании чётных адресов. Может оказаться, что четырёхбайтные целые в силу тех или иных причин должны располагаться по адресам, кратным четырём, и так далее.

Компилятор всё это учитывает и вставляет в переменную структурного типа так называемые ***выравнивающие байты*** (англ. *padding bytes*), в которых не будет храниться никакая информация; единственное предназначение этих байтов — *занимать место* так, чтобы очередное поле начиналось с адреса, кратного двум или четырём. Так, в структуре `student` из нашего примера компилятор обязательно вставит выравнивающий байт после поля `sex`, причём в зависимости от конкретной архитектуры и версии компилятора такой байт там может оказаться один (следующее поле начнётся с чётного адреса) или даже три (следующее поле начнётся с адреса, кратного четырём).

Это явление называется ***выравниванием*** (англ. *alignment*, читается «элайнмент»), и у него есть два очень важных следствия. Во-первых, мы в общем случае не можем знать, сколько ячеек памяти будет занимать переменная структурного типа; единственный достоверный источник такой информации — уже знакомая нам псевдофункция `sizeof`, которая позволяет воспользоваться знанием о размере, которым обладает сам компилятор. Например, `sizeof(struct student)` будет при компиляции заменено целым числом, означающим размер этой структуры в памяти; если, к примеру, нам нужно выделить динамическую память под размещение такой структуры, правильно это сделать так:

```
struct student *ptr;
ptr = malloc(sizeof(struct student));
```

Кстати, если такое выражение кажется слишком громоздким, можно воспользоваться тем, что `sizeof` работает не только для имён типов, но и для произвольных выражений (значение выражения в этом случае игнорируется), а выражение `*ptr` как раз имеет нужный нам тип `struct student`; с учётом этого строчку с вызовом `malloc` можно переписать так:

```
ptr = malloc(sizeof(*ptr));
```

Если вы не поняли, что произошло, не паникуйте и пишите так, как понятнее.

Указатели (и, в общем случае, адресные выражения) на переменные структурного типа используются в Си настолько часто, что для работы с ними даже придумана специальная операция. Пусть у нас некий указатель `p` указывает на структуру типа `struct student`, то есть он описан как

```
struct student *p;
```

Если мы хотим теперь, используя этот указатель, обратиться к полю структуры (например, к полю `year`), то прежде чем применять операцию «точка», нам нужно сделать из указателя на структуру, собственно говоря, саму структуру; это несложно, достаточно применить операцию разыменования: `*p` и есть нужная нам структура. Но вот прежде чем применять точку к выражению `*p`, стоит вспомнить, что точка представляет собой *унарную операцию*, то есть операцию, у которой всего один операнд: в самом деле, имя поля считать операндом невозможно, ведь оно не является выражением какого-либо типа, его нельзя вычислить, его нельзя вернуть из функции и т. п.; правильнее всего считать, что символ точки — это ещё не операция, а вот `.year` — это операция (извлечения поля `year`), причём она, очевидно, унарная и имеет суффиксную форму. Ранее мы обсуждали, что *суффиксные унарные операции имеют приоритет более высокий, нежели префиксные* (см. стр. 234). Следовательно, если написать (как это часто пытаются сделать начинающие) `*p.year`, мы получим ошибку: операция «`.year`» будет применена не к `*p`, а к самому `p`, но ведь `p` — это просто указатель, никаких полей у него нет. Приходится вмешаться в приоритетность операций, применив круглые скобки, и выглядит это так: `(*p).year`.

В принципе, никто не мешает прямо так и писать в программах, но, поскольку в серьёзных программах на Си такая ситуация встречается очень часто, а выражение такого вида выглядит громоздко и непонятно, в язык была добавлена операция «стрелка» («`->`»). **Выражение вида `a->b` означает то же самое, что и `(*a).b`**; при этом, разумеется, «`b`» должно быть именем поля, а «`a`» — адресом структуры, в которой такое поле есть. Вместо `(*p).year` мы можем, следовательно, написать `p->year`, что выглядит не так ужасно.

4.9.2. Односвязные списки

С простейшей из всех «связных» динамических структур данных — ***односвязным списком*** — мы уже знакомы, на Паскале мы с этой сущностью работали. Напомним, что под односвязным списком понимается связная структура данных, состоящая из отдельных ***звеньев***, имеющих составной тип, одно из полей которого представляет собой указатель на следующий элемент списка; в последнем элементе списка этот указатель имеет нулевое значение (`nil` для Паскаля, `NULL` для Си).

Для работы с односвязным списком используют **указатель на его первый элемент** и дополнительные указатели, где это необходимо; нулевое значение в указателе на первый элемент означает ситуацию «пустой список».

В частности, список целых чисел можно построить из таких звеньев:

```
struct item {
    int data;
    struct item *next;
};
```

Для примера рассмотрим задачу построения списка целых чисел по заданному массиву целых чисел; решение оформим в виде функции, которая будет принимать на вход адрес массива и его длину, а возвращать адрес первого элемента построенного списка. Начнём с варианта решения, при котором новые элементы заносятся *в конец* списка, для чего необходимо хранить два указателя: на первый элемент списка и на последний; их мы назовём `first` и `last`. Кроме того, нам потребуется временный указатель, который мы назовём `tmp` (от слова *temporary*), и целочисленная переменная для организации цикла по элементам массива.

```
struct item *int_array_to_list(int *arr, int len)
{
    struct item *first = NULL, *last = NULL, *tmp;
    int i;
    for(i = 0; i < len; i++) {
        tmp = malloc(sizeof(struct item));
        tmp->data = arr[i];
        tmp->next = NULL;
        if(last) {
            last->next = tmp;
            last = last->next;
        } else {
            first = last = tmp;
        }
    }
    return first;
}
```

Как видим, на каждом шаге цикла мы сначала создаём очередной элемент списка и заполняем его поля; вообще следует отметить, что заполнять поля свежесозданной структуры данных желательно сразу после её создания. Затем в зависимости от того, первый это элемент или нет, мы либо добавляем элемент «в хвост» имеющемуся списку (при этом его начало никак не затрагивается), либо, если список пустой, делаем его состоящим из единственного элемента.

Решение можно существенно упростить, если вспомнить, что добавление *в начало* односвязного списка, вообще говоря, гораздо проще, чем добавление в его конец; в частности, рассматривать отдельно случай пустого списка при этом не нужно. Элементы массива мы можем перебрать в обратном порядке ничуть не хуже, чем в прямом, так что для нашей задачи добавление в начало подходит ничуть не хуже. Выглядеть это будет примерно так:

```
struct item *int_array_to_list(int *arr, int len)
{
    struct item *first = NULL, *tmp;
    int i;
    for(i = len-1; i >= 0; i--) {
        tmp = malloc(sizeof(struct item));
        tmp->data = arr[i];
        tmp->next = first;
        first = tmp;
    }
    return first;
}
```

Наконец, можно вспомнить, что задачи, связанные со списочно-древесными структурами данных, обычно легко решаются с помощью рекурсии. На Си такое решение будет выглядеть более естественно, чем на Паскале, благодаря тому, что любой фрагмент (сегмент) массива в Си может рассматриваться как самостоятельный массив (в Паскале такого нет). Базис рекурсии оказывается тривиальным: если массив пустой, следует вернуть пустой список. Если же массив не пуст, то следует создать элемент списка для хранения числа из *первого* элемента массива, после чего, обратившись рекурсивно к своей же функции, построить остаток списка, рассматривая все элементы массива, кроме первого, как массив на один элемент короче. Выглядит это так:

```
struct item *int_array_to_list(int *arr, int len)
{
    struct item *tmp;
    if(!len)
        return NULL;
    tmp = malloc(sizeof(struct item));
    tmp->data = *arr;
    tmp->next = int_array_to_list(arr + 1, len - 1);
    return tmp;
}
```

На всякий случай для читателей, всё ещё не обретших уверенности с адресной арифметикой, поясним, что `arr + 1` — это адрес того же массива, только начиная с его второго элемента.

Не менее интересна с иллюстративной точки зрения другая задача — перебор всех элементов списка. Для примера напишем функцию, вычисляющую *сумму всех элементов* списка, для которого известен адрес первого элемента. Если применять подход, уже знакомый нам по Паскалю, эта функция может выглядеть, например, так:

```
int int_list_sum(const struct item *lst)
{
    int sum = 0;
    const struct item *tmp = lst;
    while(tmp) {
        sum += tmp->data;
        tmp = tmp->next;
    }
    return sum;
}
```

— однако в программах на Си то же самое обычно записывают иначе, с помощью цикла `for`:

```
int int_list_sum(const struct item *lst)
{
    int sum = 0;
    const struct item *tmp;
    for(tmp = lst; tmp; tmp = tmp->next)
        sum += tmp->data;
    return sum;
}
```

Учитывая, что параметр `lst` — это тоже локальная переменная и менять её никто не запрещает, можно сделать функцию ещё короче:

```
int int_list_sum(const struct item *lst)
{
    int sum = 0;
    for(; lst; lst = lst->next)
        sum += lst->data;
    return sum;
}
```

Впрочем, даже здесь остаётся пространство для дальнейшего совершенствования. Вспомнив старую добрую рекурсию, мы можем вообще обойтись без локальных переменных и циклов:

```
int int_list_sum(const struct item *lst)
{
    if(lst)
        return lst->data + int_list_sum(lst->next);
```

```
    else
        return 0;
}
```

Наконец, применив условную операцию, мы благополучно превращаем нашу функцию в так называемый «one-liner» (однострочник):

```
int int_list_sum(const struct item *lst)
{
    return lst ? lst->data + int_list_sum(lst->next) : 0;
}
```

Почти всегда, когда мы создали список, возникает вопрос, как его удалить, то есть как убрать из памяти все его элементы. Рассмотрим два очевидных решения, первое:

```
void delete_int_list(struct item *lst)
{
    while(lst) {
        struct item *tmp = lst;
        lst = lst->next;
        free(tmp);
    }
}
```

— и очень похожее второе:

```
void delete_int_list(struct item *lst)
{
    while(lst) {
        struct item *tmp = lst->next;
        free(lst);
        lst = tmp;
    }
}
```

В обоих случаях временная переменная помогает нам решить проблему нарушения связности списка при удалении его первого элемента: в самом деле, если удалить первый элемент, то обращаться к нему больше нельзя, а адрес второго элемента есть только в одном месте — в поле первого элемента; с другой стороны, если сразу переставить указатель на второй элемент, то мы потеряем адрес первого. Первое из вышеприведённых решений состоит в том, что во временном указателе сохраняется адрес первого элемента, основной указатель переставляется на второй элемент, после чего первый удаляется, для чего используется его адрес, сохранённый во временном указателе. Второе решение работает иначе: во временном указателе сохраняется адрес второго элемента, первый

удаляется с использованием основного указателя, затем в основной указатель заносится адрес следующего элемента, ранее сохранённый во временном.

Как водится, с использованием рекурсии то же самое делается несколько изящнее. В качестве базиса выбирается случай пустого списка: тогда нам делать нечего, возвращаем управление сразу же; в противном случае удаляем «хвост» списка, вызвав сами себя, а затем отдельно удаляем «голову» списка:

```
void delete_int_list(struct item *lst)
{
    if(!lst)
        return;
    delete_int_list(lst->next);
    free(lst);
}
```

Можно это записать и иначе, если не ставить целью обязательное текстуальное выделение базиса рекурсии:

```
void delete_int_list(struct item *lst)
{
    if(lst) {
        delete_int_list(lst->next);
        free(lst);
    }
}
```

В заключение рассмотрим задачу *удаления из заданного списка всех элементов, удовлетворяющих определённому условию*; к примеру, попытаемся удалить из нашего списка целых чисел все элементы, в которых хранятся отрицательные числа. Для начала не будем оформлять решение в виде функции; условимся, что у нас есть указатель `first`, хранящий адрес первого элемента, и напишем фрагмент кода, который выкинет все отрицательные числа из такого списка.

Проблема в том, что для исключения элемента из списка *необходимо модифицировать тот указатель, который указывает на удаляемый элемент*, причём это может оказаться как поле `next` одного из элементов (предшествующего удаляемому), так и указатель `first` (в случае, если нужно удалить самый первый элемент). В некоторых учебниках по программированию рассматривается такая странная конструкция, как *список с ключевым звеном*, при этом от «ключевого звена» используется только его поле `next` (в той роли, в которой мы используем отдельный указатель `first`), и всё это лишь для того, чтобы для любого из *используемых* звеньев можно было рассматривать «указатель на предыдущий элемент» (для первого элемента таким «предыдущим» оказывается как

раз «ключевое звено»). Нечего и говорить, что в реальной жизни такие решения не применяются.

Настоящее «боевое» решение состоит в том, чтобы хранить *адрес указателя на текущий элемент*. В этом случае, если нам придётся уничтожать текущий элемент, мы будем точно знать, какой указатель следует изменить, заставив указывать туда же, куда указывает поле `next` удаляемого элемента. В роли «указателя на текущий» у нас будут выступать последовательно указатель `first`, поле `next` первого элемента, поле `next` второго элемента и так далее; иначе говоря, нам нужен такой указатель, в котором сначала будет лежать *адрес указателя* ***`first`***, затем *адрес поля* ***`next`*** *из первого элемента списка*, затем из второго и так пока список не кончится, то есть не окажется. что указатель, адрес которого мы храним, равен `NULL`.

Нашу переменную цикла мы назовём `pcur` и опишем так:

```
struct item **pcur;
```

Сам цикл будет выглядеть следующим образом:

```
pcur = &first;          /* сначала pcur указывает на first */
while(*pcur) {   /* пока то, на что он указывает, не NULL */
    if((*pcur)->data < 0) {  /* если элемент надо удалить */
        struct item *tmp = *pcur;  /* запоминаем его адрес*/
        *pcur = (*pcur)->next;     /* исключаем из списка */
        free(tmp);                              /* удаляем */
                  /* при этом следующий уже стал текущим! */
    } else { /* в противном случае переходим к следующему */
        pcur = &(*pcur)->next;
    }
}
```

Попробуем теперь оформить наше решение в виде отдельной функции. Поскольку указатель `first` может потребоваться изменить, нам придётся передавать в функцию не его значение, а его *адрес*, но это в целом не проблема, нам так даже проще: мы просто назовём этот параметр `pcur` и используем в качестве переменной цикла. Выглядеть это будет так:

```
void delete_negatives_from_int_list(struct item **pcur)
{
    while(*pcur) {
        if((*pcur)->data < 0) {
            struct item *tmp = *pcur;
            *pcur = (*pcur)->next;
            free(tmp);
        } else {
            pcur = &(*pcur)->next;
```

```
        }
    }
}
```

Вызывать эту функцию придётся так:

```
delete_negatives_from_int_list(&first);
```

Как обычно, можно и для этой задачи применить рекурсию:

```
void delete_negatives_from_int_list(struct item **pcur)
{
    if(!*pcur)
        return;
    delete_negatives_from_int_list(&(*pcur)->next);
    if((*pcur)->data < 0) {
        struct item *tmp = *pcur;
        *pcur = (*pcur)->next;
        free(tmp);
    }
}
```

Рекурсивное решение станет изящнее, хотя и не короче (и даже длиннее), если несколько изменить подход к использованию параметра. Вместо того, чтобы передавать в функцию адрес указателя `first`, мы можем передать его значение, а *новое* значение вернуть из функции в качестве возвращаемого и присвоить обратно в `first`; решение тогда станет примерно таким:

```
struct item *delete_negatives_from_int_list(struct item *pcur)
{
    struct item *res = pcur;
    if(pcur) {
        pcur->next = delete_negatives_from_int_list(pcur->next);
        if(pcur->data < 0) {
            res = pcur->next;
            free(pcur);
        }
    }
    return res;
}
```

Вызов функции в этой версии выполняется иначе:

```
first = delete_negatives_from_int_list(first);
```

4.9.3. Двусвязные списки

В ***двусвязном списке*** каждое звено хранит адрес предыдущего звена и следующего звена, что позволяет, зная адрес любого из звеньев списка, добраться до всех остальных звеньев. Обычно при работе с двусвязным списком используют два указателя — на первое звено и на последнее; в таком списке можно легко вносить новые элементы и удалять существующие в начале списка, в его конце и в произвольной позиции, причём никаких хитрых техник вроде применения указателя на указатель для этого не требуется.

Рассмотрим для примера двусвязный список чисел типа `double`, который можно составить из таких звеньев:

```
struct dbl_item {
    double data;
    struct dbl_item *prev, *next;
};
```

Для работы с этим списком опишем указатели на его начало, конец, текущий элемент, а также временный (вспомогательный) указатель:

```
    struct dbl_item *first = NULL, *last = NULL;
    struct dbl_item *current = NULL, *tmp;
```

Операции внесения элемента в начало и конец списка вынужденно обрабатывают специальную ситуацию — случай пустого списка. Допустим, у нас имеется переменная `x` типа `double` и число, которое в ней хранится, нужно внести в список. Внесение в начало выглядит так:

```
    tmp = malloc(sizeof(struct dbl_item));
    tmp->data = x;
    tmp->prev = NULL;
    tmp->next = first;
    if(first)
        first->prev = tmp;
    else
        last = tmp;
    first = tmp;
```

Внесение в конец выглядит практически так же с точностью до «зеркальной симметрии»:

```
    tmp = malloc(sizeof(struct dbl_item));
    tmp->data = x;
    tmp->prev = last;
    tmp->next = NULL;
    if(last)
        last->next = tmp;
```

```
else
    first = tmp;
last = tmp;
```

Между прочим, в обоих случаях оператор `if` можно заменить однострочным присваиванием. В первом случае это будет выглядеть так:

```
*(first ? &first->prev : &last) = tmp;
```

Во втором случае — так:

```
*(last ? &last->next : &first) = tmp;
```

Разгадать этот ребус предложим читателю самостоятельно. Разумеется, мы не предлагаем в действительности так писать программы; но если вы с подобным столкнётесь в чужом коде, этот опыт, по крайней мере, поможет быстрее справиться с шоком.

Изъятие первого элемента двусвязного списка выглядит так:

```
if(first) {  /* проверка обязательна! */
    tmp = first;
    first = first->next;
    if(first)
        first->prev = NULL;
    else            /* список опустел */
        last = NULL;
    free(tmp);
}
```

Аналогичным образом выполняется изъятие последнего элемента:

```
if(last) {
    tmp = last;
    last = first->prev;
    if(last)
        last->next = NULL;
    else
        first = NULL;
    free(tmp);
}
```

Заметим, что если в любом из этих двух фрагментов заменить `if` на `while`, мы получим код, удаляющий все элементы списка; впрочем, сделать это можно и проще:

```
if(first) {
    first = first->next;
    while(first) {
        free(first->last);
        first = first->next;
```

```
        }
        free(last);
        last = NULL;
    }
```

Поясним, что здесь мы, прежде чем что-то делать, сдвигаем указатель `first` на следующую позицию, и если он после этого всё ещё указывает на очередное звено списка (то есть не стал нулевым), мы уничтожаем предыдущее звено, пользуясь указателем на него из нового «первого» звена. Последний оставшийся элемент мы так уничтожить не можем, потому что нет звена, указатель `prev` которого указывал бы на него; сдвинувшись на следующую позицию с последнего звена, мы получим указатель `first`, равный нулевому адресу; но это не страшно, ведь у нас есть ещё и `last`, через который мы благополучно ликвидируем последнее оставшееся звено списка. Примечательно, что здесь мы обошлись без вспомогательного указателя.

Рассмотрим теперь операции по модификации списка в произвольном месте с использованием указателя на *текущий элемент*. Работа с текущим элементом, естественно, предполагает, что он есть; в дальнейшем мы это не проверяем, предполагая, что проверка уже сделана. Пусть, как и раньше, значение, которое нужно вставить, находится в переменной `x`. Для начала выполним вставку *перед текущим элементом*:

```
tmp = malloc(sizeof(struct dbl_item));
tmp->data = x;                /* заполняем новый элемент */
tmp->next = current;
tmp->prev = current->prev;
current->prev = tmp; /* он будет предыдущим для текущего */
if(tmp->prev)                /* если есть предыдущий, то */
    tmp->prev->next = tmp;   /* новый для него следующий */
else               /* в противном случае новый элемент - */
    first = tmp;                          /* теперь первый */
```

Аналогично выполняется вставка *после* текущего элемента:

```
tmp = malloc(sizeof(struct dbl_item));
tmp->data = x;
tmp->prev = current;
tmp->next = current->next;
current->next = tmp;
if(tmp->next)
    tmp->next->prev = tmp;
else
    last = tmp;
```

Рассмотрим теперь *удаление текущего элемента*. Для начала мы исключим его из списка, модифицировав указатели в предыдущем и следующем элементах, если они есть, либо указатели `first` и `last`:

```
if(current->prev)
    current->prev->next = current->next;
else
    first = current->next;
if(current->next)
    current->next->prev = current->prev;
else
    last = current->prev;
```

Теперь наш текущий элемент находится вне списка, несмотря на то, что его указатели `prev` и `next` всё ещё указывают на соседние элементы или равны `NULL`, если соответствующих элементов не было (то есть текущий элемент располагался в начале и/или конце списка). Если мы согласны потерять место в списке, где находится сейчас текущий элемент, то операция завершается совсем просто:

```
free(current);
current = NULL;
```

В зависимости от решаемой задачи мы можем захотеть, чтобы новым текущим элементом стал левый или правый сосед удаляемого элемента (обычно это бывает нужно, если мы идём вдоль списка из конца в начало или из начала в конец соответственно). В этом случае нам придётся воспользоваться вспомогательным указателем. Если мы шли слева направо и новым текущим элементом хотим объявить тот элемент, что был справа от удаляемого, вместо двух «простых» строчек, приведённых выше, пишем так:

```
tmp = current;
current = current->next;
free(tmp);
```

Если мы шли справа налево и новым текущим должен стать тот, что был слева, просто меняем `next` на `prev`.

4.9.4. Простое бинарное дерево поиска

Бинарное (то есть двоичное, по числу возможных потомков у каждого узла) дерево поиска состоит из узлов, в каждом из которых присутствуют указатели на левое и правое поддеревья, а также «полезная информация», для хранения которой, собственно говоря, и создаётся дерево. Например, дерево для хранения целых чисел можно было бы составить из таких узлов:

```
struct node {
    int val;
    struct node *left, *right;
};
```

Для работы с такими деревьями в большинстве случаев применяют рекурсию. Чтобы понять, почему это так, приведём простенький пример. Допустим, у нас есть дерево из узлов типа `struct node` и нам нужно напечатать (с помощью `printf`) все числа, хранящиеся в дереве. Рекурсивное решение будет тривиальным; если дерево пустое, возвращаем управление (это базис рекурсии), в противном случае обращаемся сами к себе, чтобы напечатать левое поддерево, затем печатаем число в «корне» (то есть текущем узле) дерева и снова обращаемся сами к себе, но на этот раз печатаем правое поддерево:

```
void int_bin_tree_print_rec(struct node *r) {
    if(!r)
        return;
    int_bin_tree_print_rec(r->left);
    printf("%d ", r->val);
    int_bin_tree_print_rec(r->right);
}
```

Попытаемся теперь сделать то же самое без использования рекурсии. Для начала заметим, что нам, рассматривая любой из узлов дерева, необходимо помнить, какие узлы расположены выше, чтобы иметь возможность вернуться на верхние уровни дерева. Глубина дерева заранее не известна, так что придётся хранить соответствующую информацию то ли в массиве, то ли в списке. Кроме того, в один и тот же узел мы попадаем (то есть вынуждены его рассматривать) в общем случае трижды: «сверху» (то есть когда этот узел оказался корневым для очередного рассматриваемого поддерева), при возврате из рассмотрения левого поддерева и при возврате из рассмотрения правого поддерева. Чтобы знать, что ещё осталось сделать в данном конкретном узле дерева и куда следует двигаться после, нам придётся для каждого узла, работу с которым мы начали, но ещё не закончили, хранить *состояние*; мы обозначим вариант «ещё ничего не сделано» словом `start`, ситуацию «мы уже рассмотрели левое поддерево» словом `left_visited` и, наконец, состояние «всё сделано, пора на выход» словом `completed`. Коль скоро для каждого узла приходится хранить больше одного значения (по меньшей мере указатель на сам узел плюс состояние дел с этим узлом), это естественно приводит нас к идее использовать структуры, ну а *стековая* по сути природа этих данных — к идее использования обыкновенного односвязного списка. Последний элемент этого списка будет всегда соответствовать корню всего дерева, тогда как элемент

в начале списка — тому узлу, который рассматривается прямо сейчас. Полностью реализация обхода дерева будет выглядеть, например, так:

```
/* bintree.c */
void int_bin_tree_print_loop(struct node *r) {
    enum state { start, left_visited, completed };
    struct backpath {
        struct node *p;
        enum state st;
        struct backpath *next;
    };
    struct backpath *bp, *t;
    *bp = malloc(sizeof(*bp));
    bp->p = r;
    bp->st = start;
    bp->next = NULL;
    while(bp) {
        switch(bp->st) {
        case start:
            bp->st = left_visited;
            if(bp->p->left) {
                t = malloc(sizeof(*t));
                t->p = bp->p->left;
                t->st = start;
                t->next = bp;
                bp = t;
                continue;
            }
            /* no break here */
        case left_visited:
            printf("%d ", bp->p->val);
            bp->st = completed;
            if(bp->p->right) {
                t = malloc(sizeof(*t));
                t->p = bp->p->right;
                t->st = start;
                t->next = bp;
                bp = t;
                continue;
            }
            /* no break here */
        case completed:
            t = bp;
            bp = bp->next;
            free(t);
        }
    }
}
```

Даже на самый первый взгляд эта реализация много сложнее, чем приведённый выше вариант с использованием рекурсии. Впрочем, никакой магии здесь нет. В рекурсивном варианте рассмотрению каждого узла дерева соответствует очередной вызов функции, а значит — стековый фрейм; в нём имеется значение фактического параметра (переменная `r`), которая указывает на текущий узел. Кроме того, в тело функции мы попадаем трижды: при её вызове (попадаем в начало), после возврата из первого рекурсивного вызова (попадаем на ту строчку, где находится `printf`) и после возврата из второго рекурсивного вызова (попадаем в самый конец, в результате выходим из функции). Это в точности соответствует нашим трём *состояниям*, то есть получается, что при рекурсивном обходе дерева состояние тоже хранится (неявно) — в стековых фреймах в виде *адреса возврата*. Больше того, наш стек, реализованный в виде односвязного списка из структур `struct backpath` — есть не что иное, как модель аппаратного стека, используемого в рекурсивном решении.

Такое «принудительное развёртывание» рекурсии находит применение, когда дерево нужно обойти не за один приём, а пошагово: например, в дереве могут храниться некие значения, которые наш модуль должен отдавать по одному по мере того, как их у нас запрашивают. Однако пока такая ситуация не возникла, следует, разумеется, использовать рекурсию: как можно легко догадаться, циклическое решение здесь оказывается не только сложнее по трудозатратам и приложению интеллекта, оно вдобавок использует больше памяти и работает заметно медленнее.

Двоичное дерево обычно используют для ускорения поиска нужного элемента. Для этого элементы, ме́ньшие данного, располагают в левом поддереве, а элементы, бо́льшие данного — в правом. Добавление нового элемента в дерево с сохранением его упорядоченности можно реализовать, например, так:

```
void int_bin_tree_add(struct node **root, int n)
{
    if(!*root) {
        *root = malloc(sizeof(**root));
        (*root)->val = n;
        (*root)->left = NULL;
        (*root)->right = NULL;
        return;
    }
    if((*root)->val == n)
        return;
    if(n < (*root)->val)
        int_bin_tree_add(&(*root)->left, n);
    else
        int_bin_tree_add(&(*root)->right, n);
```

```
}
```

Поясним, что в функцию мы передаём *адрес* указателя на корневой узел поддерева; сам этот указатель может быть нулевым, что означает, что поддерево пусто, и ко всему дереву целиком это тоже относится: пока соответствующий указатель содержит нулевой адрес, дерево считается пустым. При первоначальном вызове функции она получает адрес того указателя, который используется в вызывающей программе для работы с деревом; обращаясь рекурсивно к себе самой, она передаёт параметром адрес одного из указателей `left` или `right` из текущего узла дерева.

В самой функции прежде всего (в качестве базиса рекурсии) рассматривается как раз случай пустого дерева/поддерева: в этом и только в этом случае создаётся новый элемент дерева, его поле `val` заполняется добавляемым в дерево числом, а указатели на поддеревья получают нулевые значения, что соответствует пустоте обоих поддеревьев. На этом задача считается решённой и управление возвращается вызвавшему.

Если дерево оказалось непустым, прежде всего производится проверка, не *равно* ли новое число текущему. В этом случае не делается ничего, поскольку такое число уже внесено в дерево; управление немедленно возвращается. В зависимости от решаемой задачи здесь можно было бы как-то оповестить главную программу о том, что на самом деле в дерево ничего не внесено — например, вернуть «ложь», а в других случаях возвращать «истину», и т. д. Можно было бы поступить и хитрее, например, в каждом узле дерева хранить не только само число, но и счётчик, показывающий, *сколько* раз это число было добавлено в дерево.

Наконец, если текущее поддерево не пусто, но при этом добавляемое число не равно текущему, оно может быть либо меньше, либо больше текущего; в этих случаях его следует попытаться добавить соответственно в левое или в правое поддерево. Это делается рекурсивным вызовом, при этом в качестве адреса указателя на корень поддерева используется адрес указателя `left` или `right` из текущего элемента. Дальнейшее — забота рекурсивного вызова; например, если поддерево оказалось пустым, то узел, содержащий добавляемое число, станет непосредственным потомком текущего узла, и так далее.

Как известно, с двоичными деревьями поиска связана одна очень серьёзная проблема: их топология оказывается зависима от порядка, в котором значения заносились в дерево. Так, если в пустое дерево начать заносить последовательность целых чисел, уже упорядоченных по возрастанию или по убыванию, новые числа всегда будут заноситься в самую правую (или самую левую) позицию дерева, а само дерево по сути превратится в односвязный список, официально именуемый вырожденным деревом. Ясно, что поиск в таком дереве будет происходить очень медленно. Для решения этой проблемы применяются так

называемые *сбалансированные деревья*. Рассказ о них выходит за рамки нашей книги, тем более что способы их построения и известные алгоритмы балансировки подробно расписаны в литературе и множестве источников в Интернете. Отметим только, что подходов к поддержанию дерева в сбалансированном состоянии существует довольно много, и среди них нет ни одного *простого*.

Как ни странно, во многих задачах удаётся обойтись без балансировки — если входные данные в силу своей природы оказываются хорошо распределены, дерево изначально получается достаточно «развесистым». Если же это не ваш случай, то часто оказывается проще применить какой-нибудь другой вид дерева, нежели строить двоичное дерево и пытаться его сбалансировать. Например, для хранения строк часто применяются ***префиксные деревья*** (английский термин — *trie*), в которых у каждого узла может быть столько потомков, сколько существует разных букв/символов; продвигаясь от корня к листьям, мы одновременно сдвигаемся вдоль строки, рассматривая каждый раз следующий её символ, то есть на n-ном уровне дерева мы принимаем решение, в какое поддерево пойти, используя n-ный символ из строки. «Высота» такого дерева в точности равна длине самой длинной из хранимых в нём строк, а поиск нужной строки занимает в худшем случае столько сравнений, сколько в строке есть символов.

4.9.5. Объединения и вариантные структуры

Под *объединением* (`union`) в Си понимается такой тип данных, переменная которого может хранить значение одного из нескольких типов, но только одно в каждый момент[54]. Синтаксически объединения очень похожи на структуры: они описываются точно так же и по тем же правилам, только вместо слова `struct` используется слово `union`; у объединений точно так же есть *поля*, к которым можно обращаться через точку, а если используется *адрес* объединения — то через «стрелочку». Отличие от структур, говоря технически, в том, что все поля объединения расположены в памяти начиная с одного и того же адреса — с того адреса, с которого начинается само объединение. Ясно, что присваивание значения любому из полей объединения попросту *затирает* остальные его поля.

Рассмотрим для примера объединение

```
union sample_un {
    int i;
    double k;
```

[54]Название «объединение» оправдывается соображениями из теории множеств. В самом деле, если рассматривать *тип данных* как *множество возможных значений*, то `union` действительно оказывается теоретико-множественным *объединением* типов как множеств.

```
    char str[16];
};
```

Если теперь описать переменную этого типа:

```
union sample_un su;
```

то в нашем распоряжении окажутся: переменная целого типа `su.i`; переменная типа `double su.k`; массив `char`'ов, доступный по адресу `su.str`. Все они расположены в памяти в одном и том же месте, иначе говоря, численное значение адресов `&su`, `&su.i`, `&su.k` и `su.str` одно и то же. Общий объём такого объединения, скорее всего, составит 16 байт, поскольку элемент `str` у него самый большой (хотя теоретически возможны архитектуры, на которых тип `double` окажется занимающим больше 16 байт). В общем случае размер объединения равен наибольшему из размеров его полей.

Изначально объединения были предназначены для экономии памяти, но в такой роли их давно уже никто не использует, поскольку память нынче дёшева, чего никак нельзя сказать о времени, затрачиваемом программистами на создание и отладку программ; между тем, экономия памяти с помощью объединений чревата неочевидными ошибками: можно очень легко перепутать, какое из полей мы присваивали последним, а ведь все остальные при этом содержат откровенную абракадабру.

Одно из частных применений объединения состоит в расчленении того или иного значения на составляющие его байты. Так, объединение

```
union split_int {
    int integer;
    unsigned char bytes[sizeof(int)];
};
```

позволит узнать, из каких байтов состоит представление произвольного целого числа, например:

```
int i;
union split_int si;
printf("Please enter an integer number: ");
scanf("%d", &si.integer);
for(i = 0; i < sizeof(int); i++)
    printf("byte #%d is %02x\n", i, si.bytes[i]);
```

Гораздо более интересные возможности открываются, если объединение используется в качестве поля структуры. Результат получается тот же, как при использовании вариантных записей в Паскале; как правило, поля объемлющей структуры содержат в том или ином виде информацию о том, какое из полей объединения следует использовать.

К примеру, если бы мы писали какой-нибудь «продвинутый» калькулятор или другую программу, работающую с формулами, нам, скорее всего, потребовался бы список элементов формулы, в качестве которого могут выступать целочисленные и «плавающие» константы, имена переменных, а также символы действий и скобок. Если, скажем, мы согласимся ограничить длину имени переменной семью символами, чтобы соответствующая строка не занимала больше места, чем число типа `double`, то для представления таких элементов и их хранения в виде списка мы могли бы использовать следующую структуру:

```
struct expression_item {
    char c;
    union un_data {
        int i;
        double d;
        char var[sizeof(double)];
    } data;
    struct expression_item *next;
};
```

При этом можно, например, условиться, что если значение `c` превышает число 32 (ASCII-код пробела), то в поле `c` хранится код символа операции или другого «формульного» символа (например, скобки), при этом поля объединения `data` не используются; в противном случае значение `c` (например, 0, 1 и 2) определяет, какой тип имеет наш элемент формулы и, соответственно, какое из полей объединения следует использовать. Для значений `c` следует, по-видимому, описать перечислимый тип:

```
enum expr_item_types
    { eit_int = 0, eit_dbl = 1, eit_var = 2, eit_min_op = ' ' };
```

Теперь если поле `c` равно значению `eit_int`, то это целочисленная константа и используется `data.i`, если `eit_dbl` — это дробная константа и используется `data.d`, если `eit_var` — это имя переменной и используется `data.var`; наконец, если значение `c` больше либо равно `eit_min_op`, то `data` не используется вообще, а элемент списка представляет собой символ операции или скобку.

Практически все компиляторы ещё в середине девяностых поддерживали так называемые *анонимные объединения*, хотя в стандарт языка эта возможность вошла только начиная с C11. Состоит этот механизм в том, что внутри структуры описывается объединение, причём ему ни в каком виде не даётся имени: не указывается ни его тег, ни имя поля для него. Тогда поля такого объединения становятся полями самой структуры, но располагаются, как и положено полям объединения, «одно на другом». В частности, с использованием этой возможности вышеприведённую структуру можно было бы описать так:

```
struct expression_item {
```

```
    char c;
    union {
        int i;
        double d;
        char var[sizeof(double)];
    };
    struct expression_item *next;
};
```

Теперь для структуры типа `struct expression_item` доступны поля `c`, `i`, `d`, `var` и `next`, при этом из трёх полей `i`, `d` и `var` можно использовать только какое-то одно, поскольку они занимают одну и ту же память.

4.9.6. Битовые поля

Для целочисленных полей структуры язык Си позволяет в явном виде указать количество битов, которые данное поле будет занимать. Как правило, такие поля описывают как беззнаковые; количество битов указывается сразу после имени поля через двоеточие. Например,

```
struct myflags {
    unsigned io_error:1;
    unsigned seen_a_digit:1;
    unsigned had_a_eol:1;
    unsigned signaled:1;
    unsigned count:4;
};
```

состоит из четырёх именованных *флагов*, каждый из которых, будучи однобитовым, может быть равен либо нулю, либо единице, и некоего «счётчика», на который отведено четыре бита, то есть он может принимать значение от 0 до 16. Важно понимать, что подобное имеет смысл только если несколько битовых полей расположены в структуре подряд друг за другом, поскольку любое поле, не являющееся битовым, естественно, будет начинаться с границы ячейки памяти, возможно, даже не ближайшей (напомним, что компилятор на большинстве архитектур выравнивает многобайтовые поля в структурах на чётные адреса, а в некоторых случаях — и на адреса, кратные четырём).

Больше того, даже идущие подряд битовые поля компилятор «упакует» (физически) в неявное число того типа, какой использован при описании полей. Так, структура из вышеприведённого примера использует тип `unsigned` (он же `unsigned int`), который на современных архитектурах занимает четыре байта; именно столько будет занимать и вся эта структура целиком, несмотря на то, что все её поля вполне умещаются в один байт. Если хочется всерьёз сэкономить память, вместо обычного `unsigned` следует использовать `unsigned char`; структура

```
struct myflags_c {
    unsigned char io_error:1;
    unsigned char seen_a_digit:1;
```

```
    unsigned char had_a_eol:1;
    unsigned char signaled:1;
    unsigned char count:4;
};
```

займёт в памяти один байт, то есть `sizeof(struct myflags_c)` окажется равно 1.

4.9.7. Директива `typedef`

Язык Си позволяет ввести *пользовательское имя типа* — идентификатор, обозначающий тот или иной тип значений и переменных. Для этого используется директива `typedef`; синтаксически описание имени типа получается из *описания одной переменной произвольного типа*: добавление слова `typedef` превращает его в описание имени этого типа вместо переменной. Например, описание

```
typedef int *intptr;
```

вводит имя типа `intptr`. С помощью `typedef` можно задать синоним для уже существующего типа, в том числе встроенного:

```
typedef int integral_number;
```

Такие синонимы широко используются библиотеками для введения типов, описание которых зависит от архитектуры. Например, тип `intptr_t`, обозначающий целое, разрядности которого хватает для хранения адресов на данной архитектуре, на 32-битных системах обычно вводится как синоним типа `int`, а на 64-битных — как синоним типа `long`.

Очень часто программисты, не желая каждый раз писать имя структурного типа из двух слов, вводят для него синоним с помощью `typedef`, например:

```
struct tag_mystruct {
    int i;
    double d;
};
typedef struct tag_mystruct mystruct;
```

или просто

```
typedef struct tag_mystruct {
    int i;
    double d;
} mystruct;
```

Это позволяет описывать переменные такого типа без использования слова `struct`:

```
mystruct s1, s2, s3;
mystruct *ptr;
```

Начинающие часто оказываются недовольны тем, что с помощью такого имени типа нельзя описать поле-указатель на структуру того же типа, как это требуется при построении списков, деревьев и прочих подобных структур данных. В самом деле, имя, введённое с помощью `typedef`, начинает действовать только *после* директивы; приходится поэтому по-прежнему использовать имя структуры, например

```
typedef struct tag_item {
   int data;
   struct tag_item *next;
} item;
```

хотя во всём дальнейшем тексте можно уже использовать введённое имя типа, например.

```
item *first = NULL;
```

Иногда в программах на Си можно встретить описания структур с `typedef`, в которых имя структуры и имя вводимого типа совпадают. Компилятор позволяет это, поскольку имена типов и имена структур относятся в Си к разным областям видимости и не конфликтуют. Например, следующее описание компилятор вполне нормально «скушает»:

```
typedef struct item {
   int data;
   struct item *next;
} item;
```

Несмотря на кажущуюся привлекательность такого варианта, **делать так не рекомендуется**, во всяком случае, если вы выносите всё это в заголовочный файл. Дело в том, что в языке Си++, в отличие от Си, понятие «идентификатора структуры» отсутствует, там имя структуры сразу же считается именем типа, и описание, подобное этому, вызовет конфликт имён, то есть ваш модуль нельзя будет использовать в проектах на Си++, только на чистом Си.

4.10. Макропроцессор

4.10.1. Предварительные сведения

С концепцией макропроцессора мы уже знакомы по языку ассемблера; в языке Си он тоже присутствует и, более того, мы с самого

начала используем его во всех наших примерах программ, хотя и не обращаем на это особого внимания. Дело в том, что хорошо знакомая нам директива `#include` как раз является частью макропроцессора. Общая идея макропроцессирования, напомним, состоит в том, что текст (в данном случае текст программы) в какой-то момент подаётся на вход специальной программе или подсистеме компилятора, которая называется ***макропроцессором***. Текст, подаваемый на вход макропроцессору, как правило, содержит определённые указания по его преобразованию, выраженные в виде ***макродирективов*** и ***макровызовов***[55]; результатом работы макропроцессора становится опять-таки текст, но уже не содержащий макродирективов и не требующий дальнейшего преобразования.

Все макродирективы препроцессора языка Си имеют общую особенность: они записываются на отдельной строке, и первым значащим (непробельным) символом такой строчки должен быть символ «#». Обычно перед макродирективой пробелов не оставляют, то есть символ «#» располагают в первой позиции. Макродирективы представляют собой прямой приказ макропроцессору; по окончании его работы никаких следов от макродирективов в анализируемом тексте не остаётся.

Кроме макродирективов, препроцессор обрабатывает также ***макроимена***, то есть идентификаторы, связанные с макросами, введёнными пользователем.

Макропроцессор языка Си обладает одним довольно неприятным свойством: он работает *до начала синтаксического анализа*, хотя и после анализа лексического. При компиляции текст программы на Си сначала разбивается на так называемые ***лексемы***: ключевые слова, идентификаторы, знаки арифметических операций и другие знаки препинания, такие как, например, фигурные скобки, запятые, точки с запятой и двоеточия, а также числовые, символьные и строковые литералы. Например, фрагмент

```
while(count < 10)
    printf("[%d] Hello\n", count++);
```

превратится в результате анализа в следующую последовательность лексем: «`while`», «`(`», «`count`», «`<`», «`10`», «`)`», «`printf`», «`(`», «`"[%d] Hello\n"`», «`,`», «`count`», «`++`», «`)`», «`;`». Комментарии и пробельные символы после лексического анализа исчезают без следа. На всех последующих этапах трансляции вместо изначального текста программы, состоящего из отдельных символов, рассматривается последовательность таких вот лексем, при этом каждая лексема считается единым неделимым целым. Именно на этом этапе, то есть когда текст

[55] В первом приближении можно считать, что макродирективы встроены в макропроцессор, тогда как макровызовы основаны на ***макросах***, определяемых пользователем.

уже разбит на лексемы, но ещё не проанализирован, переменным не приписаны типы, не выстроено синтаксическое дерево операторов и выражений, не определены области видимости — как раз и запускается макропроцессор.

В первом приближении это означает, что имена, введённые макропроцессором, во-первых, представляют собой отдельные лексемы-идентификаторы, то есть макровызовы не могут выполняться, например, внутри комментариев (собственно, их к этому времени уже нет), идентификаторов (что довольно очевидно) и строковых констант; во-вторых, **макроимена не подчиняются областям видимости**, поскольку во время работы макропроцессора текст программы на области видимости ещё не разбит.

4.10.2. Макроопределения и макровызовы

Макроопределение представляет собой фрагмент исходного текста программы, в котором вводится новое имя (идентификатор), предназначенное к обработке макропроцессором. В языке Си макроопределения делаются с помощью директивы `#define`; в простейшем случае эта директива содержит идентификатор, а следом за ним и до конца строки идёт текст, на который этот идентификатор должен быть заменён. Вот несколько примеров:

```
#define BUFFERSIZE 1024
#define HELLOMSG "Hello, world\n"
#define MALLOCITEM malloc(sizeof(struct item))
```

Здесь `BUFFERSIZE`, `HELLOMSG` и `MALLOCITEM` — ***макроимена*** или просто макросы; встретив любой из этих идентификаторов в дальнейшем тексте программы, компилятор заменит первый из них на число `1024`, второй — на строковую константу `"Hello, world\n"`, третий — на вызов функции `malloc`. Можно ожидать теперь появления в программе чего-то вроде следующего:

```
char buffer[BUFFERSIZE];
  /* ... */
puts(HELLOMSG);
  /* ... */
struct item *p = MALLOCITEM;
```

и так далее. Это, собственно говоря, и есть ***макровызовы***.

Тело макроса не обязано быть законченным выражением или даже законченной конструкцией; вполне можно описать, например, такое:

```
#define IF if(
#define THEN ) {
```

```
#define ELSE } else {
#define FI }
```

и затем в программе изобразить что-то вроде

```
IF a > b THEN
    printf("the first was greater then the second\n");
    b = a;
ELSE
    printf("the second was greater\n");
    a = b;
FI
```

Иной вопрос, что конкретно вот так делать всё же не стоит (хотя и можно), но в более сложных случаях подобный синтез языковых конструкций из макросов вполне может себя оправдать.

4.10.3. Соглашения об именовании

Читатель наверняка обратил внимание на то, что все имена макросов в наших примерах набраны заглавными буквами, тогда как до сих пор мы никогда заглавные буквы в идентификаторах не использовали. Безусловно, это не случайно. Как уже говорилось, **макроимена не подчиняются правилам видимости**. Что это значит на практике, нам поможет понять следующий пример. Допустим, где-то в одном из заголовочных файлов нашего проекта кто-то из участников определил макрос с именем `count`:

```
#define count 35
```

Мы, не зная об этом или попросту забыв, решаем в какой-нибудь функции объявить локальную переменную с таким же именем:

```
int f(int n, const char *str)
{
    int count;
    /* ... */
```

Поскольку макропроцессор отрабатывает до начала синтаксического анализа, он не может ничего знать ни о функциях, ни об их границах, ни о локализации переменных; как следствие, он просто заменит слово `count` на число `35`, в результате чего синтаксический анализатор «увидит» замечательную конструкцию

```
int 35;
```

и, разумеется, выдаст сообщение об ошибке, в котором ключевыми словами будет что-то вроде *identifier expected*, что буквально переводится как «ожидается идентификатор», а означает это, если не делать подстрочного перевода, «здесь должен стоять идентификатор». При этом позиция произошедшей ошибки будет как раз там, где стоит идентификатор — слово `count`. Возможна ещё более пикантная ситуация: мы уже давно написали функцию с локальной переменной `count`, после чего кто-то из наших коллег в одном из заголовочников решил ввести это слово в качестве макроса, и весь наш код разом перестал компилироваться.

Именно по этой причине программисты, работающие на Си, придерживаются традиции использовать для макросов имена, состоящие из букв верхнего регистра. Во-первых, такие имена хорошо видно в тексте программы, они как будто говорят: «Осторожно, тут может возникнуть какая-нибудь нежданная проблема» (и, что характерно, говорят весьма по делу). Во-вторых, если все следуют этому соглашению, то макросы и обычные переменные никогда не смогут вступить в конфликт и создать ситуацию, подобную вышеприведённой: в самом деле, если все макроимена набраны большими буквами, а все прочие идентификаторы — маленькими, то ни одно макроимя не совпадёт ни с одним из «прочих идентификаторов» — имён переменных, типов и так далее.

Подчеркнём, что с точки зрения компилятора Си именем макроса может быть *любой идентификатор*; соглашение насчёт заглавных букв — это традиция, введённая программистами, а не требование языка.

С этой традицией связан достаточно забавный казус. Как мы уже упоминали при обсуждении перечислимого типа (см. §4.8.2), до появления `enum`'ов макросы были в языке Си единственным способом введения именованных констант; затем, когда перечислимый тип в языке появился, программисты довольно быстро сообразили, что его (а точнее, вводимые им идентификаторы-значения) можно использовать для именования целочисленных констант; но к этому времени привычка именовать константы заглавными буквами пустила настолько глубокие корни, что об исходной причине такого именования (в качестве которой исходно выступала опасность макросов, связанная с их неподчинением областям видимости) люди уже успели позабыть и принялись именовать так *любые* константы, в том числе и введённые с помощью `enum`. Закреплению этого нонсенса в немалой степени способствовало то, что Керниган и Ритчи в своей книге использовали заглавные буквы для именования макросов, *никак этого не поясняя*, и вдобавок стали использовать заглавные буквы для именования значений в перечислимых типах, опять-таки не давая на эту тему никаких пояснений.

Итогом всей этой ахинеи стало удивительно сильное распространение среди программистов убеждённости в том, что-де *заглавными буквами следует именовать константы*, при этом никто из разделяющих этот принцип не может дать внятного объяснения, чем же столь «интересны» именно константы, а не, например, имена типов или функций. Оно и понятно, ведь изначально под «константами» в Си можно было понимать только макросы, а с ними, как мы уже

видели, всё настолько плохо, что их действительно весьма желательно делать «заметными» в тексте:

```
#define VERY_DANGEROUS_MACRO 756
```

Но вот значения перечислимых типов — это, на первый взгляд, обыкновенные идентификаторы, и в них нет ровным счётом ничего особенного. Отметим, что и *на второй взгляд* они тоже являются обычными идентификаторами, так что если вы используете `enum`'ы по назначению — для обозначения одной ситуации из предопределённого множества — то идентификаторы можно (и, пожалуй, нужно) писать, как обычно, в нижнем регистре с использованием подчёркивания. Здравый смысл подсказывает, что при использовании `enum`'а не по назначению, то есть для введения именованных констант, следует поступать точно так же:

```
enum { absolutely_safe_constant = 756 };
```

Идентификатор, подчиняющийся областям видимости, в любом случае лучше, нежели такой идентификатор, который ведёт себя как слон в посудной лавке (а именно таковы макросы в Си), поэтому метод введения констант как идентификаторов для перечислимых типов получил широкое распространение. Здесь всё ещё как будто нет повода для беспокойства, но он появится, как только мы сделаем следующий — вполне логичный — шаг: решим переделать таким способом все уже имеющиеся константы, которые были ранее объявлены с помощью `#define` и в соответствии с вышеописанными соглашениями названы именами, состоящими из заглавных букв. Возможно, *некоторые* из них мы даже переименуем. Но рано или поздно мы столкнёмся со случаем, когда переименование константы недопустимо: например, константа является частью интерфейса популярной библиотеки, включена в документацию и используется в десятке (а то и в десятке тысяч) программ. Итак, мы обнаружили константу, которую нельзя переименовать; что же теперь, так и оставить её в виде макроса? Иногда, что характерно, именно так и поступают, в современных программах на Си константы, введённые `#define`'ом, всё ещё встречаются чаще, нежели `enum`'овые. Но, если подумать, валидных причин для такого решения попросту нет: в самом деле, ну *чем хуже* станет код, если очередной макрос заменить на безопасную константу с тем же именем? Результатом становится массовое использование в качестве имён констант перечислимого типа идентификаторов, более привычных в роли макроимён, то есть состоящих из заглавных латинских букв:

```
enum { VERY_DANGEROUS_MACRO = 756 };
```

Здесь наша константа уже и не dangerous, и не macro, но если переименовать её мы не смогли, то что же теперь поделать?

К сожалению, при этом несколько нарушается единообразие имён идентификаторов: одни константы именуются в нижнем регистре, а другие в верхнем. Поэтому требование именовать с использованием букв верхнего регистра *любые* константы, в том числе и такие, которые никогда не были макросами:

```
enum { ABSOLUTELY_SAFE_CONSTANT = 756 };
```

не получается объявить заведомо неправильным: как мы видели, *некоторые* константы, пусть и не являющиеся макросами, всё же приходится так именовать, а единообразие имён ценно само по себе.

Как это часто бывает, однозначных указаний здесь дать нельзя. Если вас пригласили в существующий проект, следуйте имеющимся соглашениям; если вы начинаете проект с нуля, подумайте сами, как вам будет удобнее именовать константы, но не забывайте, откуда *на самом деле* взялось именование заглавными буквами. Если бы не макросы, не было бы никаких оснований «выделять» в тексте программы именно константы, а не что-то другое.

4.10.4. Более сложные возможности макросов

Препроцессор языка Си позволяет задавать макросы *с параметрами*. Следующий пример представляет собой своеобразную «классику жанра»:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Здесь вводится макрос `MAX`, принимающий два *параметра*, которые в теле макроса обозначаются как `A` и `B`. Вызов этого макроса может выглядеть, например, так: `MAX(x, 15)`, при этом вместо макровызова будет подставлено тело макроса, в котором, в свою очередь, параметр `A` будет заменён на `x`, а параметр `B` — на `15`. **Перед открывающей круглой скобкой, с которой начинается список параметров, не должно быть пробела**, иначе ваш список параметров будет воспринят как часть тела макроса, а сам макрос станет обычной макроконстантой (без параметров).

Вы наверняка обратили внимание на неожиданно большое количество круглых скобок; интересно, что Керниган и Ритчи, приводя этот пример в своей книге, не тратят времени на объяснения, а вместо этого дают ещё один пример:

```
#define square(x) x * x
```

и предлагают вызвать его так: `square(z+1)`, надеясь, что этот ребус окажется «по зубам» всем без исключения читателям. Мы не будем столь оптимистичны и поясним, что произойдёт: вместо макровызова `square(z+1)` препроцессор сгенерирует последовательность лексем «`z + 1 * z + 1`», что, как можно заметить, на ожидавшийся квадрат числа совершенно не похоже. Аналогично, если бы мы определили наш макрос для вычисления максимума без применения скобок:

```
#define BADMAX(A, B)  A > B ? A : B
```

а потом использовали бы его, например, в составе выражения `BADMAX(100, 10) * 5`, то вместо ожидавшегося `500` мы бы получили результат `100`: в самом деле, чему ещё должно быть равно выражение `100 > 10 ? 100 : 10 * 5`, полученное после макроподстановки? Чтобы избежать подобных вывертов, программисты применяют простое

правило: **в параметризованном макросе каждое вхождение макропараметра следует обязательно заключить в круглые скобки, а всё тело макроса — ещё в одни**. Отметим, что это правило ни в коей мере не является требованием со стороны компилятора: ему всё равно; скорее это правило следует отнести к категории придуманных программистами способов преодоления последствий одного из многих кошмарных недочётов языка Си. Заметим, что круглые скобки, даже если их правильно расставить, не способны побороть все возможные последствия применения макросов; в частности, если выражение, заданное в роли одного из параметров нашего макроса `MAX`, окажется имеющим побочный эффект, то этот эффект может проявиться дважды; к примеру, в выражении `MAX(f(x),g(x))` та функция (`f` или `g`), результат которой окажется больше, будет вызвана второй раз.

Вывод оказывается достаточно очевиден: использования макропроцессора следует по возможности избегать. К сожалению, язык Си сравнительно беден, поэтому часто экономия времени и сил, достигаемая с помощью макросов, перевешивает возможные проблемы. Язык Си позволяет писать макроопределения из нескольких строк, и даже эта возможность, несмотря на все неудобства, довольно часто используется. Строки «склеиваются» с помощью символа «обратной косой черты» «\», который ставится в последней позиции строки (то есть непосредственно перед переводом строки), например, так:

```
#define HELP_TEXT \
    "This is a very good program that displays file size\n" \
    "To use it you should specify the file name\n" \
    "as a command line parameter, such as\n" \
    "   fsize file.txt\n"
```

Вспомнив, что несколько идущих подряд строковых констант компилятор «склеивает» в одну строку, мы можем догадаться, что здесь определена большая строковая константа, содержащая в себе четыре строки текста, разделённые символами перевода строки; её можно, например, напечатать в случае, если пользователь забыл дать нашей программе нужный ей параметр командной строки:

```
if(argc < 2) {
    fputs(HELP_TEXT, stderr);
    return 1;
}
```

Возможность описания многострочных макросов часто используют для описания макросов с параметрами, которые потом разворачиваются в целые функции. Пусть, например, нам потребовалась возможность суммирования элементов заданного массива. Если тип элементов не вызывает сомнения, то всё довольно просто; в частности, массив из `int`'ов можно просуммировать так:

```
int int_array_sum(const int *a, int n) {
    int s = 0;
    while(n > 0) {
        s += *a;
        a++;
        n--;
    }
    return s;
}
```

Если теперь нам потребуется сумма элементов массива какого-то другого типа, в голову может прийти идея скопировать эту функцию и заменить в копии слово `int` на имя нужного типа, например, `double`:

```
double double_array_sum(const double *a, int n) {
    double s = 0;
          /* ... */
```

однако от реализации таких идей следует воздерживаться: копирование фрагментов кода, как известно, до добра не доводит. С помощью механизма параметризованных многострочных макросов можно поступить чуть-чуть лучше:

```
#define MAKE_ARRAY_SUM_FUNCTION(FUNNAME, TYPE) \
    TYPE FUNNAME(const TYPE *a, int n) { \
        TYPE s = 0; \
        while(n > 0) { \
            s += *a; \
            a++; \
            n--; \
        } \
        return s; \
    }
```

Теперь ввести ещё одну такую функцию для суммирования массивов очередного понадобившегося нам типа можно без всякого копирования, достаточно сделать соответствующий макровызов:

```
MAKE_ARRAY_SUM_FUNCTION(int_array_sum, int)
MAKE_ARRAY_SUM_FUNCTION(double_array_sum, double)
MAKE_ARRAY_SUM_FUNCTION(long_array_sum, long)
```

и так далее. Можно пойти ещё дальше и воспользоваться возможностью «склеивания токенов» через псевдооперацию «`##`»; поясним, что в ходе макроподстановки препроцессор «склеивает» в одну лексему две разные лексемы, между которыми обнаружилось это самое «`##`», при этом в роли склеиваемых лексем обычно выступают идентификаторы. Например, мы могли бы описать наш макрос с одним параметром, а не с двумя:

```
#define MAKE_ARRAY_SUM_FUNCTION(TYPE) \
    TYPE TYPE ## _array_sum(const TYPE *a, int n) { \
        TYPE s = 0; \
        /* ... */
```

При макровызове, например, с параметром `int` жутковатое `TYPE ## _array_sum` превратится в идентификатор `int_array_sum`, что вполне соответствует нашим целям. Правда, такое решение имеет довольно очевидный недостаток: существуют имена типов из более чем одного слова, и для таких случаев новая версия макроса не годится (как минимум, придётся предварительно ввести новое имя для такого типа с помощью директивы `typedef`, см. §4.9.7).

Неудобства работы с многострочными макросами начинаются с того, что символ обратной косой черты, «экранирующий» перевод строки, обязан быть именно *последним* символом строки; **если случайно оставить после него пробел, текст перестанет компилироваться**, при этом пробел, который стал причиной ошибки, **не будет видно**. Впрочем, к этой особенности компилятора довольно легко приноровиться; но вот если вы допустили в тексте макроса ошибку, результатом которой становится ошибка компиляции, то **компилятор выдаст ошибку в том месте, где макрос раскрывается, а не там, где он описан**, и выискивать причину ошибки в теле макроса вам придётся без помощи со стороны компилятора; фактически в вашем распоряжении останется только небезызвестный «метод пристального взгляда». К этой особенности многострочных макросов «приноровиться» невозможно, поскольку это вообще не вопрос сноровки.

Небольшую помощь в деле отладки макросов вам может оказать ключ компилятора `-E`, предписывающий прекратить компиляцию сразу после стадии макропроцессирования, а результат выдать в поток стандартного вывода; будьте готовы при этом увидеть не только вашу программу, но и всё, что получилось из всех подключённых к ней заголовочных файлов (а это обычно много).

Коль скоро мы упомянули о склеивании токенов через ##, расскажем заодно об ещё одной своеобразной возможности макропроцессора: о превращении токена в строковую константу. Достигается это с помощью одиночной «решётки», поставленной непосредственно перед именем макропараметра. Например:

```
#define VAR_PRINT(x) printf("%s = %d\n", #x, x)
```

Если теперь этот вызов использовать, скажем, так:

```
int abrakadabra = 13;
VAR_PRINT(abrakadabra);
```

то раскроется наш макрос в выражение

```
printf("%s = %d\n", "abrakadabra", abrakadabra);
```

ну а напечатано будет, как легко видеть, что-то вроде

```
abrakadabra = 13
```

Вспомнив, что встреченные в тексте два или более идущих подряд строковых литерала «склеиваются», мы можем того же эффекта достичь несколько более интересным способом:

```
#define VAR_PRINT(x) printf(#x " = %d\n", x)
```

4.10.5. (*) Макросы и конструкция `do { } while(0)`

При активном использовании макросов вы рано или поздно наткнётесь на ситуацию, когда в теле макроса по той или иной причине хочется использовать *несколько операторов* или другую сложную конструкцию вроде оператора `if` с веткой `else` или блока с локальными переменными. Бывает и так, что один и тот же макрос вы в зависимости от сложившейся ситуации определяете по-разному, то с одним действием внутри, то с двумя, то вообще как «пустышку», не делающую ничего. При этом хочется, чтобы макровызов в программе выглядел так же, как обычный оператор с вызовом функции, вроде «`MYMACRO("argument");`», с точкой с запятой на конце, и чтобы эта конструкция могла быть без опаски использована в любом контексте, где допустим *оператор вычисления выражения*, например, в качестве тела `if`.

Если не принять специальных мер и использовать один из следующих вариантов:

```
#define MYMACRO(arg) f(arg); g();
#define MYMACRO(arg) f(arg); g()
#define MYMACRO(arg) { f(arg); g(); }
```

то можно с ходу указать конструкцию, которая хоть и выглядит в тексте программы совершенно правильной, тем не менее не откомпилируется:

```
if(cond)
    MYMACRO("argument");
else
    /* ... */
```

Можно придумать и другие контексты, в которых все или некоторые из вышеприведённых определений «сломаются». Как это часто бывает с языком Си, программисты довольно быстро нашли для возникшей проблемы работающее решение, основанное на конструкции, исходно предназначенной совершенно не для этого; проблема решается путём заключения тела макроса в вырожденный цикл `do-while`:

```
#define MYMACRO(arg) do { f(arg); g(); } while(0)
```

Поскольку условие в `while` изначально установлено ложным, такой цикл, будучи *циклом с постусловием*, всегда выполняется ровно один раз; с другой стороны, синтаксис языка Си требует после него ставить точку с запятой, что полностью

соответствует нашему желанию относительно вызова MYMACRO с точкой с запятой на конце, подобно обыкновенной функции.

Подчеркнём, что подходящая конструкция нашлась в языке случайно; в частности, синтаксис мог бы и не требовать точки с запятой в конце конструкции do-while, она там явно избыточна. Программисты на Си вынуждены были стать большими мастерами на подобные выдумки.

4.10.6. Директивы условной компиляции

Препроцессор языка Си содержит средства, позволяющие временно исключать из компиляции фрагменты исходного кода, а также выбирать один из нескольких фрагментов в зависимости от заданных условий; это позволяет быстро создавать различные (например, отличающиеся набором возможностей, применёнными алгоритмами и т. д.) варианты одной и той же программы, причём организовать работу можно так, что для переключения с одного варианта на другой не потребуется вносить никаких изменений в исходные тексты, что может оказаться очень важно.

С условной компиляцией мы встречались уже дважды: в первом томе — при обсуждении отладочной печати (см. т. 1, стр. 440), а в части, посвящённой ассемблеру — при обсуждении макропроцессора ассемблера NASM (§3.5.4). Отметим, что и во Free Pascal, и в NASM макродирективы, связанные с условной компиляцией, пришли из языка Си, и это сейчас станет очевидно.

Самый простой пример применения условной компиляции — исключение фрагмента, который больше не нужен (например, вы уже написали другой вариант реализации той же задачи), но удалить его рука не поднимается. При работе на других языках программирования, в том числе на Паскале, для этого обычно используются знаки комментариев, соответствующее действие даже получило жаргонное название: «закомментарить» кусок текста. В программах на Си такой вариант не проходит, поскольку вложенные комментарии компилятором не поддерживаются; этот факт уже упоминался на стр. 195. Вместо знаков комментария в Си для «закомментаривания» фрагмента кода используют макродирективы #if 0 и #endif, которые, как и любые другие макродирективы, должны быть записаны на отдельных строках:

```
#if 0  /* this implementation was too slow */
void sort(int *a, int n)
{
   /* ... */
}
#endif
```

Всё, что написано между строками #if 0 и #endif, компилятор просто проигнорирует; там даже можно оставлять синтаксически неполные

конструкции, такие как непарные скобки в выражениях и операторах, отсутствующие условия и т. п., нельзя только допускать лексические ошибки вроде незакрытых комментариев или незавершённых строковых констант. Если потребуется снова включить такой фрагмент в компиляцию, убирать директивы `#if`/`#endif` не обязательно, достаточно заменить `#if 0` на `#if 1`.

Добавление макродирективы `#else` позволяет (опять-таки в простейшем случае) организовать выбор между разными вариантами кода. Пусть, например, мы никак не можем для себя определить, какой из двух вариантов функции `string_copy`, рассмотренных в §4.4.6, лучше использовать; применение условной компиляции позволит нам иметь в коде обе реализации, оперативно переключаться между ними и сравнивать, с какой из них программа получается лучше (например, быстрее работает):

```
#if 1
void string_copy(char *dest, const char *src)
{
    int i;
    for(i = 0; src[i]; i++)
        dest[i] = src[i];
    dest[i] = 0;
}
#else
void string_copy(char *dest, const char *src)
{
    for(; *src; dest++, src++)
        *dest = *src;
    *dest = 0;
}
#endif
```

Заменяя в макродирективе `#if` единицу на ноль и обратно, мы тем самым прикажем компилятору обрабатывать первую либо вторую версию функции, например, в зависимости от нашего настроения сегодняшним утром. Можно поступить и хитрее. В директиве `#if` на самом деле можно указать любое целочисленное константное выражение, которое компилятор сможет вычислить на этапе макропроцессирования, то есть до начала собственно компиляции; это означает, что ни к переменным, ни к функциям, ни даже к константам, введённым с помощью `enum`, мы в таких выражениях обратиться не сможем (всего этого попросту *ещё нет*, когда работает макропроцессор), но ведь существуют ещё и константы, вводимые директивой `#define`, то есть обыкновенные макросы. Итак, для начала придумаем себе имя макросимвола для выбора между реализациями; как несложно заметить, первая отличается от второй использованием индексирования, соответственно наш символ

мы так и назовём: `USE_INDEX_IN_STRING_COPY`. Строчку с директивой `#if` заменим на вот такую:

```
#if USE_INDEX_IN_STRING_COPY
```

Если соответствующий макросимвол в нашей программе не определён, то в директиве `#if` препроцессор заменит его на `0` (точнее, на `0L`, но это в данном случае неважно), и компилироваться будет, соответственно, вторая версия функции, то есть та, которая не использует индексы. Изменить это можно, вставив в программу (например, куда-нибудь в самое начало) соответствующую директиву `#define`, например,

```
#define USE_INDEX_IN_STRING_COPY 1
```

но можно поступить ещё интереснее. Мы видели, что и Free Pascal, и NASM позволяют *определять макросимволы прямо в командной строке компилятора, не трогая исходные тексты.* Естественно, такая возможность предусмотрена и в любом компиляторе Си; в частности, `gcc` для этого использует ключ `-D`, например:

```
gcc -Wall -g -D USE_INDEX_IN_STRING_COPY=1 prog.c -o prog
```

Это позволяет выбирать, какой вариант программы мы хотим собрать, *без внесения изменений в исходные тексты программы*, то есть без редактирования её исходных файлов. Этот момент часто недооценивается начинающими, но опытные программисты знают, насколько важно обходиться без «временных» изменений в файлах, составляющих программу. В серьёзных проектах любые изменения, внесённые в исходные тексты, автоматически фиксируются системами контроля версий, которые позволяют восстановить любую версию исходного текста, существовавшую в любой момент времени на протяжении разработки программы. В ряде случаев журналы изменений файлов приходится анализировать людям, время которых очень дорого стоит; наводнять историю изменений бесконечными правками в стиле «туда-обратно» значит отнимать время у своих коллег, а часто и у самого себя.

Вернёмся к нашей условной компиляции. Мы можем не полагаться на то, что компилятор заменит неопределённый макросимвол нулём; можно проверить, определён ли уже этот символ, и если нет, то определить его в тексте программы. Для этого можно воспользоваться директивами `#ifdef` и `#ifndef`, последняя в данном случае удобнее:

```
#ifndef USE_INDEX_IN_STRING_COPY
#define USE_INDEX_IN_STRING_COPY 1
#endif
```

Теперь если символ был определён в командной строке компилятора или, например, в каком-нибудь из подключённых заголовочных файлов, то он сохранит своё значение, если же его ещё никто не определил, то он будет определён и станет равным единице. К помощи `#ifdef` и `#ifndef` часто прибегают для оформления отладочной печати (см. т. 1 §2.16.3, а также §3.5.4; соответствующие конструкции в Си будут выглядеть совершенно аналогично, что и понятно, ведь в Си всё это появилось изначально, а Free Pascal и NASM *заимствовали* условную компиляцию из Си).

Остановимся подробнее на выражениях, допустимых для директивы `#if`. Эти выражения, как уже было сказано, должны быть целочисленными и вычислимыми во время работы препроцессора, то есть единственный вид идентификаторов, который в них допускается — это макроимена. Подчёркивается явным образом, что тип чисел, используемых в директивах условной компиляции — `long`. Спектр используемых операций ограничен, хотя и довольно широк: разрешается использовать все пять действий арифметики (включая взятие остатка от деления), операции сравнения, логические связки и побитовые операции, допустима также условная операция «`?:`». С другой стороны, запрещены все операции, имеющие или потенциально способные иметь побочный эффект (присваивания, инкремент/декремент, вызов функции), операции с адресами (взятие адреса, разыменование и индексирование), операция приведения типа, операция «запятая» и операция `sizeof`.

Наконец, в этих выражениях допустима одна операция, специфичная для препроцессора — это операция проверки определённости макросимвола. Она записывается с помощью слова `defined`, причём обычно её используют как *вызов псевдофункции* с круглыми скобками, хотя это и не обязательно (то есть скобки можно и не писать). Например:

```
#if defined(DEBUG_PRINT) && DEBUG_PRINT > 7
```

Упомянутые выше `#ifdef` и `#ifndef` представляют собой сокращённую запись для `#if defined()` и `#if !defined()`.

Наряду с `#if`, `#else` и `#endif` предусмотрена также директива `#elif`, означающая *else if* и предполагающая условное выражение, аналогичное тем, которые предполагаются для `#if`. При изучении макропроцессора ассемблера NASM мы рассматривали пример, когда у нас есть два «особенных» заказчика Петров и Сидоров, для которых нужно включать в программу специфические версии какого-то кода. Допустим, для всех остальных заказчиков нужно компилировать некий третий вариант; тогда мы могли бы предусмотреть символы `FOR_PETROV` и `FOR_SIDOROV`, определяемые в командной строке компилятора, а в самой программе написать что-то вроде следующего:

```
#if defined(FOR_PETROV)
```

```
   /* код для Петрова */
#elif defined(FOR_SIDOROV)
   /* код для Сидорова */
#else
   /* код для всех остальных */
#endif
```

Для `#elif` в сочетании с `defined()`, как и для `#if`, предусмотрены сокращённые названия макродиректив: вместо `#elif defined(X)` и `#elif !defined(X)` можно использовать `#elifdef X` и `#elifndef X`

Отметим, что определить символ можно, не задавая ему никакого значения (оставив его значение пустым). С помощью директивы `#define` это делается просто:

```
#define MYSYMBOL
```

В командной строке компилятора это тоже можно сделать с помощью всё того же флага `-D`:

```
gcc -Wall -g -D MYSYMBOL prog.c -o prog
```

Обычно такие макросимволы, не имеющие значений, предназначаются для управления директивами условной компиляции, то есть для использования во всевозможных `#if defined`, `#ifndef` и прочих директивах, зависящих не от значения символа, а только от факта его определённости или неопределённости.

4.10.7. Ещё несколько полезных директив

С помощью макродирективы `#undef` можно заставить препроцессор «забыть» макросимвол: после того, как препроцессор обработает строку

```
#undef MYMACRO
```

символ `MYMACRO` становится неопределённым.

Директива `#error` позволяет прервать компиляцию и выдать сообщение об ошибке, например:

```
#if !defined(FOR_PETROV) && !defined(FOR_SIDOROV)
#error Please define either FOR_PETROV or FOR_SIDOROV
#endif

#ifndef BUFFER_SIZE
#error Please specify the buffer size
#endif
```

Для случаев, когда текст на Си представляет собой результат компиляции программы с какого-то другого языка, предусмотрена директива `#line`, позволяющая заставить компилятор выдавать любую диагностику, как если бы он обрабатывал строку с заданным номером из файла с заданным именем; в обычных программах, изначально написанных на Си, эта директива никогда не используется.

Наконец, стоит упомянуть несколько макросимволов, которые определены изначально (компилятором) и не могут быть ни переопределены, ни отменены. Это символы `__LINE__`, `__FILE__`, `__DATE__` и `__TIME__`, значения которых соответствуют номеру текущей строки, имени текущего файла, текущей дате и текущему времени; номер строки представляется как целочисленная константа, остальные величины — как строковые литералы. Ещё один предопределённый макросимвол, `__STDC__`, равен единице (якобы это позволяет проверить, соответствует ли компилятор стандарту).

4.10.8. Директива `#include`

До сих пор мы сталкивались только с одним вариантом директивы `#include`, в которой имя включаемого файла обрамлялось угловыми скобками, например:

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

Препроцессор предусматривает ещё одну форму этой директивы, с двойными кавычками вместо угловых скобок:

```
#include "mymodule.h"
```

С этими двумя вариантами в наше время связана изрядная путаница, причём бóльшую часть этой путаницы, как водится, внесли всё те же особо опасные международные террористические организации, называемые по недоразумению «стандартизационными комитетами».

Изначально вариант с кавычками был предназначен для «своих» заголовочных файлов, то есть таких, которые написаны для той же программы, что и файл, из которого происходит включение, тогда как вариант с угловыми скобками — для заголовочных файлов, описывающих возможности внешних библиотек. Это простое и понятное правило почему-то не давало покоя членам комитетов по разработке стандартов, которые с упорством, достойным лучшего применения, на протяжении второго десятка лет пытаются убедить публику, что так называемая «стандартная библиотека», вопреки здравому смыслу и техническим реалиям, является частью языка Си[56]. Одним из проявлений этого

[56] Если вас уже успели в этом убедить, попробуйте ответить на один простой вопрос: если функция `printf` является *частью языка Си*, то на каком, по-вашему,

стала фиксация на уровне стандарта совершенно безумных нормативов, согласно которым вариант `#include` с угловыми скобками предназначен для «указания наборов возможностей из стандартной библиотеки», причём создатели компиляторов имеют право реализовывать этот вариант вообще без каких-либо файлов, т. е., например, заголовочный файл `stdio.h` в соответствии с этими новыми веяниями имеет право вообще не существовать; компилятор, построенный согласно этой идее, при виде хорошо знакомой нам строки `#include <stdio.h>` вместо поиска и включения заголовочного файла должен каким-то образом «включить» встроенное в него знание об объектах (функциях, типах и глобальных переменных), которые стандарт полагает описанными в рамках `stdio.h`.

К счастью, ни один компилятор не использует такой подход к реализации, поскольку это сделало бы его полностью непригодным для серьёзной работы. Тем не менее, определённые негативные последствия от неуёмной творческой активности стандартизаторов всё же наблюдаются; так, в результате всех этих нововведений в воздухе повис вопрос о том, каким же образом следует подключать заголовочные файлы *от сторонних библиотек*. В самом деле, для «своих» файлов остаётся несомненным использование `#include "..."`, для «системных» заголовочников (то есть относящихся к стандартной библиотеке) — `#include <...>`, но как быть с третьим вариантом, с заголовочными файлами, которые не являются ни частью нашего проекта, ни принадлежностью стандартной библиотеки? Ещё совсем недавно ответ на этот вопрос был очевиден: следует использовать вариант с угловыми скобками; сейчас этот ответ приходится снабжать оговоркой, что работоспособность такого решения обеспечивается исключительно здравым смыслом разработчиков компиляторов, которым хватает ума игнорировать наиболее одиозные рекомендации стандартизаторов.

Если рассмотреть вопрос с сугубо технических позиций, можно заметить, что компилятор `gcc` и многие другие компиляторы применяют, в целом, один и тот же подход к обработке двух вариантов директивы `#include`. Во время сборки самого компилятора в него зашивается некий фиксированный список «системных» директорий; кроме того, дополнительные директории для поиска заголовочных файлов можно указать через параметры командной строки, обычно с помощью ключа `-I`. Для директивы `#include <...>` компилятор выполняет поиск заданного файла сначала в директориях, указанных через командную строку,

---

языке написана сама функция `printf`? На всякий случай отметим, что она написана именно на Си, так что входить в него составной частью никак не может. Более того, ядра операционных систем пишутся без использования стандартной библиотеки; находятся люди, утверждающие на этом основании, что ядра, стало быть, написаны **не на Си**. Если учесть, что изначально Си был предназначен именно для написания ядра Unix, можно ответить этим мракобесам, что уж ядра-то точно написаны на Си, а вот то, что вы, господа, подразумеваете под «языком Си» — это не Си, а некий плод вашего чрезмерно воспалившегося воображения.

а затем в системных директориях; для директивы `#include "..."` сначала проверяется наличие указанного файла в **одной директории с файлом, из которого он включается**[57], или, точнее, наличие включаемого файла с заданным путём относительно директории, в которой находится включающий файл. Если же такого файла нет, то далее компилятор делает то же самое, что и для варианта с угловыми скобками — проверяет сначала директории, заданные в командной строке, а затем системные директории.

Очевидно, что заголовочный файл, принадлежащий сторонней библиотеке, искать в одной директории с нашими исходниками бессмысленно; больше того, если библиотека установлена в системе, то, как правило, её заголовочники доступны в тех директориях, которые компилятор считает «системными»; в частности. практически всегда для библиотек, доступных в виде пакетов в дистрибутивах Linux, заголовочные файлы устанавливаются в директорию `/usr/include`, то есть туда же, где находится `stdio.h` и прочие файлы, нежно любимые стандартизаторами.

Итак, несмотря на все потуги стандартизаторов испортить окружающий мир, рабочее решение остаётся прежним: **для включения своих собственных заголовочных файлов следует использовать вариант `#include` с двойными кавычками, а для включения любых заголовочников, не являющихся частью нашего проекта — вариант с угловыми скобками, причём вне всякой зависимости от «системности» включаемого заголовочника**.

4.10.9. Особенности оформления макродирективов

С макродирективами связан достаточно важный момент, касающийся оформления текста. Дело в том, что макродирективы, вообще говоря, *находятся вне структуры текста программы.* Во время работы макропроцессора текст ещё не до конца сформирован, даже сама структура текста программы, подчёркиваемая структурными отступами, может ещё измениться.

Исходя из этого, приходится признать некорректным рассмотрение макродирективов наравне с остальным текстом. Например, фрагмент, подобный следующему, можно встретить разве что в программах новичков:

```
void f( /* ... */ )
{
    /* ... */
    if(!key_already_known) {
        #ifdef DEBUG_PRINT
        fprintf(stderr, "DEBUG: figuring out the key\n");
```

[57] А не в текущей директории, из которой запустили компилятор, как это часто неверно полагают.

```
        #endif
        find_the_key();
    }
    /* ... */
}
```

Хотя компилятор позволяет в строках макродиректив помещать любое количество пробелов перед символом «решётки», так обычно не делают и пишут макродирективы в крайней левой позиции, не обращая внимания на окружающий текст программы — ведь во время работы макропроцессора этот текст ещё не проанализирован и на макроподстановки повлиять не может:

```
void f( /* ... */ )
{
    /* ... */
    if(!key_already_known) {
#ifdef DEBUG_PRINT
        fprintf(stderr, "DEBUG: figuring out the key\n");
#endif
        find_the_key();
    }
    /* ... */
}
```

Пока макродирективы сохраняют простую структуру, их вообще не снабжают отступами ни в каком виде, но бывают и такие ситуации, когда макродирективы во избежание путаницы могут потребовать *своих собственных* структурных отступов. В таких случаях можно воспользоваться тем, что между символом «решётки» и самой макродирективой тоже допускаются пробелы. «Решётка» ставится по-прежнему в крайней левой позиции, но макродирективы, *вложенные* в конструкцию из других макродиректив, сдвигаются вправо. Размер отступов для макродиректив может отличаться от основного размера отступов в программе. Например, возможна такая ситуация:

```
#define MAX_ARRAY_FOR_BUBBLE 30
#ifdef FIXED_ARRAY_SIZE
    int array[ARRAY_SIZE];
#else
    int *array = malloc(sizeof(*array) * arrsize);
#endif
        /* ... */
#ifdef FIXED_ARRAY_SIZE
#  if ARRAY_SIZE > MAX_ARRAY_FOR_BUBBLE
    quick_sort_int(array, ARRAY_SIZE);
#  else
```

```
        bubble_sort_int(array, ARRAY_SIZE);
#   endif
#else
    if(arrsize > MAX_ARRAY_FOR_BUBBLE)
        quick_sort_int(array, ARRAY_SIZE);
    else
        bubble_sort_int(array, ARRAY_SIZE);
#endif
```

Здесь макродирективы условной компиляции, составляющие внутренний `#if`, сдвинуты на два пробела относительно объемлющих макродирективов. Вообще говоря, подобные ситуации возникают редко, к тому же их стоит по возможности избегать, но если что-то подобное всё-таки возникло в вашей программе, желательно знать, что делать.

4.11. Раздельная трансляция

4.11.1. Общая схема раздельной трансляции в Си

Мы уже знакомы с построением программ из отдельных модулей на примере Паскаля и языка ассемблера. Средства разбиения на модули, присутствующие в Си, существенно ближе к той *раздельной трансляции*, которую мы использовали при работе на языке ассемблера, нежели к настоящей *модульности*, примером которой можно считать версии Паскаля, поддерживающие создание `unit`'ов. В сущности, поддержки модулей в Си нет вообще; всё, что есть — это средства, позволяющие делать те или иные *символы* (функции и глобальные переменные) видимыми или невидимыми для редактора связей, а также ссылаться на символы, отсутствующие в текущей единице трансляции, которые редактор связей должен потом откуда-то добыть. Всё остальное достигается использованием препроцессора и целой системы *хаков*, причём многие программисты, пишущие на Си, ухитряются в упор не видеть, что имеют дело именно с хаками, а не с чем-то иным.

Общая идея раздельной трансляции в Си такова: мы можем разбросать наши функции и глобальные переменные по разным исходным файлам, каждый из которых представляет собой текст на языке Си и обычно имеет соответствующий суффикс («`.c`») в имени. Функция с именем `main` должна при этом присутствовать только в одном из файлов, и этот файл обычно называют «главным». Далее каждый из файлов компилируется отдельным запуском компилятора, при этом компилятору сообщается, что в данном случае от него требуется только сгенерировать объектный файл, а попыток сборки финальной программы мы от него не ждём; например, компилятор `gcc` для этого поддерживает флаг командной строки «`-c`». Результатом такой компиляции становится объектный модуль, причём имя для него компилятор обычно выбирает

сам, заменяя суффикс «.c» на суффикс «.o». Например, если один из наших модулей называется mod.c, то откомпилировать его мы можем командой

```
gcc -Wall -g -c mod.c
```

в результате чего, если не произойдёт ошибок, в текущей директории появится файл mod.o. Далее у нас есть два варианта, как поступить с «главным» исходным файлом: мы можем совместить его компиляцию с вызовом редактора связей для сборки исполняемого файла, либо можно откомпилировать его точно так же, как все остальные модули, а для вызова редактора связей запустить компилятор gcc лишний раз. В обоих случаях в командной строке компилятора необходимо указать все объектные модули, которые следует использовать для построения исполняемой программы. К примеру, если наша программа состоит из главного модуля prog.c и двух дополнительных модулей mod1.c и mod2.c, мы можем собрать её так:

```
gcc -Wall -g -c mod1.c
gcc -Wall -g -c mod2.c
gcc -Wall -g prog.c mod1.o mod2.o -o prog
```

или так:

```
gcc -Wall -g -c mod1.c
gcc -Wall -g -c mod2.c
gcc -Wall -g -c prog.c
gcc prog.o mod1.o mod2.o -o prog
```

Теоретически без последнего запуска компилятора можно обойтись, запустив сразу редактор связей (напомним, он называется ld), но для этого нам надо знать, где находится файл, содержащий стандартную библиотеку Си. Компилятор этим знанием уже обладает, поэтому проще поручить запуск редактора связей ему, а не делать это самим.

4.11.2. Видимость объектов из других модулей

Напомним, что при построении раздельно транслируемой программы на языке ассемблера NASM мы использовали директивы global и extern; первая делает метку из текущего модуля видимой для редактора связей (и, как следствие, для других модулей), а вторая объявляет метку, которой в текущем модуле нет, но наличие которой предполагается в каком-то другом модуле той же программы. Все остальные метки, не попавшие под действие ни одной из этих директив, NASM считал локальными для текущей единицы трансляции и оставлял их невидимыми для редактора связей.

Подход, принятый в Си, прямо противоположен: по умолчанию компилятор делает все «глобальные объекты» (то есть функции, а также глобальные переменные) видимыми для редактора связей; аналога директивы `global` в Си, соответственно, нет, но зато есть противоположное по смыслу ключевое слово `static`: глобальные объекты, помеченные этим словом, для редактора связей останутся невидимы, так что добраться до них из других единиц трансляции невозможно. Например, если мы уверены, что какая-то функция из написанных нами настолько специфична, что её вызов из других модулей никогда не понадобится, мы можем сделать её полностью невидимой извне нашего модуля:

```
static int very_local_subroutine(int x, const char *n)
{
   /* ... */
}
```

Точно так же мы можем поступить и с глобальной переменной; конечно, лучше вообще их не использовать, но если всё же пришлось, стоит подумать на тему ограничения доступа к ней: глобальная переменная, видимая только в одном модуле, всё же лучше, чем глобальная переменная, к которой могут обращаться функции из других модулей. Достигается это с помощью того же слова `static`:

```
int very_bad_global_var;     /* видно отовсюду, очень плохо */
static int a_bit_better_var; /* только из текущего модуля */
```

Для доступа к функциям и переменным, которые описаны в других модулях, в текущем модуле помещают их ***объявления***; объявление, как мы помним, сообщает компилятору, что соответствующий объект *где-то существует* и можно не пугаться его отсутствия, оставив решение проблемы редактору связей. Объявлением для функции, как мы уже знаем, служит её заголовок; так, если в одном из исходных файлов описана функция

```
int cube(int x)
{
    return x * x * x;
}
```

то в любом другом из наших исходных файлов мы можем поместить её заголовок:

```
int cube(int x);
```

и использовать её обычным образом, не опасаясь ошибок. Компилятор, обрабатывая вызов такой функции, проверит соответствие типов фактических параметров заявленным в заголовке, а также соответствие

типа возвращаемого значения контексту вызова, после чего, если всё хорошо, спокойно откомпилирует такой вызов, оставив в объектном коде требование к редактору связей добыть где-нибудь функцию с таким именем и подставить её адрес в инструкцию вызова. Видеть тело функции компилятору для этого не нужно; собственно говоря, в этом и состоит преимущество раздельной трансляции: при обработке одного исходного файла компилятор не тратит время на анализ кода, содержащегося в других файлах. Конечно, остальные файлы тоже придётся откомпилировать, но если компиляция уже проходила и у вас есть объектные файлы для всех ваших модулей, то при внесении изменений только в один или два из них вам не придётся компилировать их все: достаточно будет перекомпилировать те модули, которые изменились, а затем выполнить финальную сборку, которая происходит достаточно быстро даже для очень больших программ.

Несколько сложнее обстоят дела с объявлением (в отличие от описания) глобальных переменных. Если в *другом* модуле есть глобальная переменная, к которой вы хотите получить доступ из текущего исходного файла, необходимо в этом файле повторить описание этой переменной, добавив к нему ключевое слово `extern`:

```
extern int very_bad_global_variable;
```

после чего её можно будет использовать, как если бы она была описана в текущем модуле.

Пользуясь случаем, напомним, что глобальные переменные есть частный случай абсолютного вселенского зла, а сцепленность модулей по глобальным переменным — это самый худший вариант взаимодействия между модулями. Это, впрочем, не относится к константам, то есть таким переменным, значение которых никогда не меняется. Например, если в одном модуле есть глобальный константный массив

```
const int prime_numbers[] = { 2, 3, 5, 7, 11, 13, 17 /* ... */
```

то в любом другом модуле можно написать

```
extern const int prime_numbers[];
```

и спокойно использовать уже готовую таблицу простых чисел из другого модуля вместо того чтобы делать свою. Такое использование ничем не плохо, ведь переменная, которая не меняется, не может *накапливать состояние* и оказывать неочевидное влияние на работу функций — как раз всё то, из-за чего глобальные переменные нежелательны.

4.11.3. Заголовочные файлы к модулям

Допустим, у нас есть модуль `m1.c`, в котором описан десяток-другой функций; этими функциями активно пользуются ещё пять-шесть модулей. Если буквально следовать инструкциям, предложенным в предыдущем параграфе, то в каждом из этих пяти-шести модулей нам придётся поместить заголовки всех функций из `m1.c`, которые в них используются. По мере увеличения проекта будет расти и общее количество таких заголовков, и расти оно будет *нелинейно*; в какой-то момент мы можем обнаружить, что такие заголовки составляют уже процентов двадцать всего объёма кода. Само по себе это ещё не очень страшно, но ведь функции иногда меняются, причём может измениться в том числе и информация, представленная в заголовке: мы можем принять решение добавить или убрать параметр, изменить тип параметра или даже тип возвращаемого значения. Каждый раз, когда мы вносим подобное изменение, нам придётся найти все файлы нашей программы, в которых имеется копия изменившегося заголовка, и соответствующим образом их отредактировать. Если забыть это сделать, *программа благополучно откомпилируется и соберётся*, но работать, разумеется, будет неправильно; в таких случаях возникают настолько феерические ситуации, что неопытные программисты, оторвавшись от экрана с отладчиком, недоумённо восклицают: «Но ведь такого не может быть!».

В самом деле, пусть у нас была функция `int func(int x, int y);`, и именно такой заголовок фигурировал в нескольких модулях. Мы приняли решение добавить в эту функцию третий параметр, `int z`, и отредактировали все эти модули, что само по себе нетривиально — про существование некоторых из них мы можем даже не догадываться, ведь над серьёзными программами обычно работает несколько человек, и кто-то вполне мог воспользоваться нашей функцией, не сообщив нам об этом, а если и сообщил, мы об этом, скорее всего, тут же забыли. Но, так или иначе, для поиска модулей, использующих функцию, есть много довольно простых инструментов, начиная с программы `grep`, которая позволяет произвести простой текстуальный поиск по всему нашему набору исходников. Итак, мы отредактировали все модули — кроме одного, про который случайно забыли. Во всех модулях, кроме этого одного, компилятор точно знает, что функция `func` должна вызываться от трёх параметров, так что, пока мы не исправим все её вызовы, добавив третий параметр, компилятор будет выдавать ошибку; но вот при компиляции последнего оставшегося модуля компилятор по-прежнему уверен, что параметров там всего два, и именно так в этом модуле функция `func` и вызывается. Сама функция, понятное дело, ожидает наличие в стеке трёх параметров, а не двух. Третий параметр она из стека в любом случае извлечёт, но вряд ли кто возьмётся предсказать,

что там окажется, когда функцию вызовут с двумя параметрами вместо трёх.

Ещё интереснее будет картина, если мы решим изменить тип одного из параметров с `int` на `double` и при этом где-то забудем подредактировать заголовок; ну а при смене типа *возвращаемого* значения (опять же, если забыть где-то это исправить) можно ожидать полного хаоса. Впрочем, степень полноты хаоса — вопрос философский; любой из перечисленных вариантов ошибки может стоить вам нескольких потерянных дней.

Во избежание таких ошибок программисты обычно не пишут в файлах модулей заголовки функций из других модулей. Вместо этого они применяют *заголовочные файлы*; в большинстве случаев отдельным заголовочным файлом снабжают каждый модуль, кроме главного, то есть на каждый файл с суффиксом `.c` за исключением файла, содержащего функцию `main`, заводят файл с таким же именем, но с суффиксом `.h` (например, для модуля `m1.c` создают заголовочный файл `m1.h`). В такой заголовочный файл выносят **объявления всех объектов, описанных в модуле, к которым предполагаются обращения из других модулей**; иначе говоря, если в вашем модуле имеется функция `func` и вы предполагаете, что в других модулях к этой функции будут обращения, то её заголовок нужно вынести в заголовочный файл вашего модуля. Аналогично, если в вашем модуле описана глобальная переменная и вы предполагаете её доступность из других модулей (вы хорошо подумали?), то в заголовочном файле следует поместить её *объявление* со словом `extern` (в отличие от *описания* без слова `extern`, которое остаётся в самом модуле).

Теперь достаточно включить этот заголовочный файл с помощью директивы `#include` (с кавычками) в каждый из модулей, использующих возможности вашего модуля, и дело, как говорится, в шляпе: при трансляции всех этих модулей компилятор будет видеть ваши объявления, но сами эти объявления не «размножаются», как описывалось в начале параграфа, а существуют в одном экземпляре — в вашем заголовочном файле. Если теперь вам придёт в голову поменять параметры какой-то из ваших функций, вам не придётся искать её заголовки по всем модулям программы, достаточно будет отредактировать ваш собственный заголовочник — то есть **один** файл (конечно, не считая самого вашего модуля). Все случаи несоответствия параметров вызовов параметрам нового заголовка вам укажет компилятор в процессе пересборки программы.

Отметим ещё два очень важных момента. Во-первых, **в модуль обязательно нужно включать (с помощью `#include`) *свой собственный* заголовочный файл**. В этом случае компилятор при обработке вашего модуля сначала увидит объявления функций и переменных, содержащиеся в заголовочнике, а потом их же *описания*,

находящиеся в тексте самого модуля; это позволит компилятору проверить их соответствие и выдать ошибку, например, если вы изменили профиль какой-нибудь функции в модуле, но отразить изменения в заголовочном файле забыли.

Во-вторых, **все функции и глобальные переменные, которые вы решили не выносить в заголовочник, нужно обязательно пометить модификатором** `static` (см. стр. 360), тем самым сделав их невидимыми для редактора связей. Глобальное пространство имён у вас одно на всю программу, при этом программисты, как правило, обращают внимание только на содержимое заголовочников, не вникая в то, что написано в модулях, особенно если эти модули написаны другими участниками работы; имя, которое вы не вынесли в заголовочник, но оставили видимым, может стать причиной неожиданного конфликта имён, ведь кто-то из ваших коллег вполне может задействовать такое же имя в одном из своих модулей. Вообще говоря, все глобальные имена, описанные в вашем модуле, делятся на две категории: те, которые вы экспортируете, и те, которые вы описали для использования в самом модуле; первые выносятся в заголовочник (для функций — прототипы, для переменных — описания со словом `extern`), вторые помечаются словом `static`. Если не сделать ни того, ни другого, можно будет предположить, что вы сами не знаете, для чего ввели то или иное имя.

Единственным исключением из этого правила является функция `main`: в заголовочник её, как правило, не выносят, но редактор связей должен её видеть, иначе собрать исполняемую программу он не сможет.

4.11.4. Описания типов и макросов в заголовочных файлах; защита от повторного включения

Пока ваши модули экспортируют только функции и глобальные переменные, всё сравнительно просто; к сожалению, в реальной жизни так не бывает. Скорее всего, уже на ранних стадиях работы вам потребуется ввести какой-нибудь *пользовательский тип*, будь то структура, объединение или перечисление, с которым будут работать ваши функции и/или который будет использован при описании глобальных переменных. Такой тип потребуется сделать доступным более чем в одном модуле; как следствие, его описание придётся вынести в заголовочный файл. Больше того, если в *другом* модуле появятся экспортируемые функции, работающие с вашим типом, вам может потребоваться включить *один заголовочный файл из другого*: в том заголовочнике, где объявлены функции, использующие ваш тип, придётся предусмотреть директиву `#include`, включающую заголовочник, содержащий описание этого типа.

Кроме пользовательских типов, существуют ещё макросы; их тоже приходится с целью экспорта выносить в заголовочные файлы, и

они тоже (хотя и существенно реже) могут оказаться причиной для включения одного заголовочника из другого.

Можно выделить общее правило для выноса тех или иных сущностей в заголовочные файлы:

- всё, что ***занимает память*** в какой-либо из секций будущей исполняемой программы, ***описывается*** в файле реализации модуля (файл с суффиксом `.c`) и либо ***объявляется*** в заголовочном файле (если экспортируется), либо помечается словом `static` (если объект предназначен только для внутреннего использования в модуле);
- всё, что ***не занимает памяти***, или, иначе говоря, всё, что используется только во время компиляции и затем бесследно исчезает, ***описывается*** в заголовочном файле (либо, если предназначено только для внутреннего использования в модуле, описывается в файле реализации).

К первой категории сущностей относятся функции и глобальные переменные, ко второй — типы и макросы. Функции объявляются своими заголовками-прототипами, переменные объявляются с помощью слова `extern`, что до описаний типов и макросов, то они в заголовочном файле выглядят точно так же, как и везде.

Теперь представьте, что вам потребовалась какая-то функция, написанная вашим коллегой в другом модуле. Заглянув в документацию или просто задав вопрос коллеге, вы узнали, что эта функция находится в модуле `ddd.c`, который снабжён заголовочным файлом `ddd.h`. Ваш логичный следующий шаг — вставить в свой модуль директиву `#include "ddd.h"` и, воспользовавшись нужной функцией, продолжить работу. При этом вряд ли вы станете подробно изучать файл `ddd.h`, а в нём, вполне возможно, тоже содержится директива `#include`, включающая ещё какой-нибудь заголовочник, например, `bbb.h`. При этом вам ещё месяц назад потребовались какие-то функции из модуля `bbb`, так что в вашем собственном тексте тоже есть `#include "bbb.h"`. Компилятор, таким образом, в процессе обработки вашего модуля «увидит» `bbb.h` дважды; это называется ***повторным включением*** заголовочного файла.

В принципе, если бы `bbb.h` содержал только объявления переменных и функций, ни к чему страшному такое повторное включение не привело бы, только чуть-чуть замедлило бы компиляцию. Но это вряд ли наш случай: заголовочники, в которых содержатся только функции и переменные, нет никакого смысла включать из других заголовочников. Так что если повторное включение произошло — видимо, в повторно включённом заголовочном файле описаны пользовательские типы и (или) макросы. В случае с типом компилятор попросту выдаст ошибку, ведь он увидел второе *описание* того же самого имени, а описывать

одно и то же имя дважды нельзя (в отличие от объявлений; объявлять одно и то же имя можно сколько угодно раз, главное, чтобы эти объявления друг другу не противоречили). В случае с макросом компилятор выдаст предупреждение, а не ошибку, но это тоже не слишком хорошо; в большинстве проектов действует требование полного отсутствия предупреждений при компиляции.

Первое, что приходит в голову — разобраться, какой из файлов включён повторно (это сделать довольно просто: компилятор при выдаче ошибки или предупреждения сам скажет, в каком файле и какой строке возникла проблема) и убрать из своего модуля соответствующую директиву `#include`, в данном случае ту, которая включает `bbb.h`; в самом деле, коль скоро этот файл всё равно включается через `ddd.h`, то этого нам уже достаточно и нет повода включать его самим. Однако и тут реальность оказывается несколько сложнее, нежели на первый взгляд: представьте себе ситуацию, когда вы включили в свой модуль файлы `bbb.h` и `ddd.h` и никакого повторного включения не произошло, а уже позже, возможно, через полгода или год, кто-то решил усовершенствовать файл `ddd.h` и в процессе совершенствования добавил там включение `bbb.h`, в результате чего ваш ни в чём не повинный модуль, который вы уже полгода как не трогали, вдруг перестал компилироваться. Возможны и более заковыристые ситуации; придумать их мы предложим читателю в качестве головоломки.

Как показывает практика, бороться вручную с каждым случаем повторного включения — идея совершенно неработоспособная, так что программистам пришлось изобрести некую технику, автоматически не позволяющую заголовочникам включаться в одну и ту же единицу трансляции больше одного раза. Для этого используется препроцессор, а точнее — его возможности *условной компиляции*, описанные в §4.10.6. Для каждого заголовочного файла выбирается некое уникальное имя макросимвола, обычно включающее в себя само имя файла, приведённое к верхнему регистру; так, автор этих строк обычно использует суффикс «`_SENTRY`», так что для заголовочника `mymodule.h` получается что-то вроде `MYMODULE_H_SENTRY`, но возможны и другие подходы. Главное — позаботиться о том, чтобы вероятность появления где-то символа с таким же именем была достаточно близка к нулю. Далее в начале заголовочного файла (то есть прямо первыми двумя его строками; возможно, после комментария, в котором описывается, что это за файл и для чего он нужен) пишутся директивы

```
#ifndef MYMODULE_H_SENTRY
#define MYMODULE_H_SENTRY
```

а последней строкой того же заголовочного файла ставится

```
#endif
```

Таким образом всё «значащее» (то есть как-то влияющее на компиляцию) содержимое нашего заголовочника оказывается обрамлено директивой условной компиляции. При обработке *первого* случая включения данного файла в отдельно взятой единице трансляции символ `MYMODULE_H_SENTRY` ещё не определён, так что содержимое заголовочника компилятором обрабатывается, причём, в числе прочего, срабатывает и директива `#define`, которая определяет символ. Если *в той же единице трансляции* файл `mymodule.h` включается второй раз (а равно и третий, и четвёртый, и последующие) — к этому времени символ `MYMODULE_H_SENTRY` уже определён, так что директива `#ifndef` исключает содержимое файла из компиляции, и компилятор его повторно не увидит. Иначе говоря, повторное включение (физически) происходит, но никак не влияет на ход компиляции.

Интересно, что эта техника защиты от повторного включения, известная под названием «защитные макросы» (*sentry macros*), многими программистами на Си воспринимается как должное, то есть люди не видят подлинной сущности этой техники; между тем это не более чем очередной *хак*, использование подвернувшегося под руку инструмента совершенно не по его прямому назначению, но приводящее при этом к нужному результату.

Как мог заметить читатель, использование защиты от повторного включения не всегда оказывается обязательным; если очередной заголовочный файл содержит только объявления функций и переменных, то его повторное включение ни к ошибкам, ни даже к предупреждениям не приведёт. Однако исходные файлы нашей программы, в том числе заголовочные — это отнюдь не мраморные скрижали, они достаточно часто и активно меняются; если мы будем снабжать защитой от повторного включения только такие заголовочники, в которых присутствуют типы или макросы, то каждый раз, вставляя в какой-нибудь заголовочный файл описание очередного нового типа или макроса, мы будем вынуждены проверять, есть ли уже в этом заголовочнике защита от повторного включения. Практика показывает, что гораздо проще снабжать **каждый** заголовочный файл защитой от повторного включения сразу, как только этот файл создан. Написать три короткие строки текста в пустом файле — вопрос нескольких секунд, при этом *каждая* проверка наличия этих строчек в файле более-менее осмысленного размера может занять столько же и даже больше.

Итак, **каждый заголовочный файл следует снабжать директивами защиты от повторного включения сразу же после его создания**; это один из тех редких случаев, когда думать вредно. Ещё одно действие, выполняемое всегда сразу после создания файла модуля – это добавление директивы `#include`, включающей в модуль его собственный заголовочный файл. Вообще, процедуру создания в программе нового модуля (назовём его `newmod`) можно описать так:

- создайте пустые файлы `newmod.c` и `newmod.h`;

- вставьте в файл `newmod.c` директиву
  ```
  #include "newmod.h"
  ```
- вставьте в файл `newmod.h` директивы
  ```
  #ifndef NEWMOD_H_SENTRY
  #define NEWMOD_H_SENTRY



  #endif
  ```
  оставив между ними пустое пространство для содержимого заголовочного файла;
- зарегистрируйте созданные файлы в системе контроля версий и системе сборки;
- при необходимости отразите появление нового модуля в документации.

4.11.5. Объявления типов; неполные типы

До сих пор мы сталкивались с *объявлениями* переменных и функций; для пользовательских типов, включая структуры, объединения и перечисления, мы использовали только *описания*, то есть конструкции, задающие исчерпывающую информацию о типе. Как ни странно, структуры в Си также можно *объявлять*, то есть сообщать компилятору о существовании типа с заданным именем, не показывая собственно сам тип. Например, можно написать:

```
struct item;
```

С этого момента компилятор будет знать, что словосочетание `struct item` обозначает некий структурный тип, но больше ничего про этот тип знать не будет; такой тип называется «неполным» (англ. *incomplete type*). Точно так же можно описать неполный тип-объединение, но это обычно не используется.

Конечно, описать переменную неполного типа компилятор не позволит, ведь он даже не знает, сколько памяти под такую переменную необходимо выделить; однако неполный тип можно вполне успешно использовать для описания *указателей*: в самом деле, размер указателя обычно не зависит от того, на переменную какого типа он указывает.

Основное назначение таких объявлений состоит в том, чтобы позволить, например, двум структурам иметь в качестве полей указатели друг на друга. Более того, мы уже использовали неполные типы, когда для работы со списками и деревьями описывали структуры с полями-указателями на структуру того же типа; например, в знакомом нам описании

```
struct item {
    int data;
    struct item *next;
};
```

на тот момент, когда компилятор видит описание поля `next`, сам тип `struct item` ещё не описан, ведь его описание ещё не закончилось; с другой стороны, поскольку словосочетание `struct item` компилятор уже видел, этот тип считается

объявленным (хотя и неполным), и компилятор полагает, что для описания указателя этого достаточно.

Вообще говоря, писать объявление структуры отдельной конструкцией особого смысла нет, поскольку использовать такой неполный тип можно лишь указав его полное имя, включающее слово `struct` (в данном случае `struct item`), а это уже само по себе послужит для компилятора объявлением типа. Как следствие, тип `struct item*` можно использовать, даже если до сей поры компилятор вообще ничего не слышал о слове `item`.

Кроме очевидного применения для создания структур со ссылками на себя или друг на друга, неполные типы дают ещё одну важную возможность: *не показывать компилятору описание типа, если нужен только указатель на такой тип*. Это часто позволяет **написать объявление структуры (неполной), вместо того чтобы включать с помощью** `#include` **тот файл, где находится её полное описание**. Так можно достичь изрядной экономии времени компиляции, выкинув лишние включения, в особенности если речь идёт о включении одного заголовочного файла из другого; кроме того, никакие два заголовочных файла по понятным причинам не могут взаимно включать друг друга, так что объявления типов оказываются просто незаменимы.

Иногда описание структуры вообще не выносят в заголовочный файл, несмотря на то, что в нём содержатся прототипы функций, работающие с указателями на эту структуру. Такой вариант применяется, если автор модуля предлагает пользователю хранить указатели на какую-то его структуру, например, для целей идентификации, но при этом предпочитает, чтобы пользователь не обращался к полям этой структуры напрямую. Кстати, примером такой ситуации является хорошо знакомый нам тип `FILE*`.

Отметим, что большинство компиляторов позволяют также объявить тип `enum`, но это в явном виде запрещено стандартами и к тому же совершенно бессмысленно: проще тогда использовать обыкновенный `int`.

4.12. И снова об оформлении кода

Во всём предшествующем тексте, начиная с первого тома, мы уделяли правилам оформления текста программы самое пристальное внимание, но тем не менее нам по-прежнему есть что обсудить в этой области. Мы надеемся, что к настоящему времени читатель уже приобрёл достаточный опыт в написании программ и лучше подготовлен к восприятию некоторых тонкостей.

Перечислим основополагающие правила, которые нам известны. Прежде всего напомним, что **текст программы предназначен в первую очередь для прочтения человеком, и лишь во вторую — для обработки компьютером**. Обеспечение хорошей читаемости текста — задача первоочередная; программа, в тексте которой нарушены правила оформления, вообще не может рассматриваться в качестве правильной или неправильной, работающей или неработающей, полезной или бесполезной. До тех пор, пока нарушения в оформлении не

будут устранены, они представляют собой *единственное* свойство программы, которое можно обсуждать, а их устранение — это единственное, что с такой программой допустимо делать.

К тексту программы на любом языке программирования предъявляется несколько универсальных требований. Текст должен содержать только символы из набора ASCII; комментарии, если они есть, должны быть написаны по-английски (не по-немецки, не по-французски и тем более не по-русски транслитом, а именно по-английски); при выборе имён для идентификаторов также должны использоваться именно английские слова, адекватно отражающие предназначение идентификатора. Короткие имена можно (и нужно) использовать только для локальных идентификаторов в случаях, если их предназначение очевидно из контекста. Комментариями лучше не злоупотреблять; вместо этого следует писать текст самой программы (в частности, выбирать идентификаторы) так, чтобы он пояснял сам себя. Выбору идентификаторов был посвящён §2.15.7.

При написании программы на языке, имеющем структурный синтаксис, то есть подразумевающем вложенные конструкции (а именно таковы и Паскаль, и Си, и большинство других языков программирования) **необходимо визуально выделять вложенные фрагменты, используя пробельные символы в начале каждой строки**. Вложенный текст должен сдвигаться вправо относительно текста, в который он вложен, на фиксированный размер отступа — два, три, четыре пробела или один символ табуляции. Строки, которые начинают и заканчивают цельную конструкцию, должны иметь одинаковый отступ, то есть их первые непробельные символы обязаны находиться точно друг над другом. Подробное обсуждение этого можно найти в первом томе, §2.1 (см. стр. 204–206).

Существует три допустимых подхода к расположению операторных скобок. Открывающую скобку можно сносить на следующую строку после заголовка оператора, располагая и её, и закрывающую скобку в той же горизонтальной позиции, в которой расположен заголовок оператора — так мы поступали в программах на Паскале. Можно оставлять открывающую скобку на одной строке с заголовком оператора, а закрывающую располагать под началом заголовка — именно так мы действовали в программах на Си. Наконец, существует третий стиль, который мы не рекомендуем, но имеем в виду его допустимость: операторные скобки в этом стиле снабжаются дополнительным отступом. Все три варианта мы подробно рассмотрели, впервые столкнувшись с составным оператором (см. т. 1, §2.3.3, стр. 225).

Ещё одна важная подробность: **операторную скобку, обозначающую начало тела функции, всегда сносят на отдельную строку**, даже если при написании сложных операторов (ветвлений и циклов) скобку решено оставлять на одной строке с заголовком; более того,

операторные скобки, обрамляющие тело функции, никогда не сдвигают — они всегда пишутся в крайней левой колонке текста вне всякой зависимости от избранного стиля. Для Паскаля это требование очевидно, достаточно вспомнить о секциях локальных описаний; для Си, возможно, это требование покажется не столь очевидным, но оно и здесь очень просто объясняется. Дело в том, что, в отличие от заголовка сложного оператора, **заголовок функции имеет самостоятельное значение**, то есть часто встречается отдельно от тела функции — при этом он служит *объявлением* соответствующей функции, в противоположность *описанию*, в котором присутствует тело (в Си в объявлениях функций вместо тела ставится точка с запятой). Разнесение заголовка и тела на разные строки позволяет подчеркнуть факт самостоятельности заголовка. Что касается дополнительного отступа для для всего тела, то он был бы попросту излишним: тело функции может прилагаться к её заголовку, но оно не вложено в заголовок. Со сложными операторами ситуация иная, там имеет место как раз вложение одного оператора в другой, поэтому, несмотря на то, что мы такой стиль не рекомендуем, в пользу дополнительного сдвига составного оператора в составе другого оператора можно найти определённые аргументы.

Итак, следующие варианты оформления будут неправильными:

```
float cube(float a)
    {
        return a * a * a;
    }
```

```
float cube(float a) {
    return a * a * a;
}
```

```
float cube(float a)
{ return a * a * a; }
```

Правильно будет написать так:

```
float cube(float a)
{
    return a * a * a;
}
```

4.12.1. Фирменные особенности Си

Начнём с соглашений об именах идентификаторов. В отличие от Паскаля, язык Си чувствителен к регистру букв, так что никакой свободы в написании ключевых слов здесь нет; например, `if` —

это название оператора, тогда как `If`, `iF` и `IF` — это обыкновенные идентификаторы, притом различные.

Из §4.10.3 мы уже знаем, что в программах на Си имена макросов обычно набираются целиком заглавными буквами, чтобы их было лучше видно (а нужно это по причине опасности макросов, обусловленной их неподчинением общим правилам видимости). Как можно догадаться по приводившимся примерам, все остальные идентификаторы в языке Си обычно набирают целиком в нижнем регистре, т. е. маленькими буквами, а отдельные слова в составе идентификаторов отделяют друг от друга символом подчёркивания. **Идентификаторы, сочетающие в себе буквы обоих регистров, при работе на языке Си обычно не используются**. Если вы увидели программу на Си, в которой имеются идентификаторы «смешанного регистра», такие как `MixedCase`, `isItGood`, `CamelIsAnAnimal`, `exGF` и прочее в таком духе — скорее всего, автор программы редко пишет на Си. Отметим, что сказанное верно лишь для чистого Си; традиции языка Си++ совершенно иные, но о них пока речи не идёт.

Иногда можно встретить, в том числе в системных библиотеках и заголовочных файлах, применение «смешанного регистра» для именования структурных типов, а в некоторых случаях — и для других целей. Было бы неправильно утверждать, что это *недопустимо*, коль скоро ясности программ такой стиль не вредит. Тем не менее мы не рекомендуем использовать такие отступления от традиционного именования.

Несколько особняком стоят идентификаторы, имена которых начинаются с подчёркивания. Традиционно считается, что эти имена *зарезервированы за системой программирования*, то есть, например, разработчики системных заголовочных файлов могут использовать такие имена для своих внутренних нужд, не внося их в документацию. В связи с этим не следует пользоваться именами, начинающимися с подчёркивания, в своих программах: такие имена (например, `_counter`, `_error_code`, `__LINES_NUM` и т. п.) могут вступить в конфликт с заголовочными файлами, поставляемыми вместе с компилятором. Следует учитывать, что, даже если одно конкретное имя ни к каким конфликтам не привело, это не является основанием для дальнейшего его использования: одно из основных достоинств языка Си — *переносимость программ*, а использование потенциально «конфликтных» имён эту переносимость, естественно, снижает, ведь если в вашей версии библиотечных заголовочников соответствующего имени нет, то это не значит, что такое же имя не появится при работе с другим компилятором, на другой платформе, да и просто в одной из будущих версий вашей системы программирования. Итак, **не начинайте имена ваших идентификаторов с символа подчёркивания**, оставьте такое именование системным заголовочным файлам.

Оформление сложных объявлений, описаний и инициализаторов тоже имеет свои особенности. В языке Си часто встречаются описания, имеющие сложный синтаксис: это описания структурных и перечислимых типов, а также снабжённые инициализаторами описания массивов и переменных-структур.

С описанием типа, предполагающим использование фигурных скобок (`struct`, `union`, `enum`) всё довольно просто: единственная степень свободы — положение открывающей фигурной скобки (её, как водится, можно оставить на одной строке с именем структуры, а можно снести на следующую строку). Чего точно не следует делать — это сдвигать фигурные скобки относительно начала описания типа, то есть вот так писать можно:

```
struct item {
    const char *str;
    item *next;
};
```

```
struct item
{
    const char *str;
    item *next;
};
```

а вот так уже не стоит:

```
struct item
    {
        const char *str;
        item *next;
    };
```

Ситуация здесь подобна ситуации с началом и концом тела функции: всё тело целиком не сдвигают даже при использовании стиля, предполагающего сдвиг составного оператора относительно заголовка сложного оператора. Отметим, что не стоит также и вытягивать описание структуры в одну строку:

```
struct item { const char *str; item *next; };
```

А вот описание перечислимого типа вытянуть можно, и смотреться это будет вполне логично, но только в случае, если оно умещается в одну строку и не содержит явно заданных значений констант:

```
enum state { home, whitespace, stringconst, ident, end };
```

Если в одну строку описание не поместилось или если имеется потребность в явном задании целочисленных значений[58], то лучше будет каждую константу описывать на отдельной строке. Обратите внимание, что для удобства чтения знаки равенства рекомендуется расположить в одну колонку (используйте пробелы, если пользуетесь

[58] Кроме случая описания анонимного псевдотипа с целью введения *одной* константы; этот случай рассмотрен в предыдущем параграфе.

ими для структурных отступов; если в качестве отступа у вас табуляция, используйте её же для выравнивания знаков равенства):

```
enum state {                          enum state {
    st_home,                              st_home         = 0,
    st_whitespace,                        st_whitespace   = 32,
    st_stringconst,                       st_stringconst  = 12,
    st_ident,                             st_ident        = 77,
    st_end                                st_end          = -1
};                                    };
```

Как обычно, положение открывающей фигурной скобки определяется избранным стилем. Если вы сносите её в описаниях структур, сделайте то же самое и при описании `enum`'ов:

```
enum state                            enum state
{                                     {
    st_home,                              st_home         = 0,
```

Аналогично обстоят дела с длинными инициализаторами (например, при описании инициализированных массивов). Если инициализатор полностью умещается в одну строку, лучше его так и записать:

```
const int some_primes[] = { 11, 17, 37, 67, 131, 257 };
```

Если инициализатор в одну строку не поместился, или если к значениям требуются комментарии, можно записать каждый элемент инициализатора на отдельной строке:

```
const unsigned long hash_sizes[] = {
    11,     /* > 8 */
    17,     /* > 16 */
    37,     /* > 32 */
    67,     /* > 64 */
    131,    /* > 128 */
    257,    /* > 256 */
    521,    /* > 512 */
    1031,   /* > 1024 */
    2053    /* > 2048 */
};
```

Нет ничего страшного и в разбиении инициализатора на несколько строк из соображений равномерности:

```
const unsigned long hash_sizes[] = {
    11,       17,       37,       67,       131,     257,
    521,      1031,     2053,     4099,     8209,    16411,
    32771,    65537,    131101,   262147,   524309,  1048583,
    2097169,  4194319,  8388617,  16777259, 33554467
};
```

Положение открывающей фигурной скобки в этих случаях также зависит от избранного стиля, причём, как ни странно, часто её сдвигают, то есть следующий вариант не только допустим, но и довольно популярен:

```
const unsigned long hash_sizes[] =
    {
        11,     /* > 8 */
        17,     /* > 16 */
        37,     /* > 32 */
        67,     /* > 64 */
        131,    /* > 128 */
        257,    /* > 256 */
        521     /* > 512 */
        1031,   /* > 1024 */
        2053    /* > 2048 */
    };
```

При инициализации двумерного массива обычно каждый элемент (то есть одномерный массив) стараются записать в одну строку, например:

```
const int change_level_table[5][5] = {
    { 4, 4, 2, 1, 1 },
    { 3, 4, 3, 1, 1 },
    { 1, 3, 4, 3, 1 },
    { 1, 1, 3, 4, 3 },
    { 1, 1, 2, 4, 4 }
};
```

Если же в одну строку кода каждая строка вашей матрицы не помещается (случай сам по себе довольно редкий), придётся их равномерно разбить на несколько строк.

Оформление оператора постусловия в Си тоже имеет свои особенности. Ключевое слово `while` в этом языке используется в двух ролях: в заголовке цикла с предусловием и в эпилоге цикла с постусловием (`do-while`). Сразу после слова `while` в обеих ситуациях следует условное выражение в круглых скобках; в конструкции `do-while` следом за этим выражением идёт точка с запятой, но такая же точка с запятой может обнаружиться и после заголовка обычного `while` — там она будет изображать пустое тело цикла. Итак, глядя на конец цикла `do-while`, можно (и очень легко) перепутать его с циклом `while`, имеющим пустое тело.

Решение этой проблемы довольно очевидно для стиля расстановки фигурных скобок, при котором открывающая скобка не сносится на следующую строку. Прежде всего отметим, что **использование фигурных скобок в цикле `do-while` строго обязательно вне всякой**

зависимости от количества операторов в этом теле[59]. Проблема со смыслом `while` теперь решается тем, что слово `while` попросту остаётся на той же строке, что и предшествующая ему закрывающая скобка, примерно так, как мы поступали со словом `else`:

```
do {
    get_event(&event);
    res = handle_event(&event);
} while (res != EV_QUIT_NOW);
```

Такое же решение мы можем применить и для стиля, при котором открывающая фигурная скобка сносится на следующую строку, но не сдвигается:

```
do
{
    get_event(&event);
    res = handle_event(&event);
} while (res != EV_QUIT_NOW);
```

Это не слишком изящно, поскольку вынуждает нас вводить исключение из общего правила, однако так всё же лучше, чем путать конец цикла с началом нового цикла. Но вот что делать при использовании стиля, где скобки не только сносятся, но и сдвигаются, оказывается непонятно. Ричард Столлман в GNU Coding Style Guide предлагает оформлять `do-while` так:

```
do
  {
    get_event(&event);
    res = handle_event(&event);
  }
while (res != EV_QUIT_NOW);
```

При всём уважении к Столлману, эта идея крайне неудачна. Если цикл занимает хотя бы десяток строк, при взгляде на слово `while` мы совершенно однозначно не заметим соответствующее ему `do`. Циклы с предусловием встречаются гораздо чаще циклов с постусловием, так что при чтении программы мы подсознательно ожидаем именно обычный `while`. После этого мы примем `while` за заголовок цикла, дальше, возможно, не заметим точку с запятой, а возможно, наоборот, заметим, решим, что это ошибка, «споткнёмся», после чего, просматривая фрагмент уже более внимательным взглядом, найдём, наконец, пресловутое `do`, при этом потратив не меньше секунды на размышления в неправильном направлении и ещё секунду-другую на то, чтобы вернуться к тем

[59] Некоторые программисты даже пребывают в уверенности, что это требование языка Си; на самом деле это не так, телом `do-while` может быть любой оператор, не только составной; но эта возможность Си никогда не используется.

мыслям, из которых нас выбил проклятый `while`. Можно определённо сказать, что такие `while`'ы обходятся читателю программы слишком дорого. Поэтому, сколь бы противно сие ни выглядело, мы возьмём на себя смелость рекомендовать что-то вроде следующего:

```
do
  {
    get_event(&event);
    res = handle_event(&event);
  } while (res != EV_QUIT_NOW);
```

Одной этой сложности может быть вполне достаточно, чтобы отказаться от такого стиля расстановки операторных скобок, по крайней мере, для языка Си (для Паскаля этот аргумент не действует, там такой проблемы нет).

Рассмотрим теперь **характерные для Си ошибки в оформлении функции**. Достаточно часто можно видеть программы, в которых описания локальных переменных в функциях, а также заключительный оператор `return` почему-то не сдвинуты на размер отступа, а написаны с крайней левой позиции экрана, примерно так:

```
int main()
{
int i;
const char *hello = "Hello";
const char *goodbye = "Good Bye";
    for (i = 0; i < 10; i++) {
        printf("%s\n", hello);
    }
    for (i = 0; i < 10; i++) {
        printf("%s\n", goodbye);
    }
return 0;
}
```

Такой стиль, разумеется, недопустим. Прежде всего, оператор `return` — это обычный оператор, встречаться он может не только в самом конце функции, но и в её середине, будучи при этом вложенным в циклы и ветвления; нет никаких оснований считать, что случай возврата значения в последней строчке функции чем-то принципиально отличается от других ситуаций, когда `return` встречается в коде.

Что касается переменных, то, конечно, их описания *операторами* не являются[60], но при таком стиле форматирования фигурная скобка, открывающая функцию, начинает сливаться с окружающим кодом;

[60] Речь идёт о чистом Си; в языке Си++ описание переменной является оператором.

совершенно неясно, какого преимущества хотят добиться сторонники такого странного форматирования.

4.12.2. Последовательности взаимоисключающих if'ов

Как в Си, так и в Паскале оператор выбора имеет очень важное ограничение: условием перехода на одну из меток является *равенство* селектирующего выражения одной из *констант*, причём как константы, так и выражение обязаны иметь порядковый тип. Часто возникает ситуация, при которой необходимо сделать выбор одной из нескольких возможных ветвей работы, основываясь на более сложных условиях или на выражении выбора, имеющем непорядковый тип (например, выбор по значению *строки*). Такой выбор приходится реализовывать с помощью длинной цепочки операторов ветвления: `if` // `else if` // `else if` // ... // `else`. Если *буквально* следовать правилам оформления вложенных операторов, тела ветвей такой конструкции выбора придётся сдвигать всё дальше и дальше вправо, примерно так:

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else
    if (0 == strcmp(cmd, "Load")) {
        printf("Loading...\n");
        load_file();
    } else
        if (0 == strcmp(cmd, "Quit")) {
            printf("Good bye...\n");
            quit_program();
        } else {
            printf("Unknown command\n");
        }
```

Несложно видеть, что при таком подходе окажется достаточно семи-восьми ветвей, чтобы горизонтальное пространство экрана кончилось; между тем ветвей может потребоваться гораздо больше. Что ещё важнее, такой (формально абсолютно правильный) стиль форматирования вводит читателя программы в заблуждение относительно соотношения между ветвями конструкции. Ясно, что эти ветви *имеют одинаковый ранг вложенности*[61]. Но при этом сдвинуты они на *разные* позиции!

[61] Если сомневаетесь, попробуйте поменять ветви местами. Очевидно, что работа программы при этом никак не изменится. А раз так — значит, предположение, что, например, первая из ветвей «главнее» второй, а вторая «главнее» третьей, оказывается неверно.

Пояснить возникающую проблему можно и другим способом. Ясно, что такая цепочка `if`'ов представляет собой *обобщение оператора выбора* и служит тем же целям, что и оператор выбора; разница лишь в выразительной мощности. Но ветви оператора выбора пишутся на одном уровне вложенности. Следовательно, вполне логично считать, что и ветви такой конструкции из `if`'ов должны располагаться на одном уровне отступа.

Достигается это рассмотрением стоящих рядом ключевых слов `else` и `if` как единого целого. Вне зависимости от избранного стиля, вы можете написать `else if` на одной строке через пробел, либо разнести их на разные строки, начинающиеся в одной и той же позиции. В частности, если вы не сносите открывающую операторную скобку на отдельную строку, то вышеприведённые фрагменты вы можете оформить вот так (`if` каждый раз с новой строки):

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else
if (0 == strcmp(cmd, "Load")) {
    printf("Loading...\n");
    load_file();
} else
if (0 == strcmp(cmd, "Quit")) {
    printf("Good bye...\n");
    quit_program();
} else {
    printf("Unknown command\n");
}
```

либо вот так (`if` на одной строке с предыдущим `else`; это тоже допустимо):

```
if (0 == strcmp(cmd, "Save")) {
    printf("Saving...\n");
    save_file();
} else if (0 == strcmp(cmd, "Load")) {
    printf("Loading...\n");
    load_file();
} else if (0 == strcmp(cmd, "Quit")) {
    printf("Good bye...\n");
    quit_program();
} else {
    printf("Unknown command\n");
}
```

Адаптировать сказанное к случаям, когда открывающая скобка сносится на следующую строку, мы предлагаем читателю самостоятельно;

отметим, что вариант с `else if` на одной строке при этом выглядит несколько странно, но всё равно остаётся допустимым.

Подчеркнём, что всё сказанное в этом параграфе относится только к случаю, когда *ветка* `else` *состоит ровно из одного оператора* `if`. Если это не так, следует применять обычные правила форматирования оператора ветвления.

4.12.3. О роли ASCII-набора и английского языка

Мы уже несколько раз упоминали о необходимости использования английского языка в комментариях и при выборе имён, но делали это обычно в сносках, не вдаваясь в подробности. Попробуем дать развёрнутое объяснение этому требованию.

Создавая текст программы, следует учитывать, что в мире существуют самые разные операционные системы и среды, программисты используют несколько десятков (если не сотен) редакторов текстов на любой вкус, а также всевозможные визуализаторы кода, функции которых могут сильно различаться. Далеко не все программисты говорят по-русски[62]; кроме того, для символов кириллицы существуют различные кодировки. Наконец, от одного рабочего места к другому могут существенно различаться размеры экрана и используемых шрифтов.

С чтением правильно оформленной программы не должно возникнуть проблем, кто бы ни читал вашу программу и какую бы операционную среду он при этом ни использовал. Этого можно добиться, вооружившись всего тремя простыми правилами: символы из набора ASCII доступны всегда, английский язык знают все, а экран не бывает меньше, чем 24x80 знакомест, но *больше* он быть не обязан. В части, посвящённой Паскалю, мы потратили целый параграф (см. т. 1, §2.15.14) на подробное объяснение причин именно такого ограничения на ширину текста; повторять эти рассуждения особого смысла нет, так что если жёсткое требование не превышать длину строк программы в 80 символов вызывает у вас сомнения, просто перечитайте тот параграф.

Ещё раньше, во вводной части, в §1.6.5 мы рассказали о достоинствах кодировки ASCII и причинах её универсальности. Если использовать в тексте программы только символы из набора ASCII, можно быть уверенным, что этот текст успешно прочитается на любом компьютере мира, в любой операционной системе, с помощью любой программы, предназначенной для работы с текстом, и так далее. Напомним, что в этот набор входят:

[62] Предположения со словом «никогда» делать вообще вредно; это относится и к предположению, что вашу программу «никогда» не станут читать программисты из других стран.

- заглавные и строчные буквы **латинского** алфавита без диакритических знаков: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`, `abcdefghijklmnopqrstuvwxyz`;
- арабские цифры: `0123456789`;
- знаки арифметических действий, скобки и знаки препинания: `.,;:'"‘-?!@#$%^&()[]{}<>=+-*/~\|`;
- знак подчёркивания `_`;
- пробельные символы — пробел, табуляция и перевод строки.

Никакие другие символы в этот набор не входят. В ASCII не нашлось места для символов национальных алфавитов, включая русскую кириллицу, а также для латинских букв с диакритическими знаками, таких как Š или Ä. Нет в этом наборе многих привычных нам типографских символов, таких как длинное тире, кавычки-«ёлочки», и многих других. **Символы, не входящие в набор ASCII, в тексте программ использовать нельзя** — даже в комментариях, не говоря уже о строковых константах и тем более об идентификаторах. Отметим, что большинство языков программирования не позволит использовать что попало в идентификаторах, но есть и такие трансляторы, которые считают символы, не входящие в ASCII-таблицу, допустимыми в идентификаторах — примером может служить большинство интерпретаторов Лиспа. Ну а на содержимое строковых констант и комментариев большинство трансляторов вообще не обращает внимания, позволяя вставить туда практически что угодно. И тем не менее, попустительство трансляторов не должно сбивать нас с толку: текст программы на любом языке программирования обязан состоять из ASCII-символов и только из них. Любой символ, не входящий в ASCII, может превратиться во что-то совершенно иное при переносе текста на другой компьютер, может просто не прочитаться и т. д.

Остаётся вопрос, как быть, если программа, которую вы пишете, должна общаться с пользователем по-русски. Ответ мы дадим чуть позже; пока же отметим, что ограничение на используемый набор символов — не единственная причина, по которой в программах не допускается использование русского языка. Скажем, набор букв, входящих в ASCII, достаточен для представления текста на итальянском, а с использованием замены отдельных букв диграфами — и на немецком языке, но ни итальянский, ни немецкий в тексте программы применять нельзя точно так же, как и русский. Есть лишь один язык, который допустим в тексте компьютерной программы, и этот язык — английский. Для программистов **английский язык — не роскошь, а средство взаимопонимания**. По сложившейся традиции программисты всего мира используют именно английский язык для общения между собой[63].

[63] Оставим в стороне вопрос о том, хорошо это или плохо, и ограничимся констатацией факта. Отметим, впрочем, что у врачей и фармацевтов всего мира есть традиция заполнять рецепты и некоторые другие медицинские документы на ла-

Как правило, можно предполагать, что любой человек, работающий с текстами компьютерных программ, поймёт хотя бы не очень сложный текст на английском языке, ведь именно на этом языке написана документация к разнообразным библиотекам, стандарты, описания сетевых протоколов, издано множество книг по программированию; конечно, многие книги и другие тексты переведены на русский (а равно и на французский, японский, венгерский, хинди и прочие национальные языки), но было бы неразумно ожидать, что *любой* нужный вам текст окажется доступен на русском — тогда как на английском доступна практически любая программистская информация.

Из этого вытекают три важных требования. Во-первых, **любые идентификаторы в программе должны состоять из английских слов или быть аббревиатурами английских слов**. Если вы забыли нужное слово, не поленитесь заглянуть в словарь. Подчеркнём, что слова должны быть именно английские — не немецкие, не французские, не латинские и тем более не русские «транслитом» (последнее вообще считается у профессионалов крайне дурным тоном). Во-вторых, **комментарии в программе должны быть написаны по-английски**; как уже говорилось, лучше вообще не писать комментарии, нежели пытаться писать их на языке, отличном от английского. И, наконец, **пользовательский интерфейс программы должен быть либо англоязычным, либо «международным» (то есть допускающим перевод на любой язык)**; этот момент мы подробно рассмотрим в следующем параграфе.

К настоящему моменту у некоторых читателей мог возникнуть закономерный вопрос: «А что делать, если я не знаю английского». Ответ будет тривиален: срочно учить. Программист, не умеющий более-менее грамотно писать по-английски (и тем более не понимающий английского), в современных условиях профессионально непригоден, сколь бы неприятно это ни звучало.

4.12.4. (*) Программы, говорящие по-русски

Пункт о языке пользовательского интерфейса нуждается в дополнительном комментарии, который послужит ответом на вопрос, как быть, если программа должна общаться с пользователем по-русски (или по-немецки, или по-китайски — это не важно).

Изоляционизм в программировании, к счастью, ушёл в далекое прошлое, и в наши дни над одной программой могут работать программисты из нескольких десятков разных стран, а пользоваться одной и той же программой могут пользователи всего мира — во всяком случае, всех частей мира, где есть компьютеры. В такой обстановке постоянно возникает ситуация, когда программу необходимо

тинском языке, а, например, официальным языком Всемирного почтового союза является французский; так или иначе, существование единого профессионального языка общения оказывается во многом полезно.

«научить» общаться с конечным пользователем на языке, которого не знает никто из её авторов, причём эта ситуация в наше время представляет собой скорее правило, нежели исключение. Адаптация программы к использованию другого языка оказывается достаточно простой, если её автор следовал определённым соглашениям; общая идея тут такова, что исходные строки, которые должен увидеть (или ввести) пользователь, прямо во время работы программы заменяются строками, написанными на другом языке, которые загружаются из внешнего файла (то есть *не являются частью программы*).

Обычно для этой цели используются готовые библиотеки, специально предназначенные для «интернационализации» сообщений в программах. В ОС Unix наиболее популярна библиотека `gettext`. При работе на языке Си, чтобы сделать возможной загрузку строк во время исполнения, все строки в программе традиционно обрамляются знаком подчёркивания и скобками — например, вместо

```
printf("Hello, world\n");
```

пишут

```
printf(_("Hello, world\n"));
```

Макрос с именем «`_`» разворачивается в вызов функции `gettext`, которая пытается найти файл с переводом сообщений на используемый язык интерфейса, а в нём, в свою очередь — перевод для строки `"Hello, world\n"`, причём сама эта строка используется как ключ для поиска. Если не удалось найти такой перевод (или сам файл с переводами), в качестве результата возвращается аргумент, то есть если `gettext` не знает, как перевести строку `"Hello, world\n"` на нужный язык, она оставит эту строку нетронутой.

Если вы по каким-то причинам не хотите использовать `gettext`, достаточно переопределить макрос `_`, чтобы он всегда возвращал свой аргумент; переделывать программу при этом не придётся.

Больше того, если даже вы не сочли нужным подготовить свою программу к использованию с библиотекой `gettext`, это может сделать другой программист, просто заключив все строковые константы в макровызов `_()`, что достаточно просто — при известной ловкости это можно сделать одной командой в текстовом редакторе. Есть, однако, одно крайне важное условие, выполнение которого необходимо для превращения моноязычной программы в «международную»: все сообщения в её тексте должны быть английскими. Двойной перевод системами интернационализации не предусмотрен, ну а переводчиков с *русского* на другие языки, будь то китайский или немецкий, найти гораздо сложнее, чем переводчиков с *английского* — особенно если учесть, что речь идёт не о профессиональных переводчиках, а, как правило, о программистах, которым пришло в голову перевести сообщения очередной программы на свой родной язык; английский знают программисты во всём мире, но вот найти нерусского программиста, при этом знающего русский, будет посложнее.

Таким образом, программа, даже не адаптированная исходно под нужды многоязычности, вполне имеет шансы стать когда-нибудь «международной» — но только если исходно все сообщения в её тексте английские. Из этого очевидным образом следует сформулированное выше утверждение: ваша программа должна быть либо «международной», либо англоязычной. Если вы не чувствуете себя

в силах или не имеете желания учитывать возможный перевод интерфейса программы на другие языки, по крайней мере не лишайте *других* программистов возможности сделать это за вас. Если же русскоязычность является требованием, не поленитесь сделать всё *правильно* — то есть в соответствии с требованиями «международности».

На случай, если вам потребуется срочно сделать русскоговорящую программу, мы приведём простой пример работы с `gettext` в надежде, что вы не станете ради сиюминутной потребности нарушать фундаментальные требования к оформлению исходных текстов (а наличие русских букв в тексте программы — это именно фундаментальное нарушение).

Начнём с простого текста программы, печатающей два сообщения:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    printf("Good bye, world!\n");
    return 0;
}
```

Эта программа, понятное дело, никоим образом не может считаться *интернационализированной*, но у неё есть все шансы такой стать, поскольку выполнено главное условие: все сообщения в программе написаны по-английски. Первый шаг в направлении интернационализации состоит в обрамлении всех сообщений вызовом макроса `_()`; сам макрос мы введём в начале программы самым простым из возможных способов:

```
#include <stdio.h>

#define _(STR) (STR)

int main()
{
    printf(_("Hello, world!\n"));
    printf(_("Good bye, world!\n"));
    return 0;
}
```

Как можно догадаться, эта программа делает то же самое, что и предыдущая, то есть по сути программа не изменилась; просто она стала «чуть более подготовленной» к превращению в «международную». Здесь есть ещё один момент: иногда в программах встречаются строки, которые либо вообще не нужно переводить, либо они будут переводиться не в том месте, где встречены. Например, если строка расположена в инициализаторе какого-нибудь массива строк, вызывать функцию `gettext` из такого инициализатора, пожалуй, всё же не стоит; возможны и другие ситуации. Строки, не подлежащие переводу, можно просто оставить как есть, но у читателя программы это может вызвать подозрение, что

про данную конкретную строку просто забыли. Поэтому рекомендуется ввести ещё один макрос:

```
#define N_(STR) (STR)
```

и в программе *все* строки оформить с вызовом одного из макросов: те, строки, что должны будут переводиться, заключить в макровызов `_()`, а те, что переводить не нужно — в макровызов `N_()`. Это можно сделать сразу же, даже если вы пока не собираетесь превращать вашу программу в международную. Когда дело дойдёт до непосредственного задействования `gettext`, вы сэкономите время, поскольку все строковые литералы в программе уже будут подготовлены к переводу.

Когда вам потребуется превратить программу в международную, придётся сделать несколько модификаций её исходного текста. Во-первых, нужно будет добавить включение заголовочных файлов `locale.h` ради функции `setlocale` (к сожалению, библиотека `gettext` намертво завязана на использование механизма так называемых системных *локалей* и без настройки локали работать не будет) и `libintl.h`, в котором объявлены функция `gettext` и вспомогательные `bindtextdomain` и `textdomain`, которые нам тоже потребуются.

Библиотека `gettext` требует первичной настройки, вся цель которой, собственно говоря, в том, чтобы сообщить, из какого файла брать переводы строк. Как это, к сожалению, часто водится в проектах группы GNU, да и вообще во многих современных проектах, просто указать имя файла библиотека не позволяет. Вместо этого требуется задать некую *базовую директорию*, где хранятся данные, имеющие отношение к локалям, и совсем непонятный *текстовый домен*, который при ближайшем рассмотрении оказывается просто именем нашей программы (или пакета программ, к которому она относится — но это не наш случай).

В качестве имени программы (пресловутого «текстового домена») мы воспользуемся словом «`hellobye`», поскольку именно так называется наша программа. С базовой директорией всё несколько сложнее. Если бы мы собирались инсталлировать нашу программу в системные директории, в этой роли выступала бы директория `/usr/share/locale` или `/usr/local/share/locale` (в зависимости от того, каким способом производится установка). Поскольку инсталляция программы в системные директории вряд ли входит в наши планы, мы поступим проще: укажем в качестве «базовой» текущую директорию (попросту строку `"."`). Это позволит запустить наш простенький пример, но для более сложных задач может оказаться непригодно, поскольку программы чаще всего запускаются не из той директории, в которой находятся принадлежащие им файлы; в настоящей «боевой» программе, возможно, имя базовой директории локализационных данных придётся получить через параметры командной строки, прочитать из конфигурационного файла или каким-то образом синтезировать из имени домашней директории пользователя. Так или иначе, нам для демонстрации работы `gettext` вполне подойдёт текущая директория.

Имя базовой директории и текстового домена вынесем в макроконстанты в начале программы:

```
#define LOCALEBASEDIR "."
#define TEXTDOMAIN "hellobye"
```

Туда же добавим новые определения для макросов `_()` и `N_()` (в нашем примере второй из них не будет задействован, но лучше сразу добавлять оба, чтобы, когда нам встретится строка, не подлежащая переводу, у нас не возникало соблазна оставить её как есть):

```
#define _(STR) gettext(STR)
#define N_(STR) (STR)
```

В начало функции `main` вставим настройку локали и самой библиотеки `gettext`:

```
    setlocale(LC_CTYPE, "");
    setlocale(LC_MESSAGES, "");
    bindtextdomain(TEXTDOMAIN, LOCALEBASEDIR);
    textdomain(TEXTDOMAIN);
```

Окончательный текст программы примет следующий вид:

```
/* gettext/hellobye.c */
#include <stdio.h>
#include <locale.h>
#include <libintl.h>

#define LOCALEBASEDIR "."
#define TEXTDOMAIN "hellobye"

#define _(STR) gettext(STR)
#define N_(STR) (STR)

int main()
{
    setlocale(LC_CTYPE, "");
    setlocale(LC_MESSAGES, "");
    bindtextdomain(TEXTDOMAIN, LOCALEBASEDIR);
    textdomain(TEXTDOMAIN);
    printf(_("Hello, world!\n"));
    printf(_("Good bye, world!\n"));
    return 0;
}
```

Подготовив исходник, приступим к созданию его «переводов», то есть файлов, задающих для английских сообщений их переводы на другие языки. Для начала нам потребуется запустить программу `xgettext`, которая автоматически извлечёт из нашего текста программы все строки, заключённые в макровызов `_()`, и на их основе создаст шаблонный файл, который можно будет отредактировать, создав файл перевода. Запускаем команду:

```
xgettext --keyword="_" hellobye.c -o hellobye.pot
```

Здесь параметр `--keyword` задаёт имя того вызова (в нашем случае — макровызова), которым помечены в тексте программы строки для перевода, ключ `-o` указывает имя файла для записи результата. Программа `xgettext` создаст в рабочей директории файл с именем `hellobye.pot` (суффикс `.pot` означает *portable object template*, то есть «шаблон переносимого объекта») примерно с таким содержимым:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-03-21 14:19+0300\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: hellobye.c:17
#, c-format
msgid "Hello, world!\n"
msgstr ""

#: hellobye.c:18
#, c-format
msgid "Good bye, world!\n"
msgstr ""
```

Этот шаблон отражает то, как, по убеждениям авторов gettext, должен выглядеть файл перевода. Скопируем этот файл под именем ru.po (ru здесь означает русский язык, po — *portable object*) и отредактируем. Прежде всего модифицируем комментарий в начале файла, указав, что́ это за файл и какие на него накладываются лицензионные ограничения (в нашем примере таких нет, что мы и напишем). Затем в строке Project-Id-Version укажем имя и версию нашей программы. В строке Report-Msgid-Bugs-To следует указать, к кому обращаться, если обнаружена ошибка в оригинальных сообщениях (не в переводе). Строку PO-Revision-Date следует оставить как есть, с ней потом разберётся программа, создающая на основе нашего файла перевода индексированный файл, в котором поиск сообщений происходит быстрее. В строке Last-Translator надо указать своё имя или псевдоним, а вот адрес электронной почты лучше не указывать, сколь бы мы ни уважали мнение группы GNU на эту тему: вряд ли вы будете рады увеличению количества поступающего вам спама. Строку Language-Team мы просто выкинем, поскольку не входим ни в какую команду по переводу; в строке Content-Type заменим слово CHARSET на используемую кодировку (в нашем примере это будет KOI8-R, но в вашей системе, скорее всего, используется UTF-8). Строки MIME-Version и Content-Transfer-Encoding оставим как есть.

Наконец, сделаем самое главное: вставим русский перевод для обеих имеющихся текстовых строк. Результат будет выглядеть примерно так:

```
# gettext/ru.po
# Russian translation for the HelloBye program
# No copyright is claimed!
```

```
# Andrey V. Stolyarov, 2016.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: HelloBye 1.0\n"
"POT-Creation-Date: 2016-03-21 14:19+0300\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: Andrey V. Stolyarov\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=KOI8-R\n"
"Content-Transfer-Encoding: 8bit\n"

#: hellobye.c:17
#, c-format
msgid "Hello, world!\n"
msgstr "Здравствуй, мир!\n"

#: hellobye.c:18
#, c-format
msgid "Good bye, world!\n"
msgstr "До свиданья, мир!\n"
```

Теперь этот файл следует откомпилировать, получив собственно тот (бинарный) файл, который gettext будет использовать в работе. От своего исходника результат такой компиляции отличается возможностью быстрого поиска нужного сообщения. Компиляцию произведёт программа msgfmt:

```
msgfmt ru.po -o ru.mo
```

Результатом станет файл ru.mo (суффикс здесь означает *machine object*). Для наглядности снабдим нашу программу переводом ещё и на немецкий, для чего создадим копию ru.po под именем de.po и отредактируем, заменив кодировку на ASCII, а русские фразы — на соответствующие немецкие:

```
# German translation for the HelloBye program
# No copyright is claimed!
# Andrey V. Stolyarov, 2016.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: HelloBye 1.0\n"
"POT-Creation-Date: 2016-03-21 14:19+0300\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: Andrey V. Solyarov\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ASCII\n"
"Content-Transfer-Encoding: 8bit\n"

#: hellobye.c:17
#, c-format
msgid "Hello, world!\n"
msgstr "Hallo Welt!\n"
```

```
#: hellobye.c:18
#, c-format
msgid "Good bye, world!\n"
msgstr "Auf Wiedersehen Welt!\n"
```

Запустив `msgfmt`, получим файл `de.mo`. Теперь эти два файла с суффиксом `.mo` нужно разместить так, чтобы привередливая `gettext` их нашла во время работы нашей программы:

```
mkdir -p ru/LC_MESSAGES
cp ru.mo ru/LC_MESSAGES/hellobye.mo
mkdir -p de/LC_MESSAGES
cp de.mo de/LC_MESSAGES/hellobye.mo
```

Поясним, что полное имя файла складывается из четырёх компонентов. Первый — обсуждавшаяся выше *базовая директория*, но в этой роли мы задали текущую директорию, что несколько облегчает дело. Второй компонент — идентификатор языка, в нашем случае `ru` для русского и `de` для немецкого. Третий компонент задаёт *подсистему локали*, в нашем случае это `LC_MESSAGES`; последний — это собственно имя файла, название которого должно совпадать с *текстовым доменом* (мы использовали `hellobye`), снабжённым суффиксом `.mo`, отражающим формат файла. Судя по всему, мы готовы. Компилируем программу как обычно (в системах, отличных от Linux, может потребоваться явное подключение библиотеки `intl`, то есть флажок `-lintl`, но в Linux это не нужно):

```
avst@host:~/gettext$ gcc -Wall -g hellobye.c -o hellobye
```

И пробуем запустить. Если в переменных окружения задана русская локализация, программа выдаст нам сообщения, заданные русским переводом:

```
avst@host:~/gettext$ ./hellobye
Здравствуй, мир!
До свиданья, мир!
avst@host:~/gettext$
```

Язык, на котором программа разговаривает, можно оперативно менять, задавая соответствующие значения для переменной окружения `LANGUAGE`:

```
avst@host:~/gettext$ LANGUAGE=en ./hellobye
Hello, world!
Good bye, world!
avst@host:~/gettext$ LANGUAGE=de ./hellobye
Hallo Welt
Auf Wiedersehen Welt!
avst@host:~/gettext$ LANGUAGE=ru ./hellobye
Здравствуй, мир!
До свиданья, мир!
avst@host:~/gettext$
```

Если соответствующий язык не найден или локаль настроена неправильно, скорее всего, программа выдаст английские сообщения.

Вскоре у вас наверняка возникнет вопрос, что делать, если в программе появились новые сообщения — не переводить же всё с нуля. Для таких случаев создатели gettext предусмотрели программу msgmerge, которая позволяет построить новый .po-файл по имеющемуся (устаревшему) .po-файлу и обновлённому .pot-файлу. Попробуем для примера вставить в нашу программу ещё один вызов printf между двумя уже имеющимися:

```
printf(_("We are speaking\n"));
```

Теперь построим обновлённый файл hellobye.pot с помощью xgettext точно так же, как мы делали это в прошлый раз, после чего обновим имеющиеся у нас файлы перевода с помощью msgmerge. Эту программу можно запускать в разных режимах; один из удобных вариантов выглядит так:

```
msgmerge -U --backup=numbered ru.po hellobye.pot
msgmerge -U --backup=numbered de.po hellobye.pot
```

При таких параметрах msgmerge запишет результаты сразу в файлы ru.po и de.po, а старые версии на всякий случай сохранит под именами ru.po.~1~ и de.po.~1~. Новые версии будут отличаться от старых датой/временем в строке POT-Creation-Date, номерами строк в комментариях к ранее переведённым сообщениям (сами переводы никуда не денутся) и наличием новых (пока ещё не переведённых) сообщений. Остаётся вписать туда переводы новых строк (в нашем примере — строки "We are speaking"), запустить для каждого из файлов программу msgfmt и скопировать полученные файлы с суффиксом .mo куда следует — точно так же, как мы делали это раньше.

Как обычно в таких случаях, приходится признать, что мы рассмотрели только самые тривиальные возможности библиотеки gettext; за более подробной информацией следует обратиться к документации, размещённой на официальной web-странице проекта gettext[64]. Сразу же хотелось бы предостеречь читателя от одной ошибки. В Интернете можно найти достаточно много текстов, в которых рассказывается, как работать с gettext через пакеты autoconf/autotools, причём в некоторых случаях авторы таких текстов безапелляционно утверждают, что последние для работы с gettext *необходимы*. К счастью, это не так; в приведённых выше примерах мы никак не задействовали autotools, не требует их задействования и текст официальной документации. Что касается самих этих пакетов, то они относятся к числу таких инструментов, которые не следует применять ни для чего, никогда и ни за что — ну разве что за очень большие деньги. Никаких проблем они в действительности не решают, что бы вам об этом ни говорили, зато создают проблемы на ровном месте, что называется, пачками.

Отметим один довольно важный идейный момент. **Библиотека** gettext **— это далеко не единственное средство создания «международных» программ.** Нет ничего плохого даже в том, чтобы сделать программу многоязычной «вручную», вообще без использования готовых библиотек. Серьёзное ограничение

[64] http://www.gnu.org/software/gettext/

только одно: все сообщения на языках, отличных от английского, должны находиться в отдельных файлах, которые читаются уже во время работы вашей программы.

4.13. Ещё об указателях

В этой главе мы введём такие типы указателей, «благодаря» которым обыкновенный заголовок функции или описание переменной может превратиться в нелепое и совершенно нечитаемое нагромождение скобочек и звёздочек. Прежде чем это будет сделано, отметим, что директива `typedef` (см. §4.9.7) всегда позволяет обойтись без таких кошмарных конструкций и, соответственно, её использование настоятельно рекомендуется во всех случаях, когда описание кажется вам сложным.

4.13.1. Многомерные массивы и указатели на массивы

С многомерными массивами мы до сих пор не сталкивались, но в Си они, в принципе, поддерживаются — во всяком случае, в том же смысле, в каком поддерживаются массивы одномерные. Например, матрицу целых чисел из трёх строк по четыре столбца в каждой можно ввести так:

```
int m[3][4];
```

Это описание можно прочитать следующим образом: *`m` есть массив из трёх массивов из четырёх элементов типа `int`*. В памяти массив располагается построчно; в данном случае сначала идут четыре элемента первой строки (с `m[0][0]` по `m[0][3]`), затем четыре элемента второй строки (`m[1][0]` ... `m[1][3]`), последними — четыре элемента третьей строки (см. рис. 4.3).

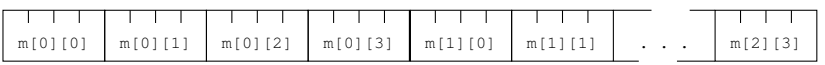

Рис. 4.3. Расположение в памяти массива `int m[3][4]`

Обращение к элементам многомерного массива производится через несколько (по количеству измерений, в данном случае два) применений операции индексирования, например `m[2][1]`. **Каждый индекс записывается в отдельных квадратных скобках**. Перечислять индексы через запятую, как в Паскале, нельзя; оно и понятно, ведь квадратные скобки представляют собой, как мы видели, обыкновенную арифметическую операцию от двух аргументов, одним из которых должен быть

адрес. Например, следующий фрагмент заполнит наш массив значениями, полученными как произведение индексов (в частности, `m[0][2]` получит значение `0`, `m[2][3]` — значение `6`):

```
for(i = 0; i < 3; i++)
    for(j = 0; j < 4; j++)
        m[i][j] = i * j;
```

Описание многомерного массива можно снабдить инициализатором, например:

```
int m[3][4] = {{0, 0, 0, 0}, {0, 1, 2, 3}, {0, 2, 4, 6}};
```

если инициализатор не помещается в одной строке, его обычно «раскидывают» на разные строки в соответствии с содержимым строк массива:

```
const int level_change[5][5] = {
    { 4, 4, 2, 1, 1 },
    { 3, 4, 3, 1, 1 },
    { 1, 3, 4, 3, 1 },
    { 1, 1, 3, 4, 3 },
    { 1, 1, 2, 4, 4 }
};
```

Теперь мы плавно подбираемся к самому интересному аспекту многомерных массивов. Если ваши мозги категорически откажутся воспринять остаток параграфа, не паникуйте: всего этого порой не знают некоторые профессиональные программисты, пишущие на Си за деньги. Впрочем, даже если с ходу вы ничего не поймёте, постарайтесь в будущем вернуться к этому параграфу и всё-таки постичь всю эту механику, которую можно считать одним из самых любопытных изобретений в области семантики языков программирования.

Итак, вернёмся к нашему массиву `int m[3][4]` и зададим себе незатейливый, на первый взгляд, вопрос: а что собой представляет имя `m`?

Напомним, что по правилам языка Си во всех случаях, кроме применения `sizeof` и ещё двух экзотических ситуаций, про которые мы даже не стали говорить, имя массива есть адрес его начала. Кроме того, заметим, что выражение `m[1]` должно быть таково, чтобы к нему можно было применять вторую операцию индексирования, результатом которой становились бы элементы второй строки матрицы. Следовательно, выражение `m[1]` должно удовлетворять двум свойствам: это, во-первых, должен быть адрес начала второй строки, и, во-вторых, применение индексирования к этому выражению должно давать, собственно говоря, сами элементы второй строки. Иначе говоря, `m[1]` должно представлять собой адрес типа `int*`, указывающий на начало второй строки матрицы.

Вспомним теперь, что `m[1]` есть не что иное, как сокращённая запись выражения `*(m+1)`. Получается, что:

- имя `m` представляет собой адрес начала матрицы;
- прибавление единицы к `m` означает **численное увеличение указателя на размер строки матрицы**;
- разыменование выражений `m`, `m+1`, `m+2` даёт значение типа `int*`.

Ни один из ранее рассмотренных нами типов не обладает всеми этими свойствами одновременно. В самом деле, единственный известный нам тип, такой, что для переменной `p` такого типа выражение `*p` будет иметь тип `int*` — это, как несложно догадаться, тип `int**`, но он нам здесь явно не подходит: прибавление к нему единицы сдвинет нас по адресному пространству на размер указателя (4 или 8 байт), а не на размер строки матрицы (в нашем примере 16 байт, но может быть сколько угодно).

Создатели языка Си решили этот вопрос, введя особенный вид типов — ***указатель на массив*** (или, строго говоря, *адрес массива*, ведь указатель — это переменная, хранящая адрес, а не сам адрес), причём это совершенно не то же самое, что хорошо знакомый нам *указатель на первый элемент массива*. Особенную пикантность этому придаёт то обстоятельство, что **типа «массив» в языке Си нет**, во всяком случае, полноценного — а вот указатели на него, как видим, есть. Для нашего примера — массива `int m[3][4]` — нужным типом будет «адрес массива из четырёх элементов типа `int`» в соответствии с параметрами строки массива. Переменная этого типа описывается так:

```
int (*p)[4];
```

(читается как «p есть указатель на массив из четырёх элементов типа `int`»). Теперь можно, например, сделать присваивание `p = m` и работать с этим `p` точно так же, как мы работали с `m`, в том числе применяя двойное индексирование. Такое описание ни в коем случае не следует путать с похожим на него

```
int *q[4];
```

Здесь `q` — это обыкновенный массив из четырёх указателей на `int`, то есть массив из четырёх элементов типа `int*`; с указателями на массивы это не имеет ничего общего.

Аналогичным образом для трёхмерного массива

```
int z[10][15][20];
```

нам потребовался бы *указатель на двумерный массив*:

```
int (*zptr)[15][20];
```

Прибавление единицы к адресу такого типа перемещает нас в адресном пространстве сразу на $15 \times 20 \times 4 = 1200$ ячеек — ровно на размер «среза» массива, представляющего собой матрицу 15x20 элементов типа `int`.

4.13.2. Указатели на функции

Откомпилированные подпрограммы, в том числе функции языка Си — это, в сущности, фрагменты машинного кода; ясно, что этот код хранится где-то в оперативной памяти, то есть *занимает область памяти* и, как следствие, можно говорить об ***адресе функции***, который равен адресу первой из ячеек, занятых этим кодом. Адрес, понятное дело, можно где-то хранить; так появляется ***указатель на функцию***. Язык Си позволяет вызывать функцию, используя её адрес.

Для примера рассмотрим несколько функций, принимающих на вход массив элементов типа `double` (в виде адреса первого элемента и целого числа, обозначающего длину массива) и возвращающих число типа `double`. Самая простая из них будет находить сумму элементов массива:

```
double dbl_sum(const double *a, int size)
{
    return size > 0 ? *a + dbl_sum(a+1, size-1) : 0;
}
```

Вторая функция будет отыскивать минимальное число в массиве:

```
double dbl_min(const double *a, int size)
{
    double d;
    if(size == 1)
        return *a;
    d = dbl_min(a+1, size-1);
    return *a < d ? *a : d;
}
```

Третью мы заставим посчитать среднее арифметическое:

```
double dbl_average(const double *a, int size)
{
    return dbl_sum(a, size) / (double) size;
}
```

Можно придумать и ещё подобных функций; заметим, что их заголовки будут различаться только именем, тогда как количество и типы параметров, а также тип возвращаемого значения у них одинаковый. Говорят, что эти функции *имеют одинаковый профиль*. Для нашего рассуждения это важно, поскольку указатели на функции в Си являются типизированными в том смысле, что при их описании необходимо указывать типы параметров функции и тип её возвращаемого значения, то есть всю информацию, входящую в упомянутый *профиль*. В частности, указатель, способный хранить адрес любой из трёх перечисленных функций, описывается так:

```
double (*fptr)(const double *, int);
```

Как и в случае с указателями на массивы, здесь очень важны круглые скобки вокруг `*fptr`; если бы их не было, мы бы получили что-то вроде

```
double *fn(const double *, int);
```

но это вовсе не описание переменной; это *объявление (заголовок) функции*, принимающей на вход два параметра и возвращающей адрес `double`.

Вернёмся к указателю `fptr`. Ему можно присвоить адрес любой из наших функций. Сделать это можно как с указанием операции взятия адреса, так и без; оба следующих присваивания правомерны:

```
fptr = &dbl_min;
fptr = dbl_min;
```

Вызов функции через указатель (или другое адресное выражение) можно, опять-таки, произвести непосредственно либо после применения операции разыменования. К примеру, если у нас есть массив и переменная:

```
double arr[100];
double res;
```

то оба следующих вызова компилятор сочтёт правильными:

```
res = (*fptr)(arr, sizeof(arr)/sizeof(*arr));
res = fptr(arr, sizeof(arr)/sizeof(*arr));
```

Глядя на эти примеры, можно заключить, что создатели Си так и не решили, существует ли в семантике этого языка *функция как таковая*; иначе говоря, представляет ли имя функции сразу её адрес или же имя представляет *саму функцию*, а её адрес можно получить соответствующей операцией. На самом деле в оригинальном языке Си, предложенном его создателями, функция рассматривалась как самостоятельная сущность; взятие адреса функции (например, для присваивания указателю) должно было осуществляться явным образом с помощью операции «&», а её вызов через указатель (или другое адресное выражение) — с применением операции разыменования («*»). При внимательном взгляде оказалось, что над «функцией как самостоятельной сущностью» нет никаких других операций, кроме её вызова, обозначаемого круглыми скобками, и взятия адреса; попросту говоря, в оригинальном Си имя функции могло встретиться либо со скобками (списком параметров) после него, либо с амперсандом перед ним, и более никак.

Авторы некоторых компиляторов решили, что операции взятия адреса и разыменования по отношению к функциям избыточны, и сделали их необязательными; такая практика была окончательно легитимизирована в стандарте ANSI,

который, судя по весьма сдержанным, но от этого даже более красноречивым комментариям в книге Кернигана и Ритчи, создателям языка не понравился.

Остаётся естественный вопрос: так как же следует писать? Однозначного ответа нет, можно найти достаточно много программистов, делающих и так, и эдак. Программисты «старой закваски», пишущие на чистом Си, часто предпочитают при вызове функции через указатель использовать операцию разыменования этого указателя, чтобы вызов через указатель зрительно отличался от вызова функции по имени. В мире Си++ от этой традиции давно уже ничего не осталось, ведь там специально сделано всё возможное, чтобы введённый пользователем объект можно было использовать «как массив», «как функцию» и т. д.

Чтобы понять, зачем могут потребоваться указатели на функции, вернёмся к примеру, который мы привели на стр. 323: там мы рассматривали функцию, которая удаляет из заданного списка целых чисел все элементы, хранящие отрицательное число. Функция получилась, как можно заметить, нетривиальная, хотя и не очень сложная. Представьте себе теперь, что вам потребовалась функция, удаляющая из такого же списка не отрицательные числа, а, например, чётные или делящиеся без остатка на семь; что же, скопировать функцию под другим именем и, изменив в ней всего одно условие, так и оставить эту копию существовать? А потом, возможно, ещё одну, и ещё, и ещё? Но мы уже знаем, что копирование фрагментов кода — это крайне порочная практика, во многих случаях даже *запрещённая*.

Адреса функций позволяют нам выйти из положения, передав одним из параметров *критерий удаления из списка*. Такой критерий оформляется в виде функции, принимающей на вход число, хранящееся в списке (в данном случае это `int`) и возвращающей «истину», если число следует удалить, и «ложь», если его следует оставить. Например, можно в качестве критериев написать следующие функции:

```
int is_negative(int x) { return x < 0; }
int is_even(int x) { return x % 2 == 0; }
int is_div7(int x) { return x % 7 == 0; }
```

Теперь мы можем переписать функцию `delete_negatives_from_list`, текст которой был приведён на стр. 323, приспособив её для удаления элементов, отвечающих *произвольному* критерию; для этого добавим в её заголовок второй параметр — указатель на функцию, задающую критерий, то есть, попросту, на функцию, которая принимает число типа `int` на вход и возвращает логическое значение (то есть тоже `int`); вместо сравнения с нулём в операторе `if` подставим вызов этой функции от числа из текущего элемента списка:

```
void delete_from_int_list(struct item **pcur, int (*crit)(int))
{
    while(*pcur) {
```

```
            if((*crit)((*pcur)->data)) {
                struct item *tmp = *pcur;
                *pcur = (*pcur)->next;
                free(tmp);
            } else {
                pcur = &(*pcur)->next;
            }
        }
    }
```

Теперь если для работы со списком используется, как и раньше, указатель с именем `first`, то удалить из этого списка все отрицательные числа можно вызовом

```
delete_from_int_list(&first, &is_negative);
```

а удалить все числа, делящиеся без остатка на семь — вызовом

```
delete_from_int_list(&first, &is_div7);
```

В качестве второго примера мы приведём более сложную ситуацию, которая позволит нам проиллюстрировать классический подход к взаимодействию подсистем в программе, называемый *callback function*[65]. Подход состоит в том, что один модуль (пользователь) поручает другому модулю (серверу) отыскать (найти в памяти, загрузить по сети, прочитать из файла, получить из базы данных и т. п.) некие данные; по мере того, как подходящие данные будут находиться, сервер должен вызывать некую функцию пользователя[66], передавая этой функции в качестве параметра найденные данные.

Поскольку сервер, как правило, хочется сделать по возможности универсальным, об устройстве пользовательского модуля он должен знать как можно меньше; поэтому сервер, естественно, не знает[67], какую именно функцию пользователя надо вызывать; ему передают указатель на нужную функцию.

Ещё один нетривиальный момент состоит в том, что для пользователя обработка найденных объектов данных может быть единым процессом, то есть на обработку последующих объектов влияют результаты

[65] На русский язык это буквально переводится как *функция для обратного вызова*, но программисты не используют ни этот, ни какой-либо другой перевод; а применяют английское слово *callback*, которое читается примерно как *колбэк*. Это, конечно, жаргонизм, но, какой бы перевод вы ни пытались использовать, вы рискуете быть непонятыми.

[66] Подчеркнём ещё раз, что под *пользователем* в данном случае понимается не человек, сидящий за компьютером, а тот *модуль нашей программы*, который использует услуги, предоставленные модулем-сервером.

[67] Возможно, программист, пишущий модуль, прекрасно знает, как устроены все модули, которые будут использовать его подсистему, но даже в этом случае он не должен своим знанием пользоваться — возможно, кто-нибудь когда-нибудь допишет другие модули.

обработки предыдущих; для хранения этих результатов пользователь вынужден завести некую память (переменную, массив и т. п.), причём к этой памяти необходимо обеспечить доступ для callback-функции; теоретически это можно сделать через глобальную переменную, но глобальные переменные, во-первых, зло сами по себе, а во-вторых, таким образом мы лишили бы себя возможности запустить два и более процесса обработки одновременно — например, обращаясь к ним по очереди. Но если не использовать глобальные переменные, то остаётся только один способ передачи информации в callback-функцию — через её параметр; при этом для разных пользователей может потребоваться применение совершенно разных данных, и сервер ничего об этом знать, по идее, не должен.

Все проблемы одним махом снимаются путём передачи в callback-функцию нетипизированного указателя, то есть адреса типа `void*`. Соответствующий параметр обычно называется *пользовательскими данными* (англ. *user data*). Заведя у себя нужную структуру данных, пользователь передаёт серверу адрес callback-функции и нетипизированный адрес созданных данных; по мере нахождения искомых объектов сервер вызывает callback-функцию, используя переданный адрес; в качестве параметров сервер передаёт ей, во-первых, очередной найденный объект, и, во-вторых, нетипизированный указатель на пользовательские данные.

За основу нашего примера мы возьмём обход двоичного дерева поиска, реализация которого приведена на стр. 329. Напомним, что мы рассматривали дерево, в котором хранятся целые числа. Функция, которую мы написали, умеет, обходя дерево, делать только одну операцию: печатать числа, хранящиеся в узлах дерева. Мы изменим её так, чтобы она *для каждого найденного в дереве числа вызывала некую «пользовательскую» функцию*, передавая ей, во-первых, найденное число, и, во-вторых, нетипизированный адрес пользовательских данных, который она сама получила в качестве параметра.

Наша callback-функция должна, таким образом, получать два параметра (`int` и `void*`), а возвращать ничего не должна. Следовательно, её профиль будет примерно таким:

```
void callback_function(int num, void *userdata);
```

Сама функция обхода дерева станет при этом такой:

```
void int_bin_tree_traverse(struct node *r,
                           void (*callback)(int, void*),
                           void *userdata)
{
    if(!r)
        return;
```

```
    int_bin_tree_traverse(r->left);
    (*callback)(r->val, userdata);
    int_bin_tree_traverse(r->right);
}
```

Чтобы решить с помощью этой функции прежнюю задачу — напечатать все элементы дерева — нам придётся ввести дополнительную функцию в качестве callback'а:

```
void int_callback_print(int data, void *userdata)
{
    printf("%d ", data);
}
```

Обратите внимание, что параметр `userdata` здесь не используется; в самом деле, печать очередного элемента никак не зависит от «результатов» печати предыдущих элементов, так что передавать в качестве пользовательских данных здесь нечего. Если адрес корневого элемента дерева находится в переменной `root`, то напечатать содержимое дерева мы теперь сможем, например, таким вызовом:

```
int_bin_tree_traverse(root, int_callback_print, NULL);
```

В качестве адреса пользовательских данных мы передали `NULL`; можно было, в принципе, передать что угодно, всё равно это значение игнорируется нашим callback'ом, но написать в подобной ситуации именно `NULL` — значит облегчить потенциальному читателю понимание нашей программы.

Пока назначение «пользовательских данных» осталось непонятным, но это лишь потому, что мы рассмотрели такой «особенный» пример. Всё станет ясно, если мы попробуем что-то сделать не с каждым элементом дерева в отдельности, а со всем деревом целиком. Для начала попробуем просуммировать элементы дерева; для этого нам потребуется переменная типа `int`, в которой будет накапливаться сумма; именно её адрес мы и передадим в качестве пользовательских данных в callback-функцию, которая для этой задачи примет следующий вид:

```
void int_callback_sum(int data, void *userdata)
{
    int *sum = userdata;
    *sum += data;
}
```

Зная, что через параметр `userdata` нам передали адрес целочисленной переменной, мы приводим этот адрес к типу `int*` и, используя его, прибавляем к текущему значению сумматора значение числа из текущего элемента дерева. Это можно записать и короче, без использования локальной переменной:

```
void int_callback_sum(int data, void *userdata)
{
    *(int*)userdata += data;
}
```

Для подсчёта суммы мы теперь можем сделать следующее:

```
int sum;
sum = 0;
int_bin_tree_traverse(root, int_callback_sum, &sum);
```

После выполнения этого фрагмента в переменной `sum` окажется сумма всех элементов дерева.

Рассмотрим более сложную задачу: пусть нам за один проход дерева требуется найти минимальное число, максимальное число и общее количество чисел. Пользовательские данные при этом будут представлять собой *структуру* из трёх полей:

```
struct minmaxcount {
    int count, min, max;
};
```

Напишем следующую callback-функцию:

```
void int_callback_minmaxcount(int data, void *userdata)
{
    struct minmaxcount *mmc = userdata;
    if(mmc->count == 0) {
        mmc->min = mmc->max = data;
    } else {
        if(mmc->min > data)
            mmc->min = data;
        if(mmc->max < data)
            mmc->max = data;
    }
    mmc->count++;
}
```

Вызов теперь будет выглядеть так:

```
struct minmaxcount mmc;
mmc.count = 0;
int_bin_tree_traverse(root, int_callback_minmaxcount, &mmc);
```

После выполнения этого фрагмента структура окажется заполнена значениями, хотя поля `min` и `max` получат значения только в случае, если хотя бы один элемент в дереве присутствовал; это проверяется по значению поля `count`: если он отличен от нуля, имеет смысл рассматривать

минимум и максимум, в противном случае их следует игнорировать, осмысленной информации они не содержат. В самом деле, каков наибольший или наименьший элемент пустого множества? Очевидно, ответ не определён.

4.13.3. Сложные описания и общие правила их прочтения

Указатели на массивы и функции резко усложняют общую структуру объявлений и описаний функций и переменных. Происходит это из-за наличия в описаниях *префиксных* и *суффиксных* символов; к префиксным относятся имена типов, символы «`*`» и модификаторы, включая `const`, а к суффиксным — квадратные скобки с обозначением размерности массива (или без неё, если она может быть восстановлена по структуре инициализатора или вообще не важна в данном контексте) и круглые скобки со списками параметров для вызовов функций. Хаоса добавляет ещё и то, что круглые скобки, помимо обозначения функций, используются также для изменения порядка применения символов в описателе, как мы это видели для указателей на массивы и функции.

Для создания общего впечатления о масштабах катастрофы приведём несколько примеров. Пусть имеется функция

```
int f(int x, int y) { return x + y; }
```

и мы решили написать такую функцию, которая принимает на вход целое число и возвращает адрес этой `f` (или другой функции с таким же профилем) в качестве своего значения. Назовём эту функцию, например, `funret`. Выглядеть это будет так:

```
int (*funret(int x))(int, int)
{
   /* ... */
}
```

Пусть теперь у нас есть десяток функций с профилем как у `f` и мы решили поместить указатели на эти функции в массив с именем, скажем, `funvec`. Попытавшись описать такой массив, мы получим следующее:

```
int (*funvec[10])(int, int);
```

Допустим теперь, что у нас есть двумерный массив чисел типа `double`, например, такой:

```
double matrix[100][10];
```

причём мы в разных случаях используем из него группы по десять идущих подряд строк, чтобы сформировать матрицу 10x10. Для этого нам может потребоваться функция, возвращающая значение выражения вида `matrix+n`, где `n` — некоторое целое число, означающее строку, начиная с которой мы хотим выделить очередную группу строк. Мы помним, что выражение такого вида, как и сама адресная константа `matrix`, имеет тип `double (*t)[10]` (указатель на массив из десяти элементов типа `double`). Функция, которая возвращает такое выражение, будет выглядеть вот так:

```
double (*select_segment(int a, int b))[10]
{
    /* ... */
}
```

А теперь представьте, что вам зачем-то потребовался **указатель** на такую функцию. Выглядеть этот монстр будет так:

```
double (*(*selptr)(int, int))[10];
```

Если кто-нибудь скажет вам, что «это всё на самом деле совсем просто», не верьте: даже самые опытные программисты на Си при виде таких построений вынуждены остановиться хотя бы на секунду, чтобы понять, что́ же здесь написано.

Всё становится ещё хуже, если указатели на функции и массивы приходится передавать в функцию через параметр, а на саму эту функцию делать указатель. Вернёмся к функции `int f(int a, int b)`, с которой мы начали этот параграф. Пусть нам потребовалась функция, которую мы назовём `replace_f` и которая получает в качестве параметра адрес функции с профилем, как у `f`, и адрес такого же типа возвращает. Например, такая функция может нам потребоваться, если какой-нибудь модуль использует внутри себя указатель на функцию, подобную `f`, и значение этого указателя меняется через вызов `replace_f`, при этом, установив новое значение такого указателя, `replace_f` возвращает его старое значение, например, чтобы вызывающий мог его сохранить, а в будущем вернуть на место. Функция `replace_f` сама по себе будет выглядеть не очень страшно, по крайней мере, после всего, что мы уже видели:

```
int (*replace_f(int (*func)(int, int)))(int, int)
{
    /* ... */
}
```

но вот описание указателя на неё способно вогнать в ступор кого угодно. Отметим, что двух идентификаторов в описании переменной быть не

может, поэтому параметр функции, который в описании самой функции имеет имя `func`, в описании указателя на эту функцию должен быть безымянным. При этом тип параметра `int (*func)(int, int)` всё-таки как-то нужно указать, и результатом становится совершенно неудобоваримая последовательность символов «`(*)`», означающая «здесь был бы идентификатор, если бы он имел право здесь быть, но его тут нет, так что начинайте отсюда». Указание на тип, соответственно, выглядит так: `int (*)(int, int)`, а полностью описание указателя на функцию `replace_f` (или другую с тем же профилем) окажется таким:

```
int (*(*replace_f_ptr)(int (*)(int, int)))(int, int);
```

Понятно, что и это не предел, но мы здесь всё же остановимся — с происходящим пора что-то делать. Вспомнив о существовании директивы `typedef`, введём для начала имя для того типа, который у нас оказался на входе и выходе из функции:

```
typedef int (*fptr)(int, int);
```

Теперь описание указателя `replace_f_ptr` можно будет написать гораздо проще:

```
fptr (*replace_f_ptr)(fptr);
```

Ясно, что именно так с самого начала и следовало действовать; к сожалению, несмотря ни на что, *это очевидно далеко не всем*, поэтому при чтении чьих-нибудь программ вы можете в любой момент нарваться на нагромождение скобочек и звёздочек, подобное чему-то из приведённого выше. Поэтому желательно уметь подобные вещи читать, сколь бы нелепым ни выглядело маниакальное нежелание отдельных авторов применять `typedef`. Тем более что правила такого чтения довольно просты, нужно только не запутаться в скобочках.

Чтобы прочитать сложное описание или объявление, нужно сначала найти описываемый идентификатор; всё начинается именно с него. Затем мы двигаемся от него «наружу» — вправо к концу описания и влево к его началу. Суффиксные символы, стоящие справа от нас, обозначающие функции и массивы, имеют приоритет над префиксными, стоящими от нас слева. Если вокруг нас оказалась пара круглых скобок, нужно исчерпать символы внутри этих скобок, и только потом выходить за их пределы. При этом начинаем мы чтение с того, что произносим описываемое имя-идентификатор и добавляем союз «это»; двигаясь «наружу», мы, увидев квадратные скобки, произносим «массив из стольких-то элементов типа...»; увидев круглые скобки справа от нас, произносим «функция-ю, которая получает на вход...», читаем

описания типов параметров, затем продолжаем словами «и возвращает...»; видя звёздочку, произносим «указатель на...»[68]; видя слово `const`, произносим слово «константный-ая»; наконец, добравшись до стоящего в самом начале имени типа, завершаем чтение подходящей концовкой вроде «переменная-ую типа `int`», «значение типа `int`», «область памяти типа `int`» (последнее приходится применять, если у нас вырисовывается константный указатель на начало массива). Конечно, не следует забывать про согласование падежей и прочую косметику, позволяющую остаться в рамках правил русского языка.

Например, уже знакомое нам

```
int (*(*replace_f_ptr)(int (*)(int, int)))(int, int);
```

можно прочитать следующим образом: `replace_f_ptr` — **это** (справа мы упёрлись в закрывающую круглую скобку, так что сначала надо посмотреть, что у нас слева) **указатель на** (содержимое скобок кончилось, выходим наружу, справа что-то есть, поэтому начинаем оттуда) **функцию, которая получает на вход** (читаем тип параметра, для чего находим «исполняющую обязанности идентификатора» конструкцию «`(*)`») **указатель на функцию, принимающую на вход два целых числа и возвращающую значение типа** `int` (список формальных параметров закончился, завершаем фразу про функцию) **и возвращает** (справа опять закрывающая круглая скобка, приходится посмотреть налево) **указатель на** (опять кончилось содержимое круглых скобок, выходим наружу) **функцию, принимающую на вход два целых числа и возвращающую число типа** `int`.

Аналогичным образом описание

```
int (*funvec[10])(int, int);
```

читается так: `funvec` — **это массив из десяти указателей на функции, принимающие на вход два целых числа и возвращающие значение типа** `int`.

Любопытно, что если случайно забыть звёздочку и скобки, можно придать описанию такой смысл, будто мы имеем в виду не указатель на функцию или массив, а «сам по себе» массив или «саму по себе» функцию. Легко можно убедиться, что бдительный компилятор не позволит нам подобных вольностей: функция «сама по себе» в Си бывает только если это действительно прототип или описание функции (а не какой-либо переменной, типа и тому подобного), и точно так же массив «сам по себе» бывает только при описании или объявлении массива.

[68] Это не всегда правильно, поскольку это не всегда именно указатель, а может быть просто адрес; по-английски можно было бы произнести что-то вроде *address of...*, но в русском языке нет соответствующего предлога, вместо этого следующую сущность нужно поставить в родительный падеж, но это уже сложнее описать; в любом случае большинство программистов не делает различия между понятиями «адрес» и «указатель», а зря.

Например, следующие описания ошибочны и компилятор их обрабатывать откажется:

```
void (*f15)(int)(int);  /* ОШИБКА! */
   /* указатель на функцию, возвращающую функцию */

void *m[5](int);        /* ОШИБКА! */
   /* массив функций, а не указателей на них */

int (*f100)()[15];      /* ОШИБКА! */
   /* указатель на функцию, возвращающую массив */
```

4.14. Ещё о возможностях стандартной библиотеки

Стандартная библиотека Си содержит достаточно большое количество разнообразных инструментов для работы; конечно, рассмотреть их все мы не сможем, да это и не нужно. Материала этой главы вам хватит, чтобы составить общее представление о возможностях стандартной библиотеки; дальнейшее — дело самостоятельного изучения. Учтите, что **большинство возможностей, которые рассматриваются в этой главе, лучше не использовать, пока вы не научитесь уверенно программировать на Си**.

4.14.1. Дополнительные функции работы с динамической памятью

Ранее мы рассмотрели всего две функции для работы с динамической памятью — `malloc` для выделения и `free` для освобождения. С освобождением всё довольно просто, ничего кроме `free` для этого не требуется, а вот для выделения памяти есть ещё одна функция — `calloc`:

```
void *calloc(int nmemb, int size);
```

Эта функция предназначена для выделения памяти под массив, содержащий `nmemb` элементов, каждый из которых имеет размер `size`, но это с таким же успехом можно проделать, перемножив эти два числа и вызвав `malloc`; фундаментальное отличие `calloc` от `malloc` состоит в том, что она забивает выделяемую память нулями. Некоторые программисты отдают ей предпочтение именно по этой причине.

Наконец, нельзя не упомянуть функцию `realloc`, которая исходно предназначена для изменения размера уже выделенной области памяти. Профиль её таков:

```
void *realloc(void *ptr, int size);
```

Параметр `ptr` указывает на имеющуюся область памяти, для которой нужно поменять размер, а `size` задаёт этот новый размер. В большинстве случаев функция вынуждена создавать новую область динамической памяти, копировать туда всё содержимое старой области (которую вы задаёте адресом `ptr`), саму эту старую область освобождать, а возвращать адрес новой. После этого адрес, переданный через `ptr`, становится невалидным, и вместо него необходимо использовать адрес, который вернул `realloc`, поэтому чаще всего результат `realloc`'а присваивают той же переменной, значение которой передают его первым параметром, примерно так: `p = realloc(p, sz);`. Несомненным достоинством `realloc` является то, что иногда, когда сразу после данной области памяти достаточное количество памяти свободно, она может сэкономить процессорное время, не копируя информацию — то есть реально увеличив размер имеющейся области памяти. Есть у `realloc` и недостаток: не зная, как в вызывающей программе используется область памяти, если всё же приходится копировать, функция вынуждена копировать всё содержимое области памяти, что в действительности требуется далеко не всегда.

Довольно интересны два специальных случая вызова `realloc`: если первым параметром передан нулевой адрес, то функция работает в точности как `malloc`, если же в качестве второго параметра передан ноль, она работает в точности как `free`. Иногда попадаются программы, написанные на Си, в которых для работы с динамической памятью используется один только `realloc`.

4.14.2. Функции обработки строк

Все функции, перечисленные в этом параграфе, можно очень легко написать самому. Бытует мнение, что библиотечные функции работают быстрее за счёт того, что в их реализации используются архитектурнозависимые ассемблерные вставки; но на самом деле это не так, при современном состоянии оптимизаторов использование ассемблерных вставок никакого выигрыша не даёт. Единственный (но, заметим, вполне достаточный) смысл применения этих функций — в том, чтобы *облегчить сторонним людям чтение вашей программы.* В самом деле, операции, реализуемые этими[69] функциями, бывают нужны очень часто; если использовать свою реализацию таких операций, читатель вашей программы будет вынужден потратить время, пусть и небольшое, на то, чтобы понять, о чём идёт речь, тогда как при использовании

[69] Во всяком случае, теми, которые будут рассмотрены; библиотека содержит много других функций, в том числе и для работы со строками, которые применяются гораздо реже и польза которых сомнительна.

стандартных функций любой, кто знает Си, сразу поймёт, что здесь делается.

Как следствие, в серьёзной разработке следует пользоваться для операций над строками именно этими функциями; но это не значит, что ими следует пользоваться с самого начала обучения программированию. Автор книги много раз видел студентов, прекрасно обращающихся со всякими `strlen`, `strcmp`, `strcpy` и прочими, при этом не понимающих, как устроена строка и не могущих, как следствие, сделать со строкой ничего сколько-нибудь нестандартного. **Пока вашей основной целью является обучение и пока вы не почувствовали полной уверенности в обращении со строками, лучше вообще не прикасаться к стандартным строковым функциям.**

Отметим, что прототипы всех функций из этого параграфа находятся в заголовочном файле `string.h`. Подключайте его только тогда, когда уже достигнете уровня свободного и уверенного владения преобразованиями строк на Си, и не раньше; до той поры реализуйте все эти операции сами, как это делалось в примерах в нашей книжке. Для самопроверки обязательно напишите свои собственные реализации всех перечисленных функций, и только когда всё получится, можете переходить к использованию их библиотечных версий.

Начнём с функции `strlen`, определяющей длину строки:

```
int strlen(const char *str);
```

Здесь вряд ли требуется много пояснений; ранее в примерах мы использовали свою собственную функцию `string_length`, которая делала абсолютно то же самое.

Функция `strcpy` предназначена для копирования строки в заранее подготовленный массив:

```
char *strcpy(char *destination, const char *source);
```

Порядок аргументов тут такой же, как в присваивании: сначала «куда», потом «откуда». Иначе говоря, параметр `destination` задаёт адрес начала массива, *в который* производится копирование, а параметр `source` указывает на начало *строки, которая должна быть скопирована.* **В массиве должно быть достаточно места, и забота об этом лежит на пользователе!** У вас должны быть веские основания считать, что вы выделили достаточно места или описали массив достаточного размера; если присутствует хотя бы малейшая неуверенность, необходимо сначала измерить длину копируемой строки. О возможных последствиях переполнения массива мы писали в §4.6.4 (см. стр. 291); напомним, что за такие вещи увольняют с работы и правильно делают.

Когда с длиной строки могут, как говорится, «быть варианты», существенно правильнее использовать другую функцию, `strncpy`:

```
char *strncpy(char *destination, const char *source, int size);
```

Первые два аргумента используются точно так же, как и для `strcpy`, а третий задаёт размер массива `destination`; если задать его правильно, функция не допустит переполнения, хотя при этом может быть скопирована не вся строка. У этой функции есть другой крайне неприятный недостаток: если не хватило места для копирования, то нулевой байт она в конец массива не записывает, так что в вашем массиве окажется нечто, не являющееся корректной строкой с точки зрения Си. Впрочем, это легко обойти, записав такой байт принудительно; более того, если в массиве осталось свободное место, `strncpy` заполняет его нулями, так что по содержимому последнего байта массива можно понять, хватило места или нет: если там ноль — значит, копия строки полностью уместилась в отведённом пространстве, а если не ноль — то строка была обрезана; если вас это не беспокоит, остаётся занести в последний байт ноль и продолжить работу.

Функции `strcpy` и `strncpy` в качестве своего значения зачем-то возвращают параметр `destination`, причём всегда. Как следствие, их обычно вызывают ради побочного эффекта, игнорируя возвращаемое значение.

Если вам не хочется возиться с самостоятельным выделением памяти под копию строки, можно воспользоваться функцией `strdup`[70]:

```
char *strdup(const char *s);
```

Эта функция сама измерит длину вашей строки, создаст с помощью `malloc` область динамической памяти нужной длины, скопирует туда заданную строку и вернёт адрес начала созданной копии. Не забудьте только потом освободить память с помощью `free`. Многие программисты не любят эту функцию, поскольку она затрудняет проверку правильности программы, ведь выделение памяти происходит не там, где её освобождение: память выделяется где-то в недрах библиотечной функции, а освобождать вы её вынуждены в своей программе.

Для сравнения строк применяются функции `strcmp` и `strncmp`:

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, int n);
```

Обе функции посимвольно сравнивают строки, начиная с адресов `s1` и `s2`; работа заканчивается, если строки различаются символом в очередной позиции, если одна из строк кончилась, а `strncmp` дополнительно заканчивает работу, если просмотрено уже `n` позиций. Если очередной символ в строках различается, та строка, в которой в очередной

[70] Отметим, что эта функция отсутствует в стандарте Си, но присутствует в стандартах, имеющих отношение к Unix; в системе, отличной от Unix, такой функции может не найтись.

позиции оказался символ с меньшим кодом, считается «меньше», чем другая. Если одна из строк кончилась, а вторая ещё нет, кончившаяся считается «меньше» второй. Функции возвращают отрицательное число, если `s1` оказалась «меньше», чем `s2`, положительное — если `s2` оказалась «меньше», чем `s1`, и ноль, если различий обнаружено не было. Правило упорядочивания строк можно выразить одной фразой: строки сравниваются *в лексикографическом (словарном) порядке*.

Иногда в программах можно встретить конструкцию вроде

```
if(!strcmp(str1, str2)) {
    /* ... */
}
```

Как можно догадаться, имеется в виду условие «если строки равны», ведь в этом случае `strcmp` возвращает ноль. Но так писать не надо. Подсознательно операция «not» в условии воспринимается как «что-то не получилось», а в данном случае всё, наоборот, замечательно получилось: строки оказались равны. Лучше написать так:

```
if(0 == strcmp(str1, str2)) {
    /* ... */
}
```

причём ноль оставить слева от операции сравнения: так вы избавляете читателя от необходимости искать глазами правый край условного выражения.

С помощью следующих функций можно найти в заданной строке заданный символ:

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

Обе функции отыскивают символ с кодом `c` в строке, начинающейся с адреса `s`. Различие между ними проявляется только в случае, если таких символов больше одного: функция `strchr` находит *первый* такой символ, а `strrchr` — *последний* (`r` от слова *right*, то есть «правый»). Обе функции возвращают адрес того места, где в строке располагается искомый символ; если символа с заданным кодом в строке не нашлось, возвращается `NULL`.

Функция `strstr` позволяет найти вхождение подстроки в строку:

```
char *strstr(const char *haystack, const char *needle);
```

Этот заголовок заимствован из официальной man-страницы по этой функции; *haystack* по-английски «стог сена», *needle* — иголка. Как можно догадаться, первый параметр задаёт строку, где нужно искать, второй — подстроку, которую нужно искать. Возвращается адрес того места в строке, где найдена искомая подстрока, либо `NULL`, если ничего не нашлось. Например,

```
char ak[] = "abrakadabra";
/* ... */
p = strstr(ak, "kada");
```

занесёт в `p` адрес `ak+4`.

Следующие три функции не имеют прямого отношения к строкам в том смысле, что они не рассматривают нулевой байт в качестве признака конца полезной информации; их объектом работы являются произвольные области памяти. Функция `memset` позволяет заполнить область памяти заданного размера, начиная с заданного адреса, заданным значением байта:

```
void *memset(void *memory, int value, int size);
```

Первый параметр задаёт адрес начала, второй — значение, которым следует заполнять ячейки памяти, и третий — сколько ячеек надо заполнить. Функция возвращает свой первый параметр; зачем она это делает — неизвестно.

Следующие две функции делают практически одно и то же — копируют содержимое заданного размера из одной области памяти в другую:

```
void *memcpy(void *dest, const void *src, int size);
void *memmove(void *dest, const void *src, int size);
```

Для обеих функций первый параметр задаёт, куда производится копирование, второй — откуда, а третий — размер копируемой области. Различие между функциями в том, что `memcpy` предназначена для копирования информации между областями, которые не могут накладываться друг на друга; `memmove` в этом плане хитрее, она делает соответствующие проверки и если области памяти `dest` и `src` перекрываются, производит копирование от начала к концу, когда `dest` находится левее `src`, и от конца к началу, когда правее. Результат копирования, таким образом, всегда остаётся корректным. Эту функцию можно использовать, например, когда в имеющуюся строку нужно вставить несколько символов (в середину) или, наоборот, изъять из строки несколько символов; в обоих случаях остаток строки приходится сдвигать — соответственно вправо и влево. Если по смыслу происходящего области памяти перекрываться не могут (например, `dest` указывает на только что выделенную память, или они относятся к заведомо разным объектам, и т. д.), лучше использовать `memcpy`: она работает быстрее за счёт отсутствия лишних проверок (кто левее, кто правее).

4.14.3. Генерация псевдослучайных чисел

Потребность внести в выполнение программы какое-то разнообразие чаще всего возникает при создании компьютерных игр, хотя, конечно, не только; случайные числа используются и в математических расчётах (небезызвестный «метод Монте-Карло»), и в области обеспечения безопасности (например, можно узнать, что мы имеем дело с тем, за кого он себя выдаёт, если он сможет подписать своей электронной подписью предложенное нами число, но для этого нужно, чтобы заранее этого числа никто не знал).

Современные операционные системы способны генерировать такие случайные числа, которые *по-настоящему случайны*, то есть совершенно непредсказуемы; это делается путём анализа случайных событий, таких как временны́е промежутки между приходящими по сети пакетами или нажатиями клавиш на клавиатуре, длительность дисковых обменов (точнее, не вся длительность, которая довольно хорошо предсказывается, а её отклонения от прогнозируемых величин) и т. п. Такой случайной информации. называемой в приложениях словом «энтропия», в систему поступает сравнительно немного, поэтому *настоящих* случайных чисел можно сгенерировать довольно ограниченное количество.

Во многих случаях, как правило, не связанных с безопасностью — например, в тех же компьютерных играх — непредсказуемость случайных чисел не слишком важна, поскольку никто, скорее всего, не станет пытаться их предсказывать. В такой ситуации можно выбрать некую «хитрую» формулу, по которой каждое следующее число будет получаться из предыдущего, причём для человека, не знающего формулы, такие числа будут выглядеть совершенно случайными. Теперь достаточно каким-то случайным образом задать *самое первое* из этих чисел, и мы сможем получить сколько угодно последующих чисел, которые будут непредсказуемы примерно так же, как тот ковбой из старого анекдота, которого называли неуловимым, потому что никто его не ловил: попросту говоря, никто не делает целенаправленных попыток предсказывать последовательность таких чисел, а на первый взгляд закономерности не видны.

Числа, полученные в составе таких последовательностей, называются ***псевдослучайными***. Конечно, формулу, по которой они вычисляются, никто не держит в секрете, так что если всерьёз задаться такой целью, предсказать псевдослучайные числа можно; именно поэтому они не годятся для применения в области информационной безопасности: там как раз велика вероятность, что кто-то попытается установить закономерность в последовательности. Но тратить время на выяснение скрытых закономерностей в какой-нибудь игрушке никто, скорее всего, не станет, поскольку не сможет из этого извлечь для себя никакой пользы.

Получить «настоящие» случайные числа в ОС Linux можно, открыв на чтение файл псевдоустройства `/dev/random`; это делается обычным вызовом `open`, само чтение выполняется вызовом `read`. Этим методом стоит пользоваться только в случаях, когда вам действительно нужны хорошо распределённые и непредсказуемые числа — например, при генерации всевозможных паролей и криптографических ключей.

Стандартная библиотека языка Си предусматривает две функции для генерации псевдослучайных чисел, прототипы которых описаны в заголовочном файле `stdlib.h`:

```
int rand(void);
void srand(unsigned int seed);
```

Функция `rand`, не принимающая параметров, возвращает случайное (точнее, псевдослучайное) число от `0` до значения `RAND_MAX` включительно. Это значение в разных реализациях библиотеки может отличаться; например, на машине автора книги оно равно максимально возможному значению для 32-битного знакового целого. Если `rand` использовать одну, без `srand`, то получающаяся последовательность выдаваемых чисел будет одна и та же при каждом запуске программы — а именно, такая же, как если бы мы в начале программы написали `srand(1)`.

Чтобы последовательность чисел получалась каждый раз другая, нужно *один раз в начале программы* вызвать функцию `srand`, дав ей в качестве параметра какое-то число, которое будет каждый раз новым. Эта функция инициализирует датчик псевдослучайных чисел, задав начальное число последовательности (само это число в последовательность псевдослучайных чисел, конечно, не войдёт).

Естественным образом возникает вопрос, откуда взять число для `srand`. Пожалуй, самый простой вариант — это «скормить» функции текущее время. Например, если подключить заголовочный файл `time.h`, вам станет доступна функция `time`, выдающая *текущее время как число секунд, прошедшее с 1 января 1970 года*. Функция принимает некий параметр, в качестве которого практически всегда передают `NULL`. С учётом этого вы можете инициализировать генератор случайных чисел примерно так:

```
srand(time(NULL));
```

Подчеркнём, что **это надо сделать один раз в начале выполнения программы!** Начинающие иногда делают ошибку, вызывая `srand` (например, так, как показано выше) перед каждым обращением к `rand`; в такой ситуации `rand`, например, будет возвращать одинаковые числа, если к ней обратиться несколько раз на протяжении одной секунды; это наверняка не то, чего вы хотите.

В большинстве случаев нужно не просто случайное число, а число из заданного диапазона. Проще всего это сделать с помощью взятия

остатка от деления; например, случайное число r от 1 до 12 можно получить с помощью выражения `rand()%12+1`. Но так делать не рекомендуется, поскольку распределение младших бит псевдослучайных чисел может быть неравномерным, а именно младшие биты при таком применении `rand` оказывают влияние на получаемые числа. Одна из версий страницы справочника `man` по функции `rand`, ссылаясь на книгу [3], рекомендует применять в подобных ситуациях следующее выражение:

```
1 + (int)(12.0*rand()/(RAND_MAX+1.0))
```

Это позволяет использовать в равной степени все составляющие псевдослучайного числа, а в целом псевдослучайные числа имеют распределение, достаточно близкое к равномерному.

4.14.4. (*) Средства создания вариадических функций

Под ***вариадическими функциями*** понимаются функции, способные принимать переменное число аргументов. Мы уже видели такие функции: именно таковы `printf`, `scanf` и всех их «родственники». Все эти функции сами написаны на Си. Любопытно отметить, что средства создания вариадических функций, как и многое другое, вытеснены из языка в библиотеку. Чтобы их задействовать, нужно подключить заголовочный файл `stdarg.h`, в котором описан тип `va_list` (нас не должно волновать, что́ он собой представляет; в разных системах реализации этого типа могут быть совершенно различны) и три макроса — `va_start`, `va_end` и `va_arg`.

Сама вариадическая функция должна иметь хотя бы один параметр, снабжённый именем; имя последнего из именованных параметров используется макросом `va_start` для инициализации переменной типа `va_list`. Последующие параметры извлекаются из стека макросом `va_arg`, которому нужно указать, параметр какого типа требуется извлечь; в конце работы (обязательно в той же самой функции!) мы должны поместить макровызов `va_end`, который приведёт дела в порядок.

Припомнив свои знания архитектуры i386 и конвенции CDECL, мы можем прикинуть, что простейшая реализация всей этой механики состоит во взятии адреса именованного параметра, что позволит с помощью арифметики указателей двигаться вдоль стекового фрейма, извлекая последующие параметры; однако не во всех случаях это делается именно так, на некоторых архитектурах и при использовании некоторых компиляторов параметры полностью или частично передаются через регистры, да и при использовании стека существуют другие

возможные решения — например, создать «массив из одного элемента» и обращаться к элементам этого массива, и тому подобное.

Следует особо подчеркнуть, что имеющиеся средства не позволяют понять, сколько параметров было фактически передано в функцию; это предмет соглашения между вызывающим и вызываемым. Например, в функцию `printf` должно быть, помимо форматной строки, передано ровно столько параметров, сколько форматных директив имеется в форматной строке.

Приведём простой пример. Следующая функция принимает произвольное (но не менее одного) количество параметров типа `int`, не равных нулю, и возвращает их сумму; последний параметр в списке параметров должен быть, напротив, равен нулю — это позволяет функции понять, что список параметров кончился:

```
int sum(int c, ...)
{
    va_list vl;
    int s = c, k;

    va_start(vl, c);
    while((k = va_arg(vl, int)) != 0)
        s += k;
    va_end(vl);

    return s;
}
```

Как видим, все три макроса в качестве первого параметра принимают переменную типа `va_list`, причём, поскольку это макросы, нет необходимости с этой переменной делать что-то дополнительное, брать её адрес и т. п. Для макроса `va_arg` вторым параметром выступает тип очередного безымянного параметра нашей функции, и этот макрос разворачивается в выражение, имеющее именно такой тип. У макроса `va_end` всего один параметр — всё та же переменная типа `va_list`.

Приведём более сложный пример. Следующая функция принимает на вход попеременно указатель на строку и целое число, и так сколько угодно раз, пока очередной указатель не окажется нулевым; каждая строка печатается столько раз, сколько указано в следующем за ней параметре (числе):

```
; variadic.c
void print_times(const char *str, ...)
{
    va_list vl;
    const char *p;
```

```
        va_start(vl, str);
        for(p = str; p; p = va_arg(vl, const char *)) {
            int n, i;
            n = va_arg(vl, int);
            for(i = 0; i < n; i++)
                printf("%s ", p);
            printf("\n");
        }
        va_end(vl);
    }
```

Пример вызова этой функции приведём такой:

```
    print_times("once", 1, "twice", 2, "seven times", 7, NULL);
```

В заключение рассказа о вариадических функциях обратим внимание читателя на функции

```
int vprintf(const char *fmt, va_list ap);
int vfprintf(FILE *stream, const char *fmt, va_list ap);
int vsprintf(char *str, const char *fmt, va_list ap);
int vsnprintf(char *str, int size, const char *fmt, va_list ap);
int vscanf(const char *fmt, va_list ap);
int vsscanf(const char *str, const char *fmt, va_list ap);
int vfscanf(FILE *stream, const char *fmt, va_list ap);
```

Все эти функции работают точно так же, как и их аналоги без буквы `v` в имени (то есть `printf`, `fprintf` и т. д.), за исключением того, что вместо списка аргументов переменной длины все эти функции принимают параметр типа `va_list`; это позволяет создавать функции, аналогичные функциям семейств `printf`/`scanf`, то есть принимающие на вход форматную строку и параметры, но при этом производящие какие-то дополнительные действия, а затем вызывающие соответствующие библиотечные функции из приведённого списка для выполнения основной работы.

4.15. (*) Полноэкранные программы на Си

При изучении языка Паскаль мы рассматривали библиотечный модуль `crt`, позволяющий создавать полноэкранные терминальные программы, то есть такие программы, которые работают в окне терминала, используя все его возможности — выводя текст в произвольные места, изменяя цвет фона и букв, заставляя буквы мигать, реагируя на нажатия клавиш, не дожидаясь, пока пользователь нажмёт Enter, и т. д. (см. т. 1, §2.11). Естественно, аналогичные программы можно писать и на Си; больше того, наши возможности при этом не ограничиваются

интерфейсом, созданным во времена MS-DOS для работы в текстовом режиме персональных компьютеров той эпохи.

Пожалуй, самый простой и в то же время надёжный подход к созданию полноэкранных программ для терминала заключается в использовании библиотеки `ncurses`, которая берёт на себя заботу об особенностях низкоуровневой «кухни», связанной с многообразием терминалов и их эмуляторов. Программа, написанная с использованием `ncurses`, будет корректно работать практически на любом терминале — как в окне вашего `xterm`'а, так и, например, на терминале DEC VT52 (1974 года производства), если вы такой, конечно, найдёте — несмотря на то, что для VT52 требуются совершенно иные управляющие последовательности. В современных условиях, когда алфавитно-цифровые терминалы больше не выпускаются, может сложиться обманчивое впечатление, что подобная универсальность (основанная на специальной базе данных, описывающей особенности разных терминалов) более не актуальна, но это не так. Кроме виртуального терминала, эмулируемого в окошке `xterm`'а, ваша программа может столкнуться с текстовой консолью Linux, с той же текстовой консолью для FreeBSD, с запущенным на Windows клиентом удалённого доступа (например, putty), с эмулятором терминала для последовательного порта (самый популярный из них — Minicom, но есть и другие), с различными программами из менее популярных версий Unix (таких как AIX или Solaris). Все эти эмуляторы терминалов хотя и похожи друг на друга, но всё-таки различаются.

4.15.1. Простой пример

Мы начнём с того же примера, с которого начали при изучении паскалевского модуля `crt` — выведем фразу «`Hello, world!`» в центр пустого терминального окна, подождём пять секунд и завершимся. При этом мы сразу же заметим, что наши возможности оказались шире, чем при использовании `crt`: программа после завершения полностью восстановит состояние терминала, вернув на место весь текст, который там был перед её запуском.

Надо сказать, что такое восстановление возможно не на любом терминале. Так, если мы заставим нашу программу поверить, что она работает на классическом DEC vt100, текст на экране она уже не восстановит; чтобы проверить это, достаточно перед запуском нашей программы дать команду «`export TERM=vt100`»; различия в поведении программы будут видны невооружённым глазом. Для программ, использующих управление цветом текста, разница будет ещё заметнее: vt100 был чёрно-белым, так что раскрашивать текст `ncurses` тоже откажется. При этом все возможности, которые терминал vt100 поддерживал, работают и в `xterm`'е, поэтому программы в целом продолжают корректно функционировать, несмотря на то, что тип терминала им сообщён заведомо неправильный.

Для нашей простейшей программы потребуется, во-первых, умение *инициализировать* библиотеку `ncurses`, то есть сообщать ей, что пора

принять на себя управление терминалом. Это делается вызовом функции `initscr` без параметров. Дальше нам потребуется узнать, сколько строк имеет наше окно терминала и сколько знакомест содержится в каждой строке. Для этого мы опишем две целочисленные переменные `row` и `col` и напишем такую строку:

```
getmaxyx(stdscr, row, col);
```

Поясним, что слово `stdscr` (*standard screen*) означает *окно* на весь терминал; дело в том, что библиотека `ncurses` позволяет объявлять отдельные «окна» в разных местах терминала и вывод в каждое такое окно выполнять независимо от других окон, очищать окно, производить в нём скроллинг текста; это бывает удобно при построении сложных пользовательских интерфейсов. В нашей простой программе работа с окнами не нужна, так что мы ограничимся окном `stdscr`. Ещё одна особенность `ncurses` состоит в том, что при работе с экранными позициями сначала всегда указывается координата по вертикали (номер строки), а координата по горизонтали (номер столбца) идёт второй; буквы `yx` в названиях макросов и функций помогут об этом не забыть.

Внимательный читатель может удивиться, почему перед именами переменных мы не поставили операцию взятия адреса, ведь `getmaxyx`, очевидно, должна *записать* значения в эти переменные. Дело в том, что `getmaxyx` — это на самом деле не функция, а *макрос*, который разворачивается во фрагмент текста, содержащий обычные присваивания. Возможно, создатели `ncurses` (точнее, создатели её предшественницы `curses`, с которой поддерживается совместимость) избрали здесь далеко не лучшее решение: строка действительно выглядит странно и требует пространного пояснения. Впрочем, у такого решения есть и несомненное достоинство: переменные `row` и `col` могут иметь произвольный целочисленный тип, лишь бы его разрядности хватило для хранения соответствующих чисел.

Для перемещения курсора в нужную позицию мы воспользуемся функцией `move`, которая получает два параметра: номер строки и номер столбца. В отличие от паскалевского `gotoxy`, здесь, как и везде в `ncurses`, первой указывается координата по вертикали; кроме того, необходимо помнить, что координаты отсчитываются с нуля, а не с единицы, то есть координаты верхнего левого угла экрана задаются парой $(0, 0)$. Координаты для выдачи сообщения мы вычислим точно так же, как когда-то делали это для аналогичной паскалевской программы: вертикальный размер экрана просто поделим пополам, а из горизонтального вычтем длину нашего сообщения и пополам поделим уже то, что останется.

В отличие от Паскаля, для вывода на экран в полноэкранных программах придётся воспользоваться специально предназначенными для

этого функциями, предоставленными библиотекой `ncurses`; стандартные функции вроде `printf` или `putchar` здесь не подойдут. Кроме того, вывод на экран в программах на основе `ncurses` имеет одну особенность, которая поначалу кажется странной и неудобной, но к которой можно довольно быстро привыкнуть: **операции вывода не оказывают никакого влияния на реальное содержимое экрана до тех пор, пока мы не потребуем привести экран в актуальное состояние**.

Простейшая функция, которая просто выводит на экран заданную строку в текущей позиции курсора, называется `addstr` и принимает один параметр — собственно выводимую строку; эту функцию мы и используем. Отметим, что аналогичная функция для вывода одного символа называется `addch`, и её мы тоже будем использовать в последующих примерах. Проделав операцию вывода, мы спрячем курсор, чтобы он не отвлекал внимание от выданного сообщения; в программе на Паскале мы делали это, перемещая курсор в верхний левый угол, но `ncurses` позволяет сделать правильнее: вызвав `curs_set(0)`, мы вообще уберём курсор с экрана. Что касается приведения экрана в соответствие с нашим выводом, то это делается с помощью функции `refresh`, которая вызывается без параметров.

Закончив с выводом, мы подождём пять секунд. Надо сказать, что функция, позволяющая сделать такую пятисекундную задержку, не имеет никакого отношения к `ncurses` (в отличие от Паскаля, где процедура `delay` предоставляется модулем `crt`); мы воспользуемся стандартной функцией `sleep`, описанной в заголовочном файле `unistd.h`. После этого останется только завершить работу с `ncurses`, вызвав функцию `endwin` (тоже без параметров) и на этом закончить нашу программу. Полностью текст программы получается такой:

```
/* curses_hello.c */
#include <curses.h>
#include <unistd.h>

const char message[] = "Hello, world!";
enum { delay_duration = 5 };

int main()
{
    int row, col;
    initscr();
    getmaxyx(stdscr, row, col);
    move(row/2, (col-(sizeof(message)-1))/2);
    addstr(message);
    curs_set(0);
    refresh();
    sleep(delay_duration);
    endwin();
```

```
    return 0;
}
```

Как уже было сказано, эта программа, в отличие от своей паскалевской предшественницы, восстановит после себя состояние терминала; есть и ещё одно важное отличие: если мы не хотим ждать пять секунд, мы можем выполнение этой программы прекратить так же, как прекращали выполнение других программ — нажатием Ctrl-C. В принципе, `ncurses` умеет действовать так же, как модуль `crt` — перепрограммировать терминал, чтобы он обрабатывал всевозможные комбинации клавиш как обычные символы, но об этом мы должны библиотеку попросить сами. Как это делается, мы расскажем в следующем параграфе.

4.15.2. Обработка клавиатурных и других событий

В обычных условиях терминал работает в так называемом ***каноническом режиме ввода***, при котором активная программа получает введённые пользователем символы только после нажатия Enter (сразу всю строку); некоторые события, поступающие от клавиатуры, драйвер терминала обрабатывает сам — например, нажатие Backspace приводит к удалению последнего введённого символа, то есть активная программа в итоге не получает ни этот удалённый символ, ни спецсимвол, сгенерированный клавишей Backspace; комбинации `Ctrl-C`, `Ctrl-D`, `Ctrl-Z` и некоторые другие имеют специальный смысл. Вводимые символы драйвер терминала отображает на экране.

Как правило, в полноэкранных интерактивных программах такой режим ввода неудобен. Мы видели, что паскалевский модуль `crt` перенастраивает драйвер терминала: активная программа начинает получать информацию о нажатых клавишах сразу после их нажатия и может обрабатывать не только алфавитно-цифровые клавиши, но и всевозможные стрелочки, функциональные клавиши и т. п., комбинации клавиш разом утрачивают особый смысл (так что мы не можем прервать зациклившуюся программу с помощью `Ctrl-C`), вводимые символы на экране не отображаются, если только программа сама их не выведет.

Библиотека `ncurses` тоже предоставляет аналогичные возможности, но поскольку, в отличие от модуля `crt`, она изначально была ориентирована на управление терминалами в ОС Unix, её подход к установке режима терминала существенно более гибок: отдельные особенности режима работы терминала могут включаться и выключаться независимо друг от друга, так что мы можем, например, выключить «канонический» режим и начать обрабатывать нажатия на клавиши немедленно, не дожидаясь конца строки, но при этом сохранить в действии комбинации `Ctrl-C`, `Ctrl-D` и прочие (хотя при желании можем выключить и их; тогда, например, нажатие `Ctrl-C` приведёт к получению нашей

программой псевдосимвола с кодом 3). Отдельно мы можем указать, нужно ли отображать на экране вводимые символы; обычно, впрочем, нам это не нужно — весь вывод лучше производить самим.

Для начала следует решить, хотим ли мы, чтобы комбинации клавиш вроде Ctrl-C и Ctrl-D продолжали работать как обычно. Если да, то следует вызвать функцию cbreak, в противном случае — функцию raw; обе вызываются без параметров. При желании мы можем вернуть терминал в исходный («канонический») режим, вызвав соответственно функцию nocbreak или noraw. Далее следует выключить автоматическое отображение вводимых символов на экране, вызвав функцию noecho (отменить её эффект можно с помощью функции echo). Наконец, в большинстве случаев следует поручить библиотеке ncurses обработку escape-последовательностей, генерируемых «специфическими» клавишами вроде стрелок; это делается так: keypad(stdscr, 1) (если указать 0 вместо 1, режим обработки последовательностей будет выключен).

После такой настройки терминала мы можем узнавать о нажатиях клавиш на клавиатуре, а также о некоторых других происходящих событиях, вызывая функцию getch, которая, не принимая параметров, возвращает значение типа int. При нажатии на обычные «символьные» клавиши getch возвращает код введённого символа. Если же пользователь нажимает «специальные» клавиши, getch (при условии, что мы не забыли включить режим keypad, см. выше) возвращает значения, находящиеся за пределами диапазона кодов символов; библиотека ncurses предоставляет программисту символические имена для этих значений, такие как KEY_UP, KEY_DOWN, KEY_LEFT и KEY_RIGHT для «стрелочек», KEY_F(1), KEY_F(2), KEY_F(10) и т. д. для функциональных клавиш. В действительности все эти имена обозначают целые числа, например, KEY_RIGHT в версии библиотеки, имевшейся у автора, соответствует числу 261, но другие версии могут предусматривать для этих целей другие численные значения, так что в программе следует, конечно же, использовать имена, в том числе и из соображений наглядности.

При нажатии на Enter и Backspace — клавиши, традиционно генерирующие управляющие псевдосимволы — функция getch может повести себя по-разному в зависимости от реализации: для Enter может быть возвращено как значение 10 (оно же '\n', то есть код символа перевода троки), так и значение KEY_ENTER (в версии, имеющейся у автора, этим именем обозначено число 343); для Backspace — значение 8 ('\b') или KEY_BACKSPACE. Полезно знать, что клавиша Insert обозначается KEY_IC, клавиши PgUp и PgDn — соответственно KEY_PPAGE и KEY_NPAGE.

Выше мы упоминали, что функция getch может сообщить нам не только о нажатиях на клавиши, но и о других событиях; к таким событиям относятся действия с мышью (её перемещения, нажатия и отпускания кнопок), а также *изменение размера экрана*. Обработку

мыши мы рассматривать не будем (заинтересованные читатели могут это сделать сами, в Интернете много хороших текстов на эту тему), а вот обрабатывать изменения размера экрана оказывается, с одной стороны, очень просто, а с другой — весьма полезно. Каждый раз, когда пользователь меняет размер окна терминала, функция `getch` в активной программе возвращает специальное значение `KEY_RESIZE`. Получив это значение, следует воспользоваться уже знакомым нам макросом `getmaxyx`, чтобы узнать новые максимально допустимые значения для координат по вертикали и горизонтали, после чего при необходимости «перерисовать» изображение.

Для примера рассмотрим программу, очень похожую на ту, что мы писали в аналогичной ситуации на Паскале (см. т. 1, стр. 326): программа выведет в середине экрана сообщение «`Hello, World!`», которое затем можно будет перемещать по экрану «стрелочными» клавишами. Выйти из программы можно будет, нажав Escape либо «убив» её с помощью `Ctrl-C` (мы воспользуемся режимом терминала, при котором специальные комбинации клавиш работают).

Для начала мы напишем несколько вспомогательных функций. Само сообщение будет вынесено в константу в начале программы; функция `show_message` будет выдавать сообщение в указанном месте экрана (получая координаты через параметры); функция `hide_message` будет печатать в указанном месте соответствующее количество пробелов, чтобы убрать ранее выданное сообщение с экрана (для вывода отдельного пробела мы воспользуемся функцией `addch`). Функция `move_message` будет перемещать сообщение в новую позицию экрана, одновременно изменяя переменные, в которых хранятся текущие координаты; для этого ей будут передаваться адреса этих двух переменных, а также максимально допустимые значения координат и то, *на сколько* позиций следует сдвинуть сообщение по горизонтали и по вертикали.

Обработку события, связанного с изменением размера терминала, мы вынесем в функцию `handle_resize`, которая, узнав с помощью `getmaxyx` новый размер, вычислит новые максимально допустимые значения координат сообщения и при необходимости переместит сообщение так, чтобы оно по-прежнему находилось в пределах экрана.

Программа целиком будет выглядеть так:

```
/* movehello.c */
#include <ncurses.h>

static const char message[] = "Hello, world!";
enum { key_escape = 27 };

static void show_message(int x, int y)
{
    move(y, x);
```

```
    addstr(message);
    refresh();
}

static void hide_message(int x, int y)
{
    int i;
    move(y, x);
    for(i = 0; i < sizeof(message)-1; i++)
        addch(' ');
    refresh();
}

static void check(int *coord, int max)
{
    if(*coord < 0)
        *coord = 0;
    else
    if(*coord > max)
        *coord = max;
}

static void
move_message(int *x, int *y, int mx, int my, int dx, int dy)
{
    hide_message(*x, *y);
    *x += dx;
    check(x, mx);
    *y += dy;
    check(y, my);
    show_message(*x, *y);
}

static void handle_resize(int *x, int *y, int *mx, int *my)
{
    int row, col;
    getmaxyx(stdscr, row, col);
    *mx = col - sizeof(message) + 1;
    *my = row - 1;
    hide_message(*x, *y);
    check(x, *mx);
    check(y, *my);
    show_message(*x, *y);
}

int main()
{
    int row, col, x, y, max_x, max_y, key;
```

```
    initscr();
    cbreak();
    keypad(stdscr, 1);
    noecho();
    curs_set(0);
    getmaxyx(stdscr, row, col);
    x = (col-(sizeof(message)-1))/2;
    y = row/2;
    max_x = col - sizeof(message) + 1;
    max_y = row - 1;
    show_message(x, y);
    while((key = getch()) != key_escape) {
        switch(key) {
        case KEY_UP:
            move_message(&x, &y, max_x, max_y, 0, -1);
            break;
        case KEY_DOWN:
            move_message(&x, &y, max_x, max_y, 0, 1);
            break;
        case KEY_LEFT:
            move_message(&x, &y, max_x, max_y, -1, 0);
            break;
        case KEY_RIGHT:
            move_message(&x, &y, max_x, max_y, 1, 0);
            break;
        case KEY_RESIZE:
            handle_resize(&x, &y, &max_x, &max_y);
            break;
        }
    }
    endwin();
    return 0;
}
```

Существуют и другие удобные функции для вывода. Одна из них называется `printw`; она принимает такие же аргументы, как привычная нам `printf`, и используется для *форматированного вывода*. Например, если в вышеприведённой программе в функцию `show_message` перед вызовом `refresh` добавить следующие две строки:

```
    move(0, 0);
    printw("(%d,%d)    ", x, y);
```

то в левом верхнем углу экрана программа будет печатать текущие координаты сообщения. Обратите внимание на четыре пробела в конце нашей форматной строки — они добавлены, чтобы с гарантией затереть всё, что осталось от предыдущих координат (например, на случай, если координаты только что выражались трёхзначными числами, а теперь

стали однозначными). Такого же эффекта можно достичь и в одну строку с помощью функции `mwprintw`:

```
mvprintw(0, 0, "(%d,%d)    ", x, y);
```

(нужно только не забывать, что координаты для вывода указываются в последовательности «`y`, `x`», то есть сначала вертикальная, потом горизонтальная).

4.15.3. Управление цветом и атрибутами символов

Изучая Паскаль, мы узнали, что при выводе текста на экран можно с каждым символом связать те или иные *атрибуты*, влияющие на его внешний вид. Библиотека `ncurses` позволяет управлять атрибутами выводимых символов, в том числе цветом самого символа и его фона, если только терминал, с которым мы работаем, цветной[71]. Некоторые атрибуты не зависят от «цветности» терминала: символ можно сделать жирным, инверсным (когда цвета фона и символа меняются ролями), подчёркнутым, мигающим и т. п. Каждый такой атрибут обозначается своим идентификатором: например, `A_BLINK` означает «мигающий», `A_UNDERLINE` — «подчёркнутый», `A_BOLD` — «жирный», `A_DIM` — «тусклый», `A_REVERSE` — «инверсный» и т. д., полный список можно посмотреть, например, в тексте страницы системного справочника, которая называется `attr` (для этого дайте команду `man attr`). Каждый такой атрибут — это отдельный бит, так что их можно сочетать, используя операцию побитового «или».

Пожалуй, проще всего управлять такими атрибутами с помощью функций `attron` и `attroff`; обе принимают по одному параметру, первая *включает* указанные атрибуты, вторая их, наоборот, *выключает*. Например, фрагмент

```
attron(A_BOLD);
addstr("Hello, ");
attron(A_UNDERLINE);
addstr("wonderful");
attroff(A_BOLD|A_UNDERLINE);
addstr(" world!");
```

напечатает фразу «`Hello, wonderful world!`», причём первые два слова будут напечатаны жирными, второе к тому же подчёркнуто, а третье

[71] Конечно, все эмуляторы терминалов цветные, но пользователь по каким-то причинам может захотеть, чтобы запускаемые программы не использовали цвет; для этого достаточно установить соответствующий тип терминала, см. замечание на стр. 416. Кроме того, в частных коллекциях попадаются работающие (и даже используемые) настоящие терминалы, выпущенные 20–30 лет назад. Интересы владельцев таких терминалов можно, конечно, проигнорировать, но нужно ли?

будет напечатано обычным — ни жирности, ни подчёркивания. Есть и другие способы работы с атрибутами текста; функций, предназначенных для этого, `ncurses` предоставляет больше двух десятков, но рассматривать их мы для экономии места не будем. Отметим только, что при выводе по одному символу с помощью `addchar` атрибуты можно «присоединить» к коду символа с помощью всё той же операции побитового «или». Например, чтобы выдать мигающую жирную звёздочку, можно сделать так:

```
addch('*' | A_BOLD | A_BLINK);
```

Рассмотрим теперь возможности по управлению цветом. Прежде чем пытаться с ним работать, следует проверить, возможно ли это на имеющемся терминале. Это делается вызовом функции `has_colors`; если она вернула «ложь» (то есть `0`), то цвета нам недоступны; что делать в этом случае — зависит от решаемой задачи, но лучше всего написать программу так, чтобы на чёрно-белом терминале она сохраняла работоспособность, не делая попыток задействовать цвет. Например, мы можем завести переменную `work_bw` (`bw` — это обычное сокращение для слов *black&white*, обозначающих чёрно-белое, будь то фото, изображение или режим работы) и во всех наших функциях, осуществляющих вывод, прибегать к управлению цветом лишь в том случае, если эта переменная хранит значение «ложь», в противном случае обходиться атрибутами символов, не зависящими от цветности. В начале нашей главной функции следует предусмотреть примерно такой фрагмент:

```
initscr();
work_bw = !has_colors();
if(!work_bw)
    start_color();
```

Заметим, мы можем при необходимости занести в переменную `work_bw` значение «истины», даже если терминал у нас цветной, но пользователь каким-то образом (например, через параметры командной строки или как-то ещё) потребовал от нас работать в чёрно-белом режиме; это вполне обычная ситуация, ведь при этом гораздо меньше устают глаза.

В дальнейших примерах мы для краткости предполагаем, что `has_colors` вернула «истину» и работать в чёрно-белом режиме нас никто не заставляет. Наша цель сейчас — показать, как *работать* с цветом, а не как *обходиться* без него.

Библиотека `ncurses` поддерживает восемь основных цветов, перечисленных в табл. 4.3. Спецификация интерфейса библиотеки предусматривает средства для создания других цветов, но в реальной жизни эти средства никогда не работают — соответствующие возможности не поддерживаются ни терминалами, ни их эмуляторами, а библиотечные

Таблица 4.3. Обозначения цветов в библиотеке `ncurses`

`COLOR_BLACK`	чёрный	`COLOR_BLUE`	синий
`COLOR_RED`	красный	`COLOR_MAGENTA`	фиолетовый
`COLOR_GREEN`	зелёный	`COLOR_CYAN`	голубой
`COLOR_YELLOW`	жёлтый	`COLOR_WHITE`	белый

функции, предназначенные для этого, просто не реализованы (всегда возвращают ошибку).

Цвета символов в `ncurses` всегда устанавливаются так называемыми *цветовыми парами*; пара состоит из цвета символа и цвета фона. С цветовыми парами связано довольно много путаницы. Сами пары различаются по номерам, причём пара номер `0` соответствует цветам «по умолчанию» и не должна изменяться. Чтобы вывести на экран цветную надпись, нужно выбрать номер цветовой пары, с помощью функции `init_pair` назначить этой паре цвет текста и цвет фона, после чего использовать в качестве атрибута выводимого текста выражение `COLOR_PAIR(n)`, где `n` — номер цветовой пары. Функция `init_pair` принимает три параметра: номер пары, номер цвета для текста и номер цвета для фона. Например,

```
init_pair(1, COLOR_WHITE, COLOR_BLUE);
attrset(COLOR_PAIR(1) | A_UNDERLINE);
addstr("White on blue");
refresh();
```

выведет надпись «`White on blue`» белыми буквами по синему фону, используя для этого цветовую пару №1, причём надпись будет подчёркнута. Если вы будете применять несколько разных сочетаний цветов для символа и его фона (а так, скорее всего, и будет), то для каждого сочетания нужно будет использовать свой собственный номер цветовой пары. Использовать один и тот же номер для разных сочетаний не получится: **если изменить цвета для существующей цветовой пары, то на экране соответствующим образом изменятся цвета всех символов, выданных с использованием этого номера цветовой пары**. Кстати, это свойство само по себе можно использовать для создания интересных визуальных эффектов.

Заслуживает внимания вопрос, *какие* числа можно использовать в качестве номеров цветовых пар. Как ни странно, эту тему документация и всевозможные руководства тщательно обходят стороной. Можно определённо сказать, что значения номеров цветовых пар, превышающие `255`, создадут проблемы, обусловленные распределением отдельных разрядов в атрибуте символа. Скорее всего, проблемы могут возникнуть и с мéньшими номерами, причём вопрос о максимально допустимом номере цветовой пары, судя по всему, зависит от реализации библиотеки.

В некоторых описаниях упоминается константа `COLOR_PAIRS` (хотя при этом не говорится, чему она должна быть равна), но в наиболее популярной реализации `ncurses` этот идентификатор представляет собой не константу, а переменную.

С другой стороны, ясно, что при наличии всего восьми различных цветов можно (практически заведомо[72]) обойтись не более чем 64 цветовыми парами. Конечно, если бы синтез дополнительных цветов работал, это соображение ничем бы помочь не могло — но он не работает, так что каким-то ещё цветам попросту неоткуда взяться. Поскольку пару с нулевым номером трогать нельзя, нам могут потребоваться для представления всех возможных цветовых комбинаций номера от `1` до `64`; как показывает практика, в существующих реализациях `ncurses` использование всех этих номеров проблем не создаёт.

Отметим ещё один момент. Добавление атрибута `A_BOLD` фактически изменяет цвет символа; это как раз та причина, по которой паскалевский модуль `crt` вводит восемь цветов для фона и целых шестнадцать — для самих символов.

Для демонстрации возможностей управления цветом мы напишем программу, очень похожую на ту, что писали для той же цели на Паскале (см. т. 1, стр. 331): она тоже будет выводить на экран звёздочки всеми возможными цветами. Номера используемых цветовых пар мы жёстко привяжем к составляющим их цветам по формуле $8 \cdot b + f + 1$, где b — номер цвета для фона, f — для символа. Единицу мы прибавляем, чтобы не использовать «запрещённую» цветовую пару № 0.

В принципе, константы `COLOR_BLACK`, `COLOR_RED` и т. д. равны числам от `0` до `7` включительно, но поскольку этот момент нигде в документации не зафиксирован, мы не будем это использовать; вместо этого опишем массив из восьми элементов и занесём в его элементы значения для разных цветов; идентифицировать цвета будем индексами этого массива, что гарантирует нам использование именно номеров от 0 до 7, даже если в какой-нибудь реализации библиотеки цветовые константы будут равны каким-нибудь другим числам.

Привязку номеров цветовых пар к номерам самих цветов будет обеспечивать функция `setpair`, которая, получив номера цветов для символа и фона, будет вычислять по вышеприведённой формуле нужный номер цветовой пары, устанавливать для этой пары соответствующие цвета, а сам номер пары возвращать в качестве значения.

За вывод одной строки будет отвечать функция `make_line`; символы в каждой строке будут печататься одним и тем же цветом (самого

[72] Можно себе представить программу, в которой некоторые цветовые пары повторяются, если они исходно соответствуют не цветовым сочетаниям, а, например, элементам визуального интерфейса, чтобы можно было синхронно «перекрасить» элементы одного типа; впрочем, даже в этом случае трудно себе представить больше 64 цветовых пар в одной программе.

символа) поверх различных цветов фона, причём каждый второй символ получит дополнительно атрибут `A_BOLD`, а каждая вторая пара символов будет сделана ещё и мигающей. Наконец, всю картинку на экране будет формировать из отдельных строк функция `make_screen`.

После того, как картинка будет сформирована, в главной функции программы мы предусмотрим цикл чтения с клавиатуры, который будет прекращаться при получении символа `Escape` (код 27), а при нажатии любых других клавиш из тела цикла будет вызываться функция `shift_pairs`, которая для всех задействованных 64 цветовых пар будет изменять значения цветов, что позволит наблюдать интересный визуальный эффект.

Полностью текст программы получается таким:

```
/* curses_col.c */
#include <stdio.h>
#include <curses.h>

enum { color_count = 8 };
static const int all_colors[color_count] = {
    COLOR_BLACK, COLOR_RED, COLOR_GREEN, COLOR_YELLOW,
    COLOR_BLUE, COLOR_MAGENTA, COLOR_CYAN, COLOR_WHITE
};

static int setpair(int fg, int bg)
{
    int n = bg*8 + fg + 1;
    init_pair(n, fg, bg);
    return n;
}

static void make_line(int line, int width, int fgcolor)
{
    int i, j, w, pn, att;
    w = width / color_count;
    for(i = 0; i < color_count; i++) {
        move(line, i*w);
        for(j = 0; j < w; j++) {
            pn = setpair(fgcolor, all_colors[i]);
            att = COLOR_PAIR(pn);
            if(j % 2 == 0)
                att |= A_BOLD;
            if((j / 2) % 2 == 1)
                att |= A_BLINK;
            attrset(att);
            addch('*');
            refresh();
        }
    }
}

static void make_screen(int h, int w)
{
    int i;
```

```
    clear();
    for(i = 0; i < h; i++)
        make_line(i, w, all_colors[i % color_count]);
}

static void shift_pairs(int shift)
{
    int i;
    for(i = 1; i <= color_count * color_count; i++) {
        int fg = (i + shift) % color_count;
        int bg = ((i + shift) / color_count) % color_count;
        init_pair(i, fg, bg);
    }
}

int main()
{
    int row, col, ch, shift;
    initscr();
    if(!has_colors()) {
        endwin();
        fprintf(stderr, "I can't show colors on a BW screen\n");
        return 1;
    }
    cbreak();
    keypad(stdscr, 1);
    noecho();
    curs_set(0);
    start_color();
    getmaxyx(stdscr, row, col);
    make_screen(row, col);
    shift = 2;
    while((ch = getch()) != 27) {
        shift_pairs(shift++);
        refresh();
    }
    endwin();
    return 0;
}
```

4.15.4. Клавиатурный ввод с тайм-аутами

До сих пор в наших примерах функция `getch` ждала, пока пользователь не нажмёт какую-нибудь клавишу, то есть в ожидании очередного события программа ничего не делала; но как быть, если мы хотим всё время выполнять какие-то действия, позволяя пользователю вмешаться в происходящее путём нажатия клавиш, но не тратя время на ожидание? В программах на Паскале мы решали эту проблему с помощью функции `keypressed`, вызывая её время от времени и выполняя чтение символа с клавиатуры лишь если `keypressed` вернула «истину». В `ncurses` аналога для `keypressed` нет, зато есть возможность гибкой настройки режима ввода. По умолчанию, как мы видели, функция `getch` ждёт

наступления очередного события, если оно ещё не произошло на момент её вызова; но режим работы можно изменить так, чтобы `getch` всегда немедленно возвращала управление, причём если никаких событий не произошло, возвращаемое значение будет равно константе `ERR`. Более того, можно задать для `getch` ***тайм-аут*** (в миллисекундах); в этом случае функция будет ожидать наступления события, но не более чем в течение заданного тайм-аутом временно́го периода, после чего, если никакое событие так и не произойдёт, благополучно вернёт `ERR`.

Управлять режимом тайм-аута можно с помощью функции, которая так и называется `timeout`; функция принимает один целочисленный параметр, означающий величину тайм-аута в миллисекундах, то есть, например, значение параметра `100` предписывает функции `getch` ждать наступления события не более чем в течение 0,1 секунды, и если ничего за это время не произойдёт, вернуть значение `ERR`. Ноль в качестве аргумента для `timeout` означает, что ждать не следует вовсе: если к моменту вызова `getch` никаких событий не произошло, она при этом вернёт `ERR` немедленно (такой режим называется неблокирующим). Наконец, любое отрицательное значение аргумента `timeout` (например, `-1`) установит обычный блокирующий режим, при котором `getch` ждёт наступления события без ограничения по времени.

Совместную работу функций `timeout` и `getch` мы продемонстрируем на примере программы, которая очень похожа на программу `MovingStar`, написанную нами на Паскале (см. т. 1, стр. 328): на экран выводится неподвижная звёздочка, которую можно заставить двигаться по экрану, нажав одну из «стрелочных» клавиш. Звёздочка при этом начнёт перемещаться, не дожидаясь нажатий на другие клавиши, но её можно будет в любой момент запустить в другом направлении, нажав соответствующую «стрелку», или остановить, нажав пробел.

Для удобства всю информацию о текущем состоянии звёздочки (текущие координаты и вектор скорости) мы соберём в одну структурную переменную, для работы с ней напишем несколько вспомогательных функций. В ходе инициализации мы вызовем `timeout` с константой, имеющей значение `100`, в качестве аргумента, так что `getch`, используемая нами в главном цикле программы, будет возвращать обычные коды клавиш при их нажатии, а при отсутствии событий — значение `ERR`, которое можно рассматривать как признак того, что прошли очередные `100` миллисекунд и звёздочку пора перемещать в следующую позицию. Полностью текст программы выглядит так:

```
/* movingstar.c */
#include <ncurses.h>

enum { delay_duration = 100 };
enum { key_escape = 27 };

struct star {
```

```
    int cur_x, cur_y, dx, dy;
};

static void show_star(const struct star *s)
{
    move(s->cur_y, s->cur_x);
    addch('*');
    refresh();
}

static void hide_star(const struct star *s)
{
    move(s->cur_y, s->cur_x);
    addch(' ');
    refresh();
}

static void check(int *coord, int max)
{
    if(*coord < 0)
        *coord += max;
    else
    if(*coord > max)
        *coord -= max;
}

static void move_star(struct star *s, int max_x, int max_y)
{
    hide_star(s);
    s->cur_x += s->dx;
    check(&s->cur_x, max_x);
    s->cur_y += s->dy;
    check(&s->cur_y, max_y);
    show_star(s);
}

static void set_direction(struct star *s, int dx, int dy)
{
    s->dx = dx;
    s->dy = dy;
}

static void handle_resize(struct star *s, int *col, int *row)
{
    getmaxyx(stdscr, *row, *col);
    if(s->cur_x > *col)
        s->cur_x = *col;
    if(s->cur_y > *row)
        s->cur_y = *row;
}

int main()
{
    int row, col, key;
    struct star s;
```

```
    initscr();
    cbreak();
    timeout(delay_duration);
    keypad(stdscr, 1);
    noecho();
    curs_set(0);
    getmaxyx(stdscr, row, col);
    s.cur_x = col/2;
    s.cur_y = row/2;
    set_direction(&s, 0, 0);
    while((key = getch()) != key_escape) {
        switch(key) {
        case ' ':
            set_direction(&s, 0, 0);
            break;
        case KEY_UP:
            set_direction(&s, 0, -1);
            break;
        case KEY_DOWN:
            set_direction(&s, 0, 1);
            break;
        case KEY_LEFT:
            set_direction(&s, -1, 0);
            break;
        case KEY_RIGHT:
            set_direction(&s, 1, 0);
            break;
        case ERR:
            move_star(&s, col-1, row-1);
            break;
        case KEY_RESIZE:
            handle_resize(&s, &col, &row);
            break;
        }
    }
    endwin();
    return 0;
}
```

4.15.5. Обзор остальных возможностей `ncurses`

Как обычно в таких случаях, мы не рассмотрели и десятой доли возможностей библиотеки `ncurses`; их полное изучение потребовало бы отдельной книги. Наша цель состояла в том, чтобы продемонстрировать основные возможности и более-менее осознать концепцию в целом. Так, мы рассмотрели только несколько функций из двух десятков, предназначенных для управления атрибутами выводимого текста; мы не стали рассматривать возможности по созданию так называемых *окон* — прямоугольных областей на экране, позволяющих выводить текст с использованием своих координат в каждом окне; мы оставили за рамками нашего текста бóльшую часть функций инициализации и настройки

режима терминала, совсем не упомянули возможности скроллинга (как всего экрана, так и отдельных его частей).

Кроме прочего, вместе с библиотекой `ncurses` распространяются дополнительные библиотеки: `menu`, `panel` и `cdk`. Первая из них, `menu`, позволяет составлять и выдавать на экран текстовые меню, в которых в зависимости от настроек пользователь может выбрать один или несколько пунктов. Библиотека `panel` предназначена для работы с прямоугольными окнами, которые могут частично перекрываться, накладываться одно на другое, окна можно перемещать по экрану, убирать и снова показывать и т. д. Наконец, `cdk` (*curses development kit*) предоставляет целый ряд *виджетов* — готовых элементов для построения диалоговых окон: поля ввода текста, кнопки, переключатели и тому подобное.

4.16. (*) Программа на Си без стандартной библиотеки

В процессе изучения языка Си мы неоднократно повторяли, как мантру, фразу «Си мы любим не за это». Теперь, когда практически все возможности этого языка нам известны, настала пора понять, *за что же* мы его всё-таки любим. Эту небольшую главу мы посвятим эксперименту, который наглядно продемонстрирует одно уникальное свойство языка Си, делающее его во многих случаях незаменимым.

Начнём с того, что напишем простенькую программу, которая принимает ровно один параметр командной строки, рассматривает его как имя и здоровается с человеком, чьё имя указано, фразой `Hello, dear NNN!` (имя подставляется вместо `NNN`). Для начала попробуем написать эту программу самым очевидным способом:

```
/* greet.c */
#include <stdio.h>

int main(int argc, char **argv)
{
    if(argc < 2) {
        printf("I don't know how to greet you\n");
        return 1;
    }
    printf("Hello, dear %s\n", argv[1]);
    return 0;
}
```

Откомпилировав эту программу, мы получим сравнительно небольшой исполняемый файл; в системе, имевшейся в распоряжении автора этих строк, полученный «бинарник» занимал около 8,5 Кб при включённой

отладочной информации и меньше семи Кб — при её отключении (то есть при компиляции без флага `-g`). Однако тут имеет место некоторое жульничество: в этот исполняемый файл не включён код функции `printf` и всего, что нужно для её использования (она сама вызывает много других библиотечных функций); вместо этого компилятор построил нам так называемый *динамически загружаемый исполняемый файл*, что означает, что в процессе выполнения в память сначала будет загружен сам этот файл, потом код, помещённый в него, потребует загрузки в память динамически загружаемой версии стандартной библиотеки языка Си (если интересно, это обычно файл `/lib/libc.so.X`, где `X` — номер версии библиотеки). Недостаток такого подхода легко обнаружить, если запустить нашу программу под управлением утилиты `strace`, позволяющей увидеть список всех выполненных программой системных вызовов[73]; нас, впрочем, будет интересовать только их количество — чтобы узнать его, достаточно посчитать строки в полученном файле: количество системных вызовов будет на единицу меньше.

```
avst@host:~$ strace -o LOG ./greet Andrey
Hello, dear Andrey
avst@host:~$ wc -l LOG
29 LOG
avst@host:~$
```

Итак, наша программа сделала 28 системных вызовов, причём полезны из них только два последних — `write`, который напечатал строчку, и `exit_group`, используемый современными версиями стандартной библиотеки вместо традиционного `_exit` для завершения программы. Все остальные 26 вызовов — подготовительная работа, которая в основном состоит как раз в загрузке динамической библиотеки.

Попробуем запретить компилятору жульничать — потребуем от него создать статически собранный исполняемый файл, не зависящий ни от каких динамических библиотек. Это достигается ключом `-static`:

```
gcc -Wall -g -static greet.c -o greet
```

Посмотрев на размер полученного исполняемого файла, мы сразу, как говорится, *почувствуем разницу*: размер «бинарника» у нас вырос раз этак в семьдесят; у автора на его системе получилось около 565 Кб при включённой отладочной информации и около 510 Кб при выключенной. С другой стороны, снова сосчитав системные вызовы, мы обнаружим, что вместо 28 их стало всего 10 — что, впрочем, тоже не так мало.

Поскольку в наши планы не входит реализация функции `printf`, попробуем обойтись без неё, обращаясь напрямую к операционной системе. Сразу же отметим, что на большинстве систем (хотя и не на всех)

[73] Напомним, что системным вызовом называется обращение пользовательской задачи к ядру операционной системы.

мы при этом не достигнем совершенно никакой экономии в размере исполняемого файла, а системных вызовов можем сэкономить «целых» два (опять же, результаты на вашей системе могут отличаться), и то при условии, если уложимся в один вызов `write`; но для этого понадобилось бы писать функцию конкатенации строк и множество всяческих проверок на переполнение буфера, что существенно загромоздило бы наш пример. Поэтому мы поступим проще: выводить надпись будем в три приёма, то есть в три обращения к ядру; всю экономию это, увы, сведёт на нет. Выглядеть программа будет так:

```
/* greet2.c */
#include <unistd.h>

static const char dunno[] = "I don't know how to greet you\n";
static const char hello[] = "Hello, dear ";

static int string_length(const char *s)
{
    int i;
    for(i = 0; s[i]; i++)
        ;
    return i;
}

int main(int argc, char **argv)
{
    if(argc < 2) {
        write(1, dunno, sizeof(dunno)-1);
        return 1;
    }
    write(1, hello, sizeof(hello)-1);
    write(1, argv[1], string_length(argv[1]));
    write(1, "\n", 1);
    return 0;
}
```

Попробовав откомпилировать эту программу и с разочарованием убедившись, что мы ничего толком не добились, приступим собственно к тому, ради чего всё затеяли — попытаемся избавиться от использования стандартной библиотеки Си. Для этого нам для начала потребуется заменить ту её часть, в которой реализована небезызвестная «невидимая функция» (а точнее, подпрограмма в ассемблерном смысле) `_start`, хорошо знакомая нам со времён изучения языка ассемблера. Именно эта подпрограмма служит *настоящей* точкой входа в программу, то есть операционная система *на самом деле* вызывает её, а вовсе не `main`; как можно легко догадаться, функцию `main` вызывает как раз подпрограмма `_start`, именно на неё возлагаются обязанности по подготовке

аргументов `main`, а равно и вызов `_exit` (или что там используется вместо него) после того, как `main` завершит свою работу.

Припомнив, что и как расположено в стеке на момент вызова `_start`, мы легко убедимся, что для функции `main` аргументы придётся готовить, так как то, что в стеке уже есть, хотя и представляет собой именно ту информацию, которая нам нужна, но *не совсем в том виде*; напомним, что по адресу `[ESP]` лежит количество параметров командной строки, то есть прямо-таки значение `argc`, но вот дальше лежат указатели на сами параметры, то есть фактически прямо в стеке лежит массив `argv`, но нам ведь нужно, чтобы в стеке лежал адрес этого массива (параметр `argv`). Поэтому мы поступим проще: загрузим из стека в регистр `ECX` значение `argc`, потом в регистре `EAX` сформируем значение `argv`, затем, не трогая имеющуюся в стеке информацию, занесём туда в соответствии с конвенцией вызовов сначала `argv`, потом `argc` и, наконец, вызовем функцию `main`. После её завершения — если, конечно, до этого дело дойдёт, ведь кто-то вполне мог вызвать `_exit` без нашего участия — нам останется извлечь её возвращаемое значение из регистра `EAX` и вызвать `_exit`.

Полностью файл, содержащий подпрограмму `_start`, с учётом конвенции вызовов ОС Linux будет выглядеть так:

```
; start.asm
global _start
extern main
section         .text
_start:         mov     ecx, [esp]      ; argc in ecx
                mov     eax, esp
                add     eax, 4          ; argv in eax
                push    eax
                push    ecx
                call    main
                add     esp, 8          ; clean the stack
                mov     ebx, eax        ; now call _exit
                mov     eax, 1
                int     80h
```

Для ОС FreeBSD придётся заменить последние три строчки на следующие:

```
                push    eax              ; now call _exit
                push    eax
                mov     eax, 1
                int     80h
```

Следующим этапом нашего эксперимента станет написание своей реализации «обёрток» для системных вызовов. По большому счёту, нам достаточно одного лишь вызова `write`, но, чтобы показать, как всё

выглядело бы для более серьёзной программы, мы напишем обёртки для двух вызовов: `write` и `read`; назовём их, во избежание лишней путаницы, `sys_write` и `sys_read`. Как и для «настоящих» обёрток системных вызовов, мы предусмотрим глобальную переменную для хранения кода ошибки, которую, опять-таки во избежание путаницы, назовём `sys_errno`.

Отметим сразу же, что реализации этих двух вызовов, как и вообще всех системных вызовов, имеющих три параметра, разрядность которых не превышает 32 бит, требуют совершенно одинаковой последовательности команд и отличаются только номером системного вызова. Поэтому мы создадим общую для всех «трёхместных» вызовов реализацию, а сами обёртки системных вызовов будут представлять собой лишь точки входа (а не целые подпрограммы), которые будут помещать нужный номер системного вызова в регистр `EAX` и делать простой (безусловный) переход на общую реализацию трёхместного вызова. Что касается этой реализации, то она будет действовать в соответствии с конвенцией CDECL: сохранять старое значение `EBP`, создавать классический стековый фрейм, уже в нём сохранять старое значение `EBX` (согласно правилам CDECL, вызванная подпрограмма имеет право испортить `EAX`, `ECX` и `EDX`, а наша реализация обёрток вынуждена также использовать `EBP` и `EBX`, поэтому эти два регистра она сохраняет в стеке и потом восстанавливает), затем, обратившись к соответствующим позициям стека, извлечь оттуда параметры системного вызова, разложить их в соответствии с конвенцией ОС Linux по регистрам `EBX`, `ECX` и `EDX`, затем, наконец, отдать управление ядру операционной системы; по возвращении выяснить (опять же, в соответствии с конвенцией Linux), не находится ли возвращённое значение в диапазоне от `0xfffff000` до `0xffffffff`, если да — занести его в `sys_errno`, а в `EAX` оставить значение `-1`, если же нет, то просто вернуть управление вызвавшему, предварительно восстановив состояние регистров и стека.

Полностью наш модуль будет выглядеть так:

```
; calls.asm
global          sys_read
global          sys_write
global          sys_errno

section         .text

generic_syscall_3:
                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+8]
                mov     ecx, [ebp+12]
                mov     edx, [ebp+16]
```

```
                int     80h
                mov     edx, eax
                and     edx, 0fffff000h
                cmp     edx, 0fffff000h
                jnz     .okay
                mov     [sys_errno], eax
                mov     eax, -1
.okay:          pop     ebx
                mov     esp, ebp
                pop     ebp
                ret

sys_read:       mov     eax, 3
                jmp     generic_syscall_3
sys_write:      mov     eax, 4
                jmp     generic_syscall_3

section         .bss
sys_errno       resd    1
```

Для FreeBSD подпрограмма `generic_syscall_3` будет выглядеть иначе и гораздо проще благодаря тому, что параметры системного вызова передаются через стек, а в стеке они у нас уже лежат, притом в нужном порядке. Здесь мы сможем себе позволить не оформлять полноценный стековый фрейм, поскольку никаких локальных переменных использовать не будем, сохранять и восстанавливать регистры нам тоже не нужно. Выглядеть это будет так:

```
generic_syscall_3:
                int     80h
                jnc     .okay
                mov     [sys_errno], eax
                mov     eax, -1
.okay:          ret
```

Отметим, что больше различий между Linux и FreeBSD у нас не предвидится, то есть весь оставшийся текст нашего примера подойдёт для обеих систем.

Вернёмся к нашей главной программе и перепишем её с расчётом на использование только что написанных обёрток вместо функции `write` из стандартной библиотеки. Для этого нам придётся убрать директиву `#include`, добавить прототип функции `sys_write` и заменить все вызовы `write` вызовами `sys_write`. Получится следующее:

```
/* greet3.c */
int sys_write(int fd, const void *buf, int size);
```

```
static const char dunno[] = "I don't know how to greet you\n";
static const char hello[] = "Hello, dear ";

static int string_length(const char *s)
{
    int i;
    for(i = 0; s[i]; i++)
        ;
    return i;
}

int main(int argc, char **argv)
{
    if(argc < 2) {
        sys_write(1, dunno, sizeof(dunno)-1);
        return 1;
    }
    sys_write(1, hello, sizeof(hello)-1);
    sys_write(1, argv[1], string_length(argv[1]));
    sys_write(1, "\n", 1);
    return 0;
}
```

Сборку этой программы произведём так:

```
nasm -f elf start.asm
nasm -f elf calls.asm
gcc -Wall -c greet3.c
ld start.o calls.o greet3.o -o greet3
```

Обратим внимание, что для финальной сборки мы вызвали редактор связей вручную, не используя возможности `gcc`; в самом деле, ведь мы отказались от использования библиотеки Си, так что знания компилятора относительно того, где брать эту библиотеку, нам больше не нужны.

Убедившись, что программа `greet3` работает в соответствии с поставленной задачей, мы можем посмотреть, наконец, на размер исполняемого файла, и увидеть, что он занимает *меньше одного килобайта*; на машине автора этих строк размер файла составил 816 байт, что **в семьсот с лишним раз меньше**, чем размер статического исполняемого файла, получавшийся до этого. Но дело даже не в этой экономии — в конце концов, если сравнивать результат не с размером статического «бинарника», а с размером исполняемого файла, получаемого штатным способом с использованием возможностей динамической подгрузки библиотек, разница будет меньше чем в десять раз; тоже, конечно, много, но не настолько.

Гораздо важнее сам принцип: **язык Си позволяет полностью отказаться от возможностей стандартной библиотеки**. Кроме Си, таким свойством — абсолютной независимостью от библиотечного кода, также иногда называемым *zero runtime*[74] — обладают на сегодняшний день только языки ассемблеров; ни один язык высокого уровня не предоставляет такой возможности. В частности, суперпопулярный язык Си++ утратил независимость от библиотек времени исполнения ещё в середине 1980-х годов, когда в него был встроен механизм обработки исключений. **Для того, чтобы обладать свойством zero runtime, язык не должен включать никаких средств, реализация которых на уровне машинного кода столь сложна, чтобы имело смысл выносить её в подпрограмму**; иначе говоря, свойство zero runtime достигается благодаря *отсутствию* в языке определённых возможностей, а не благодаря их наличию.

Именно это свойство — zero runtime — делает Си единственным и безальтернативным кандидатом на роль языка для реализации ядер операционных систем и прошивок для микроконтроллеров. Тем удивительнее, насколько мало людей в мире этот момент осознают; и стократ удивительнее то, что людей, понимающих это, судя по всему, вообще нет среди членов комитетов по стандартизации, которые в течение последних полутора, если не двух десятков лет упорно пытаются «срастить» Си с его стандартной библиотекой, а сам язык утяжеляют возможностями, которые в некоторых компиляторах уже реализуются подпрограммами; иначе говоря, тот бастард, который вышел из-под пера стандартизаторов, более не обладает главным, да и едва ли не единственным достоинством языка Си, зато обладает при этом всеми его недостатками в виде контрлогичной семантики, высокой трудоёмкости работы, опасностей непосредственной работы с указателями и тому подобного.

К сожалению, даже с имеющимися на сегодняшний день реализациями Си есть определённые проблемы. Например, при использовании gcc наша программа могла бы не пройти финальную сборку, при этом компоновщик пожаловался бы на отсутствие каких-то неведомых функций, чьи имена начинаются с двух подчёркиваний. Это означало бы, что в нашей программе мы использовали некую возможность, которую компилятор реализует через вызов «своей собственной» функции. Примером такой возможности служит умножение чисел типа `long long` на 32-битной системе. К счастью, в такой ситуации требуется подключать не обычную «стандартную библиотеку», а сравнительно небольшую библиотеку, содержащую как раз такие вот «хитрые» функции. Для gcc эта библиотека называется `libgcc` и подключается добавлением флага `-lgcc` при вызове линкера:

```
ld start.o calls.o greet3.o -lgcc -o greet3
```

[74] Таким образом подчёркивается нулевой размер библиотеки времени исполнения, то есть той части библиотеки, которая обязательно присутствует во всех исполняемых файлах, создаваемых конкретным компилятором.

Так или иначе, язык Си допускает свойство Zero Runtime, но не все его реализации этим свойством обладают: как видим, gcc в некоторых случаях требует библиотечных функций для реализации возможностей, встроенных в язык. Нам остаётся утешаться тем, что это свойство конкретного компилятора, а не языка в целом.

4.17. Инструментарий программиста

4.17.1. Компилятор gcc

Компиляторы семейства GCC (Gnu Compiler Collection) являются компиляторами командной строки, т. е. все необходимые действия задаются при запуске компилятора и выполняются уже без непосредственного участия пользователя. Это, в частности, позволяет использовать компилятор в командных файлах (скриптах).

Команда `gcc` предназначена для компиляции программ на языке Си, а команда `g++` — на языке Си++. На самом деле используется один и тот же компилятор; оба имени являются обычно символическими ссылками на исполняемый файл компилятора. Поведение компилятора зависит от того, по какому имени его вызвали; прежде всего различие выражается в наборе стандартных библиотек, подключаемых по умолчанию при сборке исполняемого файла. Имена файлов, подлежащих компиляции и сборке, компилятор принимает через параметры командной строки. Кроме того, компилятор воспринимает большое количество разнообразных ключей, часть из которых мы уже знаем; вот несколько более широкий их список:

- `-o` <*filename*> задает имя исполняемого файла, в который будет записан результат компиляции (если не указать этот флаг, результат компиляции будет помещен в файл `a.out`);
- `-Wall` приказывает компилятору выдавать все разумные предупредительные сообщения (warnings); **обязательно всегда используйте этот флаг**, он поможет вам сэкономить немало времени и нервов;
- `-ggdb` и `-g` используются для включения в результирующие файлы отладочной информации, т. е. информации, используемой отладчиком, включая имена переменных и функций, номера строк исходных файлов и т. п.; флаг `-ggdb` снабжает файлы расширенной отладочной информацией, понятной только отладчику `gdb`; если вам кажется, что что-то не в порядке с отладчиком, попробуйте использовать `-g`;
- `-c` указывает компилятору, что результатом должна быть не вся программа, а отдельный ее модуль; в этом случае имя файла для объектного модуля можно не задавать, оно будет сгенерировано автоматически заменой расширения на `.o`;

- `-On` задает уровень оптимизации. `n=0` означает отсутствие оптимизации (значение по умолчанию); для получения более эффективного объектного кода рекомендуется использовать флаг `-O2`; учтите, что оптимизация может затруднить работу с отладчиком;
- `-ansi` приказывает компилятору работать в соответствии со стандартом ANSI C;
- `-pedantic` запрещает все расширения языка, не входящие в выбранный стандарт;
- `-E` останавливает компилятор после проведения стадии макропроцессирования; реультат макропроцессирования выдаётся на стандартный вывод; этот режим работы может быть полезен, если ваши макроопределения повели себя не так, как вы ожидали, и хочется понять, что на самом деле происходит;
- `-S` останавливает процесс компиляции после стадии генерации ассемблерного кода; получившийся текст на языке ассемблера записывается в файл с суффиксом `.S`; к сожалению, генерируемый ассемблерный код имеет синтаксис AT&T, который существенно отличается от изучавшегося нами синтаксиса Intel, но разобраться в нём вполне реально, а посмотреть, что конкретно сделал компилятор из вашего исходника, иногда бывает крайне любопытно, особенно на разных уровнях оптимизации;
- `-D` позволяет с командной строки (т. е. без изменения исходных файлов) определить в программе некий макросимвол; это полезно, если в вашей программе используются директивы условной компиляции и требуется, не изменяя исходных файлов, быстро откомпилировать альтернативную версию программы; например, `-DDEBUG=2` имеет такой же эффект, какой дала бы директива «`#define DEBUG 2`» в начале исходного файла;
- `-include` включает в вашу программу текстовый файл, как если бы он был подключён директивой `#include`;
- `-I` добавляет к рассмотрению директорию, в которой следует искать файлы, подключаемые с помощью `#include`; этот флаг оказывает влияние как на `#include "..."`, так и на `#include <...>`;
- `-isystem` делает примерно то же самое, но добавленная директория рассматривается как «системная»; это сравнительно недавнее новшество в компиляторах семейства `gcc`, которое может, в частности, оказаться важным при использовании флага `-MM`;
- `-l` позволяет подключить к программе библиотеку функций; так, если в вашей программе используются математические функции (`sin`, `exp` и другие), необходимо при компиляции задать ключ `-lmath`; в некоторых вариантах ОС Unix (например, в SunOS/Solaris) при использовании сокетов вам понадобится ключ `-lnsl`;

- `-L` добавляет к рассмотрению директорию, в которой следует искать файлы библиотек, подключённых с помощью `-l`;
- `-MM` анализирует заданные исходные файлы и строит информацию об их взаимозависимостях; о том, как использовать полученную информацию, рассказывается в §4.17.4;
- `-s` предписывает компилятору и линкеру выкинуть из генерируемого кода всё, что оттуда возможно выкинуть; в частности, если использовать `-s` при финальной сборке многомодульной программы, то из итогового исполняемого файла будет исключена отладочная информация — даже из тех модулей, которые были откомпилированы с флагом `-g`.

Например, чтобы откомпилировать программу, написанную на языке Си и целиком находящуюся в файле `prog.c`, следует дать команду

```
gcc -g -Wall prog.c -o prog
```

При этом результат компиляции будет помещен в файл `prog` в текущей директории.

Чтобы откомпилировать программу, состоящую из нескольких модулей `mod1.c`, `mod2.c`, `mod3.c` и главного файла `prog.c`, следует сначала откомпилировать все модули:

```
gcc -g -Wall -c mod1.c
gcc -g -Wall -c mod2.c
gcc -g -Wall -c mod3.c
```

и получить объектные файлы `mod1.o`, `mod2.o`, `mod3.o`. После этого для компиляции основного файла и сборки готовой программы следует дать команду

```
gcc -g -Wall mod1.o mod2.o mod3.o prog.c -o prog
```

4.17.2. Отладчик gdb

С отладчиком `gdb` мы уже встречались, изучая Паскаль (см. т. 1, §2.16.4); исходно `gdb` ориентирован именно на язык Си, так что в определённом смысле работать с ним теперь будет проще, чем когда приходилось отлаживать программы на Паскале. С другой стороны, в первом томе мы рассмотрели далеко не все возможности отладчика. В частности, там не упоминается режим анализа `core`-файла, который полезен в случае аварийного завершения программы; исполняемые файлы, создаваемые компилятором Free Pascal, никогда не создают `core`-файлов, так что этот режим отладчика при изучении Паскаля был для нас бесполезен. Напротив, при работе на Си этот режим анализа `core`-файлов требуется то и дело, так что мы его подробно разберём. Кроме того, мы расскажем про некоторые команды отладчика, оставшиеся

«за кадром» в паскалевской части, и дадим кое-какие дополнительные рекомендации.

Для нормальной работы отладчика нужно, чтобы все модули вашей программы были откомпилированы с ключом `-ggdb` или `-g` (см. §4.17.1). В некоторых случаях работе отладчика может помешать включённый режим оптимизации, так что перед отладкой оптимизацию лучше отключить.

Напомним, что запустить отладчик для программы, исполняемый файл которой называется `prog`, можно командой

```
gdb ./prog
```

Отладчик сообщит свою версию и некоторую другую информацию, после чего выдаст приглашение своей командной строки, обычно выглядящее так: `(gdb)`.

Перечислим основные команды отладчика:

- `run` осуществляет запуск программы в отладочном режиме. Перед запуском целесообразно задать точки останова (см. ниже);
- `start` запускает программу в отладочном режиме, остановив её в начале функции `main()`;
- `list` показывает на экране несколько строк программы, предшествующих текущей и идущих непосредственно после текущей;
- `inspect` позволяет просмотреть значение переменной (в том числе и заданной сложным выражением вроде `*(a[i+1].p)`);
- `backtrace` или `bt` показывает текущее содержимое стека, что позволяет узнать последовательность вызовов функций, приведшую к текущему состоянию программы;
- `frame` позволяет сделать текущим один из фреймов, показанных командой `backtrace`, что дает возможность исследовать значения переменных в этом фрейме и т.п.;
- `step` позволяет выполнить одну строку программы; если в строке содержится вызов функции, текущей строкой станет первая строка этой функции (т. е. процесс трассировки зайдёт внутрь функции);
- `next` подобна команде `step`, с тем отличием, что вход в тела вызываемых функций не производится;
- `until` <номер-строки> позволяет выполнять программу до тех пор, пока текущей не окажется строка с указанным номером;
- `call` позволяет выполнить вызов произвольной функции;
- `set var` <присваивание> позволяет занести заданное значение в переменную; например, `set var i=17` занесёт в переменную `i` в вашей программе значение `17`;
- `break` позволяет задать точку приостановки выполнения программы (breakpoint); точка останова может быть задана именем функции, номером строки в текущем файле или выражением <имя-файла>:<номер-строки>, например `file1.c:73`;

- `disable` <номер-точки-останова> позволяет временно отменить точку останова; отладчик продолжает помнить о ней, но программа в этом месте не останавливается;
- `enable` <номер-точки-останова> снова активирует точку останова, ранее выключенную командой `disable`;
- `ignore` <номер-точки-останова> <число> предписывает отладчику проигнорировать заданную точку останова указанное число раз; например, `ignore 5 200` означает, что точка останова №5 должна быть 200 раз проигнорирована, а при попадании в эту точку в 201-й раз программу следует приостановить;
- `cond` <номер-точки-останова> <условие> задаёт условие, при выполнении которого следует приостановить выполнение программы в данной точке останова; например, `cond 5 i<100` означает, что в точке останова №5 следует остановиться только в случае, если значение переменной `i` в программе будет меньше 100;
- `cont` позволяет продолжить прерванное выполнение программы;
- `help` позволит узнать подробнее об этих и других командах отладчика;
- `quit` завершает работу отладчика (можно также воспользоваться комбинацией клавиш `Ctrl-D`).

Эти команды позволяют организовать уже знакомое нам *выполнение программы в пошаговом режиме*; именно такой режим подразумевается, если мы запустим отладчик простой командой `gdb ./prog`, не указав никаких дополнительных параметров. Если отлаживаемой программе нужно задать аргументы командной строки, это можно сделать одним из двух способов. Во-первых, можно воспользоваться при запуске флагом `--args` примерно так:

```
gdb --args ./prog abra schwabra kadabra
```

(здесь слова `abra`, `schwabra` и `kadabra` выступают в роли аргументов командной строки). Во-вторых, можно запустить отладчик без указания аргументов командной строки, а при запуске программы на пошаговое выполнение указать эти аргументы в команде `run`:

```
(gdb) run abra schwabra kadabra
```

Рассмотрим теперь режим **анализа причин аварийного завершения по core-файлу**. Когда ошибки в программе приводят к её аварийному завершению, операционная система создаёт[75] так называемый core-файл. При этом выдаётся сообщение

```
Segmentation fault (core dumped)
```

[75] Если это не запрещено настройками системы; чтобы узнать это, дайте команду «`ulimit -c`», чтобы отменить запрет, если он установлен — команду «`ulimit -c unlimited`».

Это означает, что в текущей директории аварийно завершённого процесса создан файл с именем `core` (или `prog.core`, где `prog` — имя вашей программы), в который система записала содержимое сегмента данных и стека программы на момент её аварийного завершения. Сегмент кода в core-файл не записывается, поскольку его можно взять из исполняемого файла.

С помощью отладчика gdb можно проанализировать core-файл, узнав, в частности, при выполнении какой строки программы произошла авария, откуда и с какими параметрами была вызвана функция, содержащая эту строку, каковы были значения переменных на момент аварии и т. д. Отладчик в режиме анализа core-файла запускается командой:

```
gdb ./prog prog.core
```

где `prog` — имя исполняемого файла вашей программы, а `prog.core` — имя созданного системой core-файла[76]. Очень важно при этом, чтобы в качестве исполняемого файла выступал именно тот файл, при исполнении которого получен core-файл. Так, если уже после получения core-файла вы перекомпилируете свою программу, анализ core-файла в большинстве случаев приведёт к ошибочным результатам.

Сразу после запуска отладчика в режиме анализа core-файла рекомендуется дать команду `backtrace` или просто `bt`; в большинстве случаев по одной только её выдаче вы уже поймёте, что и где произошло, в остальных случаях вам помогут команды `frame`, `list` и `inspect`. Следует учитывать, что при анализе core-файла ваша программа уже не работает, она аварийно завершилась; как следствие, вам доступны только команды, показывающие текущее состояние программы. Команды, подразумевающие выполнение программы (обычное или пошаговое), такие как `step`, `next`, `cont`, `break` и прочие, в этом режиме применять нельзя.

В заключение напомним о **режиме анализа причин зацикливания**. Если ваш процесс «завис», не торопитесь его убивать. С помощью отладчика можно понять, какой фрагмент кода выполняется в настоящий момент, и проанализировать причины зацикливания (либо узнать, что никакого зацикливания на самом деле не произошло, так тоже бывает). Для этого нужно *присоединить* отладчик к существующему процессу. Определите номер процесса с помощью команды `ps`. Допустим, имя исполняемого файла вашей программы — `prog`, и она выполняется как процесс с номером 12345. Тогда команда запуска отладчика должна выглядеть так:

```
gdb ./prog 12345
```

[76] Подобное имя core-файла характерно для ОС FreeBSD. В ОС Linux файл, как правило, будет называться просто `core`.

При подключении отладчика выполнение программы будет приостановлено, однако вы сможете при необходимости продолжить его командой `cont`. В случае повторного зацикливания можно приказать отладчику вновь приостановить выполнение нажатием комбинации `Ctrl-C`. Остальные команды (за исключением команд запуска, ведь процесс уже запущен) в этом режиме тоже прекрасно работают, то есть вы можете выполнять программу пошагово, устанавливать и убирать точки останова, просматривать и модифицировать значения переменных и т. д.

Если выйти из отладчика, находящегося в этом режиме, процесс продолжит обычное выполнение, если, конечно, за время отладки вы его не убили.

Возможности отладчика `gdb` не ограничиваются перечисленными; на первых порах вам этого, скорее всего, будет достаточно, но в перспективе мы рекомендуем обратиться к документации и другим источникам, чтобы составить впечатление о возможностях `gdb`, которые остались за рамками нашей книги.

4.17.3. Программа valgrind

Утилита `valgrind` позволяет в автоматическом режиме обнаружить в поведении вашей программы такие особенности, которые свидетельствуют о допущенных ошибках и могут рано или поздно привести к тяжёлым последствиям. Многие ошибки, выявляемые при помощи `valgrind`, носят характер труднообнаружимых, поскольку никак себя не проявляют при выполнении ошибочного кода и либо приводят к неправильному функционированию программы позже, либо, что самое неприятное, могут оставаться незамеченными в отладочных запусках, а проявиться (причём, возможно, с тяжёлыми последствиями) уже на этапе активной эксплуатации программы.

Сама утилита `valgrind` представляет собой *интерпретатор машинного кода*, то есть она реализует возможности центрального процессора программно; отлаживаемая программа запускается на выполнение в этом эмуляторе процессора, что позволяет тщательно анализировать каждое её действие. Конечно, программа, выполняемая таким способом, работает во много раз медленнее, чем при обычном запуске, но результат того стоит. Контролируемое исполнение машинного кода в режиме интерпретации позволяет выявить такие заведомо ошибочные действия, как чтение из памяти, в которую программа не занесла никаких значений (использование неинициализированных переменных), обращение к несуществующим элементам массива, использование освобождённых областей динамической памяти, потерю связи с фрагментами динамической памяти и многие другие.

Для примера напишем программу, содержащую заведомую ошибку:

```
#include <stdio.h>
int main()
{
    int x, i;
    for(i = 0; i < 10; i++) {
        if(x < 10)
            printf("First\n");
        else
            printf("Second\n");
        x = i*i;
    }
    return 0;
}
```

Ошибка здесь в том, что в переменной `x` изначально содержится мусор, а осмысленное значение заносится в неё лишь в конце тела цикла; следовательно, на *первой* итерации цикла значение `x` не определено, то есть работа оператора `if` зависит от неопределённого значения. Сохраним эту программу в файл `badcode.c`, откомпилируем и попробуем выполнить её под контролем `valgrind`. Команда запуска будет выглядеть так:

```
valgrind --tool=memcheck ./badcode
```

На самом деле единственный параметр, который мы здесь указали (`--tool=memcheck`), можно было и не указывать, поскольку именно `memcheck` (*memory checking tool*, инструмент для проверки корректности работы с памятью) `valgrind` использует по умолчанию; тем не менее мы рекомендуем всегда указывать используемый инструмент в явном виде. В некоторых версиях `valgrind` «умолчание» может отличаться.

Запустив `valgrind`, мы увидим довольно пространный текст, в котором строки, которые выдал сам `valgrind`, будут перемешаны со строками, печатаемыми нашей программой. Отметим, что всю свою выдачу `valgrind` направляет в поток диагностики (`stderr`), так что можно, например, перенаправить этот поток в файл:

```
valgrind --tool=memcheck ./badcode 2> VG_LOG
```

В этом случае вы увидите на экране только строки, выдаваемые вашей программой, а информацию от `valgrind` найдёте в файле `VG_LOG`. Впрочем, даже если всё будет выдано на экран вперемешку, отличить строки, которые напечатал `valgrind`, очень просто: они начинаются с `==NNNN==` или `--NNNN--`, где `NNNN` — номер процесса. Например, последняя строка, выданная утилитой `valgrind` при запуске нашего примера неправильной программы, будет выглядеть так:

```
==6503== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
```

(здесь номер процесса — 6503). Как видим, найдена одна ошибка; пролистав полученную выдачу вверх, мы обнаружим текст, подобный следующему:

```
==6503== Conditional jump or move depends on uninitialised value(s)
==6503==    at 0x80483FC: main (badcode.c:6)
```

Английская фраза в первой строке переводится как *условный переход или копирование зависит от неинициализированного значения (значений)*; вторая строка сообщает, что ошибка содержится в функции `main`, в строке 6 файла `badcode.c`. Адрес `0x80483FC` позволяет найти соответствующее место в машинном коде, что может быть полезно, например, при использовании отладчика; впрочем, `valgrind` и сам может запустить для нас отладчик, как только обнаружит ошибочные действия нашей программы. Эта возможность активируется дополнительным ключом командной строки:

```
valgrind --tool=memcheck --db-attach=yes ./badcode
```

В этом случае `valgrind`, обнаружив любую ошибку, предложит нам продолжить исследование нашей программы с использованием отладчика. Выглядит это примерно так:

```
==6561== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----
```

Для запуска отладчика нужно нажать `y` или `Y` (от слова *yes*). `Valgrind` ухитряется запустить отладчик так, чтобы из него мы видели только свою программу и совершенно не видели, что эта программа на самом деле выполняется под управлением `valgrind`. В большинстве возможности отладчика позволяют быстро выяснить, что послужило причиной ошибки.

Подробное описание всех основных возможностей `valgrind` заняло бы целую книгу, причём такие книги, собственно говоря, уже есть, некоторые из них даже можно найти в Интернете. Дальнейшее изучение этой программы оставим читателю. Чтобы почувствовать себя увереннее в обращении с `valgrind`, лучше всего будет написать ещё несколько маленьких программ с очевидными ошибками, такими как неосвобождение выделенной динамической памяти, затирание единственного указателя, хранящего адрес того или иного объекта в динамической памяти и т. п., и попробовать их все позапускать под управлением `valgrind`; в этом случае его диагностика не станет для вас сюрпризом при отладке более сложных программ. Обязательно освойте `valgrind`! Это займёт у вас от силы полтора часа, которые окупятся, скорее всего, не одну сотню раз.

4.17.4. Система автоматической сборки (утилита make)

Рано или поздно сложность вашей программы заходит за невидимый рубеж, когда вы просто-таки вынуждены разбить её на отдельные модули (см. §4.11). Практически сразу вы можете заметить, что при сборке такой программы давать все нужные команды вручную довольно неудобно и здесь напрашивается некая автоматизация; именно такую автоматизацию предоставит вам `make`. Кратко говоря, эта утилита позволяет автоматически строить одни файлы на основании других (например, исполняемые файлы на основании исходных текстов программы) в соответствии с заданными правилами. При этом make отслеживает даты последней модификации файлов и производит перестроение только тех целевых файлов, для которых исходные файлы претерпели изменения.

Существует несколько различных версий утилиты make. Изложенное в данном параграфе соответствует варианту Gnu Make; именно этот вариант используется в большинстве дистрибутивов Linux. **При работе с FreeBSD для вызова Gnu Make вместо команды `make` применяется `gmake`.** Если в системе такой команды нет, обратитесь к системному администратору с просьбой её установить.

Правила для утилиты make задаются в файле `Makefile`, который утилита ищет в текущей директории.

Простейший Makefile

Допустим, ваша программа состоит из главного модуля `main.c`, содержащего функцию `main()`, а также из дополнительных модулей `mod1.c` и `mod2.c`, имеющих заголовочные файлы `mod1.h` и `mod2.h`. Для сборки исполняемого файла (назовем его `prog`) нужно дать следующие команды:

```
gcc -g -Wall -c mod1.c -o mod1.o
gcc -g -Wall -c mod2.c -o mod2.o
gcc -g -Wall main.c mod1.o mod2.o -o prog
```

Первые две команды даются для компиляции дополнительных модулей. Полученные в результате файлы `mod1.o` и `mod2.o` используются в третьей команде для сборки исполняемого файла.

Допустим, мы уже произвели компиляцию программы, после чего внесли изменения в файл `mod1.c` и хотим получить исполняемый файл с учетом внесённых изменений. При этом нам надо будет дать только две команды (первую и третью), поскольку перекомпиляции модуля `mod2` не требуется.

Чтобы подобные ситуации отслеживались автоматически, мы можем использовать утилиту `make`. Для этого напишем следующий `Makefile`:

```
mod1.o: mod1.c mod1.h
        gcc -g -Wall -c mod1.c -o mod1.o

mod2.o: mod2.c mod2.h
        gcc -g -Wall -c mod2.c -o mod2.o

prog: main.c mod1.o mod2.o
        gcc -g -Wall main.c mod1.o mod2.o -o prog
```

Поясним, что файл состоит из так называемых *целей* (в нашем случае таких целей три: `mod1.o`, `mod2.o` и `prog`). Описание каждой цели состоит из заголовка и списка команд. Заголовок цели — это **одна** строка, начинающаяся всегда с первой позиции (т. е. перед ней не допускаются пробелы и т. п.). В начале строки пишется имя цели (обычно это просто имя файла, который мы хотим построить). Оставшаяся часть заголовка отделяется от имени цели двоеточием. После двоеточия перечисляется, от каких файлов (или, в более общем случае, от каких целей) зависит построение файла. В данном случае мы указали, что модули зависят от их исходных текстов и заголовочных файлов, а исполняемый файл — от основного исходного файла и от двух объектных файлов.

После строки заголовка идёт список команд (в нашем случае все три списка имеют по одной команде). Строка команды **всегда начинается с символа табуляции**, замена табуляции пробелами недопустима и ведет к ошибке. Утилита `make` считает признаком конца списка команд первую строку, начинающуюся с символа, отличного от табуляции.

Имея в текущей директории вышеописанный Makefile, мы можем для сборки нашей программы дать команду `make prog`.

Переменные

В предыдущем параграфе описан Makefile, в котором можно обнаружить несколько повторяющихся фрагментов. Так, строка параметров компилятора "`-g -Wall`" встречается во всех трех целях. Помимо необходимости повторения одного и того же текста, мы можем столкнуться с проблемами при модификации. Предположим, нам понадобилось задать компилятору режим оптимизации кода (флаг `-O2`). Для этого нам пришлось бы внести совершенно одинаковые изменения в три разные строки файла. В более сложном случае таких строк может понадобиться несколько десятков и даже сотен. Аналогичная проблема встанет, например, если мы захотим произвести сборку другим компилятором.

Решить проблему позволяет введение *make-переменных*. Обозначим имя компилятора C переменной `CC`, а общие параметры компиляции — переменной `CFLAGS`[77]. Тогда наш Makefile можно переписать следующим образом:

[77] Причины выбора именно таких обозначений станут ясны чуть позже.

```
CC = gcc
CFLAGS = -g -Wall -ansi -pedantic

mod1.o: mod1.c mod1.h
        $(CC) $(CFLAGS) -c mod1.c -o mod1.o

mod2.o: mod2.c mod2.h
        $(CC) $(CFLAGS) -c mod2.c -o mod2.o

prog: main.c mod1.o mod2.o
        $(CC) $(CFLAGS) main.c mod1.o mod2.o -o prog
```

Предопределённые переменные и псевдопеременные

Существуют соглашения об именах переменных, причём некоторым переменным утилита `make` присваивает значения сама, если соответствующие значения не заданы явно. Вот некоторые традиционные имена переменных:

- `CC` — команда вызова компилятора языка C;
- `CFLAGS` — параметры для компилятора языка C;
- `CXX` — команда вызова компилятора языка C++;
- `CXXFLAGS` — параметры для компилятора языка C++;
- `CPPFLAGS` — параметры препроцессора (обычно сюда помещают предопределённые макропеременные);
- `LD` — команда вызова системного линкера (редактора связей);
- `MAKE` — команда вызова утилиты `make` со всеми параметрами.

По умолчанию переменные `CC`, `CXX`, `LD` и `MAKE` имеют соответствующие значения, справедливые для данной системы и в данной ситуации. Значения остальных перечисленных переменных по умолчанию пусты. Так, при написании Makefile из предыдущего параграфа мы могли бы пропустить строку, в которой задаётся значение переменной `CC`, в надежде, что соответствующее значение переменная получит без нашей помощи.

Кроме таких переменных общего назначения, в каждой цели могут использоваться так называемые *псевдопеременные*. Перечислим наиболее интересные:

- `$@` — имя текущей цели;
- `$<` — имя первой цели из списка зависимостей;
- `$^` — весь список зависимостей.

С использованием этих переменных мы можем переписать наш `Makefile` следующим образом:

```
CFLAGS = -g -Wall

mod1.o: mod1.c mod1.h
        $(CC) $(CFLAGS) -c $< -o $@
```

```
mod2.o: mod2.c mod2.h
        $(CC) $(CFLAGS) -c $< -o $@

prog: main.c mod1.o mod2.o
        $(CC) $(CFLAGS) $^ -o $@
```

Обобщённые цели

Как можно заметить, в том варианте Makefile, который мы написали в конце предыдущего параграфа, правила для сборки обоих дополнительных модулей оказались совершенно одинаковыми. Можно пойти дальше и задать одно обобщённое правило построения объектного файла для любого модуля, написанного на языке Си, исходный файл которого имеет имя с суффиксом `.c`, а заголовочный файл — имя с суффиксом `.h`:

```
%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@
```

Если теперь задать список дополнительных модулей с помощью переменной, получим следующий вариант Makefile:

```
OBJMODULES = mod1.o mod2.o
CFLAGS = -g -Wall -ansi -pedantic

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@

prog: main.c $(OBJMODULES)
        $(CC) $(CFLAGS) $^ -o $@
```

Теперь для добавления к программе нового модуля нам достаточно добавить имя его объектного файла к значению переменной `OBJMODULES`. Если перечисление модулей через имена объектных файлов представляется неестественным, можно заменить первую строку Makefile следующими двумя строками:

```
SRCMODULES = mod1.c mod2.c
OBJMODULES = $(SRCMODULES:.c=.o)
```

Запись `$(SRCMODULES:.c=.o)` означает, что нужно взять значение переменной `SRCMODULES` и в каждом входящем в это значение слове заменить суффикс `.c` на `.o`.

Псевдоцели

Утилиту `make` можно использовать не только для построения файлов, но и для выполнения произвольных действий. Добавим к нашему файлу две дополнительные цели:

```
run: prog
        ./prog

clean:
        rm -f *.o prog
```

Теперь по команде `make run` утилита `make` произведет, если нужно, сборку нашей программы и запустит её. С помощью же команды `make clean` мы можем очистить рабочую директорию от объектных и исполняемых файлов (например, если нам понадобится произвести сборку программы с нуля).

Такие цели обычно называют *псевдоцелями*, поскольку их имена не обозначают имен создаваемых файлов.

Автоматическое отслеживание зависимостей

В более сложных проектах модули могут использовать заголовочные файлы других модулей, что делает необходимой перекомпиляцию модуля при изменении заголовочного файла, не относящегося к этому модулю. Информацию о том, какой модуль от каких файлов зависит, можно задать вручную, однако в больших программах этот способ приведет к трудностям, поскольку программист при модификации исходных файлов может случайно забыть внести изменения в Makefile.

Разумнее будет поручить отслеживание зависимостей компьютеру. Утилита `make` позволяет наряду с обобщённым правилом указать список зависимостей для построения конкретных модулей. Например:

```
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

mod1.o: mod1.c mod1.h mod2.h mod3.h
```

В этом случае для построения файла `mod1.o` будет использовано обобщённое правило (поскольку никаких команд в цели `mod1.o` мы не указали), но список зависимостей будет использован из цели `mod1.o`.

Списки зависимостей можно построить с помощью компилятора. Получить строку, подобную последней строке вышеприведённого примера, можно, дав команду

```
gcc -MM mod1.c
```

Если результат выполнения такой команды перенаправить в файл, то этот файл можно будет включить в наш Makefile директивой `include`. Эта директива имеет специальную форму со знаком «-», при использовании которой `make` не выдаёт ошибок, если файл не найден. Если использовать для файла зависимостей имя `deps.mk`, директива его включения будет выглядеть так:

```
-include deps.mk
```

Более того, если предусмотреть цель для генерации файла, включаемого такой директивой, например:

```
deps.mk: $(SRCMODULES)
        $(CC) -MM $^ > $@
```

утилита `make`, прежде чем начать построение любых других целей, будет пытаться построить включаемый файл.

Отметим, что такое поведение нежелательно для псевдоцели clean, поскольку для очистки рабочей директории от мусора построение файлов зависимостей не нужно и только отнимает время. Чтобы избежать этого, следует снабдить директиву `-include` условной конструкцией, исключающей эту строку из рассмотрения, если единственной целью, заданной в командной строке, является цель clean. Это делается с помощью директивы `ifneq` и встроенной переменной `MAKECMDGOALS`:

```
ifneq (clean, $(MAKECMDGOALS))
-include deps.mk
endif
```

Окончательно Makefile будет выглядеть так:

```
SRCMODULES = mod1.c mod2.c
OBJMODULES = $(SRCMODULES:.c=.o)
CFLAGS = -g -Wall -ansi -pedantic

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@

prog: main.c $(OBJMODULES)
        $(CC) $(CFLAGS) $^ -o $@

ifneq (clean, $(MAKECMDGOALS))
-include deps.mk
endif

deps.mk: $(SRCMODULES)
        $(CC) -MM $^ > $@

clean:
        rm -f *.o prog
```

Относительно флага gcc -MM следует сделать одно важное замечание. В ранних (вплоть до 3.*) версиях gcc этот флаг означал, что анализировать надо все заголовочные файлы, подключаемые с помощью #include "...", при этом файлы, подключаемые через #include <...>, следует игнорировать. К сожалению, это простое и понятное поведение был «модифицировано» (точнее, просто сломано) под давлением безответственных лиц из стандартизационного комитета, которым не даёт покоя классическая семантика #include, завязанная на файлы файловой системы. В современных версиях gcc флаг -MM не различает виды директив #include; вместо этого gcc -MM игнорирует файлы, включённые из «системных директорий», вне зависимости от того, какой директивой они включаются.

Всё это может оказаться важно, если вы используете возможности сторонней библиотеки, которая в системе не установлена, так что приходится в явном виде указывать пути к её заголовочным файлам. Поскольку эта библиотека не является частью вашей программы, использовать для подключения её заголовочников следует вариант #include <...>; но само по себе это не спасёт вас от генерации зависимостей от файлов, не являющихся частью вашего проекта и, следовательно, не изменяющихся и не требующих внимания со стороны make. Проблема решается использованием флага -isystem вместо классического -I; благодарите стандартизаторов за необходимость всех этих нелепых танцев с бубнами.

4.17.5. Сравнение файлов и наложение изменений

При работе с текстами, особенно если это исходные тексты программ, часто возникает потребность быстро понять, чем друг от друга отличаются две версии одного текстового файла. Выявить такие различия позволяет программа diff — получив две версии одного файла и предполагая, что первая из них — «исходная», а вторая — «изменённая», эта программа покажет, какие строки изменились, а какие были вставлены или удалены. Рассмотрим для примера хорошо знакомую нам программу «Hello, world»:

```
#include <stdio.h>

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

Запишем этот текст в файл hello.c, создадим его копию под именем hello2.c и внесём в новый файл небольшое изменение — например, уберём восклицательный знак в печатаемой фразе; после этого попробуем применить к этим двум файлам программу diff:

```
avst@host:~/work$ diff hello.c hello2.c
```

```
5c5
<     printf("Hello, world!\n");
---
>     printf("Hello, world\n");
avst@host:~/work$
```

Странное «5c5» в начале выдачи означает, что строка № 5 превратилась (*changed*) в строку № 5 — в самом деле, мы ведь не изменили количество строк, только отредактировали одну из них, так что её номер не поменялся. Далее `diff` печатает фрагмент оригинального текста, предваряя каждую его строку символом «<», затем выдаёт разделитель «---» и печатает фрагмент в его изменённой версии, в этот раз используя символ «>». Если бы мы не редактировали существующую строку, а вставили новые, `diff` отразил бы этот факт буквой «a» (от слова *append*); удалению строк соответствует буква «d» (*delete*). Для примера вернём на место восклицательный знак, а после строки, выдающей «Hello, world», вставим ещё две, которые будут печатать фразы «Glad to meet you, World» и «Good bye, World». Выдача `diff` при этом будет выглядеть так:

```
avst@host:~/work$ diff hello.c hello2.c
5a6,7
>     printf("Glad to meet you, world!\n");
>     printf("Good bye, world!\n");
avst@host:~/work$
```

Загадочное «5a6,7» означает, что после пятой строки исходного файла были добавлены строки, которые в результирующем файле стали шестой и седьмой. Если поменять местами исходный файл и результирующий, мы создадим у программы `diff` впечатление, что строки, наоборот, были удалены; об этом она сообщит так:

```
avst@host:~/work$ diff hello2.c hello.c
6,7d5
<     printf("Glad to meet you, world!\n");
<     printf("Good bye, world!\n");
avst@host:~/work$
```

«6,7d5» следует читать как «строки с шестой по седьмую были удалены, а если бы их не удалили, они располагались бы в результирующем файле после его пятой строки». Такой формат выдачи программы `diff` называется «нормальным»; надо сказать, что это лишь один из многих форматов, поддерживаемых программой `diff`.

Другой любопытный формат программа использует при указании флага `-y`; при этом она организует вывод в две колонки, слева — текст оригинального файла, справа — текст изменённого; колонки отделяются друг от друга столбцом символов, обозначающих, что произошло с

данной строкой: если строка не изменилась, программа печатает пробел, если строка была удалена (иначе говоря, она есть только в оригинальном файле) — печатается символ «<», для добавленной строки (имеющейся только в изменённом файле) печатается «>», для изменившейся строки — «|». По умолчанию выдача `diff` в этом режиме имеет ширину 130 позиций, что соответствует длине строки на так называемых «широкоформатных» принтерах, широко применявшихся вплоть до конца 1990-х годов. Вы можете специально для работы с `diff -y` «распахнуть» свой xterm на нужную ширину, либо, если ширина текста вашей программы это позволяет, указать программе `diff`, какой ширины выдачу вы хотели бы видеть, с помощью параметра `-W`; например, «`-W 80`» «загонит» `diff` в рамки стандартной ширины экрана.

Для примера снова возьмём программу «Hello, world», но на этот раз внесём в неё сразу три изменения: уберём пустую строку между `#include` и заголовком функции `main`, уберём восклицательный знак из печатаемой фразы, а после первого вызова `printf` добавим второй, печатающий «Good bye, world». Сравнение этих файлов в исполнении `diff -y` будет выглядеть так:

```
avst@host:~/work$ diff -y -W 80 hello.c hello2.c
#include <stdio.h>                      #include <stdio.h>
                                      <
int main()                              int main()
{                                       {
    printf("Hello, world!\n");        |     printf("Hello, world\n");
                                      >     printf("Good bye, world!\n");
    return 0;                               return 0;
}                                       }
avst@host:~/work$
```

Пожалуй, именно этот формат наиболее удобен для человеческого восприятия. Как ни странно, от программы `diff` чаще требуется прямо противоположное — представить изменения в виде, предназначенном для другой программы. Эта другая программа называется `patch` и предназначена для того, чтобы к копии исходного файла «применить» изменения, ранее обнаруженные с помощью `diff`. Используя связку `diff`/`patch`, программисты могут обмениваться изменениями, не пересылая друг другу весь исходный текст программы. Форматы описания изменений, специально предназначенные для такого применения, сделаны так, что в большинстве случаев программа `patch` может в полностью автоматическом режиме внести требуемые изменения даже в копию исходного файла, «не совсем» тождественную той, которая использовалась при запуске `diff`: скажем, если два программиста редактировали один и тот же файл, но в разных его местах (например, один в начале файла, другой — в конце), и один отправил свои изменения другому в виде выдачи `diff`, то в большинстве случаев второй программист

сможет получить в своей копии исходных текстов изменения, внесённые первым программистом, несмотря на то, что в его копии файла нумерация строк уже совершенно другая.

Достигается это включением в выдачу `diff` так называемого *контекста* — нескольких (чаще всего трёх) неизменившихся строк, предшествующих модифицированному фрагменту, и стольких же — непосредственно следующих за ним. Программа `patch` использует контекст, чтобы привязаться к изменившейся нумерации строк исходного файла. Основных форматов выдачи `diff`, подразумевающих контекст, существует два: собственно «контекстный» (флаг `-c`) и так называемый «унифицированный» (`-u`). Чтобы составить общее впечатление об этих форматах, приведём выдачу `diff` с соответствующими ключами, полученную на тех же версиях файлов, что и в предыдущем примере:

```
avst@host:~/work$ diff -c hello.c hello2.c
*** hello.c     2016-05-26 18:48:44.000000000 +0300
--- hello2.c    2016-05-26 21:56:28.000000000 +0300
***************
*** 1,7 ****
  #include <stdio.h>
-
  int main()
  {
!     printf("Hello, world!\n");
      return 0;
  }
--- 1,7 ----
  #include <stdio.h>
  int main()
  {
!     printf("Hello, world\n");
!     printf("Good bye, world!\n");
      return 0;
  }
avst@host:~/work$ diff -u hello.c hello2.c
--- hello.c     2016-05-26 18:48:44.000000000 +0300
+++ hello2.c    2016-05-26 21:56:28.000000000 +0300
@@ -1,7 +1,7 @@
 #include <stdio.h>
-
 int main()
 {
-    printf("Hello, world!\n");
+    printf("Hello, world\n");
+    printf("Good bye, world!\n");
     return 0;
 }
```

Как можно заметить, унифицированный формат изрядно компактнее; у него имеются и другие достоинства, так что в основном в современных условиях используется именно он. Многие авторы утверждают, что этот формат годится только для версий `diff` и `patch`, созданных Фондом свободного программного обеспечения (GNU diff и GNU patch), но сейчас это уже не так — формат в действительности поддерживается практически всеми версиями этих программ, которые можно встретить.

Поскольку обычно исходные тексты программ представляют собой набор файлов, `diff` позволяет, что вполне естественно, сравнивать не отдельные файлы, а директории целиком, со всем их содержимым. Здесь обычно используют два флага — `-r` (от слова *recursive*), который включает режим сравнения директорий, и `-N` (от слова *new*), который предписывает `diff`'у рассматривать отсутствующие файлы как пустые. Для примера создадим две директории, `old/` и `new/`; в первую положим оригинальный вариант файла `hello.c`, во вторую — версию без восклицательного знака. Кроме того, в первой директории создадим файл `first.txt`, содержащий единственную строку текста «`Early bird gets the worm`», а во второй — файл `second.txt` с фразой «`But the second mouse gets the cheese`». Если применить к этим директориям команду `diff -Nur`, получится следующее:

```
avst@host:~/work$ diff -Nur old new
diff -Nur old/first.txt new/first.txt
--- old/first.txt       2016-05-27 18:07:57.000000000 +0300
+++ new/first.txt       1970-01-01 03:00:00.000000000 +0300
@@ -1 +0,0 @@
-Early bird gets the worm
diff -Nur old/hello.c new/hello.c
--- old/hello.c 2016-05-26 18:48:44.000000000 +0300
+++ new/hello.c 2016-05-27 18:08:47.000000000 +0300
@@ -2,6 +2,6 @@

 int main()
 {
-    printf("Hello, world!\n");
+    printf("Hello, world\n");
     return 0;
 }
diff -Nur old/second.txt new/second.txt
--- old/second.txt      1970-01-01 03:00:00.000000000 +0300
+++ new/second.txt      2016-05-27 18:08:16.000000000 +0300
@@ -0,0 +1 @@
+But the second mouse gets the cheese
avst@host:~/work$
```

Странная дата 1970-01-01 на самом деле означает *отсутствие* даты; действительно, ведь соответствующего файла просто нет. Дата и время в ОС Unix

традиционно представляется как число секунд, прошедшее с полуночи по Гринвичу 1 января 1970 года, так что появление такой даты в выдаче программы соответствует арифметическому нулю; ну а загадочные три часа ночи на самом деле обусловлены часовым поясом Москвы, где проводился этот эксперимент.

Перенаправив вывод программы `diff` в файл, мы получим так называемый «патч» (англ. *patch*, см. также сноску 20 на стр. 43) — список изменений, который можно применить автоматически. Опытные программисты часто, особенно при работе на языке Си, рекомендуют к флагам `-u`, `-r` и `-N` добавить ещё `-p`, который заставит `diff` указывать для каждого изменения, к какой функции языка Си это изменение относится. Если заголовок функции попал в выдаваемый контекст, то `diff` на этом успокаивается, если же заголовок в контекст не попал, программа добавляет его к заголовку изменения (после символов `@@`). Для некоторых других языков программирования `-p` тоже срабатывает. Итоговый набор флагов для получения патча проще всего запомнить в другом порядке: `-Nurp`. Некоторые программисты применяют эту комбинацию, давно забыв, что конкретно значит каждый из четырёх флажков.

Итак, получаем патч (назовём его, например, `old-new.diff`):

```
avst@host:~/work$ diff -Nurp old new > old-new.diff
```

Зайдём теперь в директорию `old` и попробуем на её содержимое наложить изменения, описанные в нашем патче. Для этого мы воспользуемся программой, которая так и называется «`patch`». Файл с изменениями (собственно патч) подаётся ей в поток стандартного ввода, например, простым перенаправлением из файла. Кроме того, в большинстве случаев (в том числе и в нашем) потребуется указать программе `patch` параметр `-p1`; параметр `-p` в общем случае предписывает игнорировать заданное количество элементов имён файлов, или, если говорить совсем точно, программа `patch`, получив флаг `-p` с неким числом N, удаляет из имён файлов N первых символов «`/`» вместе со всеми буквами между ними. В нашем случае нужно убрать из имён файлов в патче имена директорий `old` и `new`. **В подавляющем большинстве случаев программу `patch` запускают именно так — с параметром `-p1`**, что и понятно: как правило, патч-файл получают путём сравнения двух директорий точно так же, как это сделали мы, а имена этих директорий при наложении патча следует, очевидно, проигнорировать. Итак, действуем:

```
avst@host:~/work$ cd old
avst@host:~/work/old$ ls
first.txt  hello.c
avst@host:~/work/old$ patch -p1 < ../old-new.patch
patching file first.txt
patching file hello.c
```

```
patching file second.txt
avst@host:~/work/old$ ls
hello.c  second.txt
avst@host:~/work/old$
```

Несложно убедиться (например, с помощью команды `cat`), что содержимое `hello.c` при этом тоже изменилось (исчез восклицательный знак), а `second.txt` содержит именно тот текст, который соответствующий файл содержал в директории `new`. При желании мы можем вернуть директорию в исходное состояние, применив тот же патч «в обратную сторону»; для этого укажем параметр `-R` (от слова *reverse*):

```
avst@host:~/work/old$ patch -p1 -R < ../old-new.patch
patching file first.txt
patching file hello.c
patching file second.txt
avst@host:~/work/old$ ls
first.txt  hello.c
avst@host:~/work/old$
```

Теперь, если вас попросят «прислать патч», вы будете точно знать, что делать: записать в файл выдачу команды `diff -Nurp` и полученный файл отправить по электронной почте или любым другим способом. Просьба «наложить патч» или «накатить патч» тоже не вызовет больших проблем: берёте пресловутый патч, заходите в директорию, где находятся тексты, которые нужно исправить, и запускаете команду «`patch -p1`», подав ей на стандартный ввод содержимое патча.

Наши эксперименты с программами `diff` и `patch` подсказывают причины возникновения одного очень важного правила, применяемого при оформлении программного кода: **в конце строки в тексте нельзя оставлять незначащие (невидимые) пробелы**. Поставьте себя на место человека, который, анализируя выдачу `diff`, видит, что некая строка заменена *на абсолютно такую же* строку (ведь пробелов-то не видно). О том, что причина на самом деле заключается в незначащих пробелах, можно догадаться не сразу; при этом немало времени уйдёт на внимательное побуквенное сличение двух строк. Между тем случайно оставить невидимый пробел в конце строки очень просто. К примеру, разбивая строку вроде

```
if(str) printf("%s\n", str);
```

на две, мы можем машинально поставить курсор на букву `p` в слове `printf` и нажать на Enter. Пробел, стоявший между скобкой и словом `printf`, так и останется на своём месте, но теперь он станет последним символом в своей строке, то есть превратится именно что в незначащий пробел. Будьте внимательны!

4.17.6. Системы контроля версий

В ходе работы над любой мало-мальски сложной программой рано или поздно возникает вполне понятная боязнь испортить что-нибудь из уже реализованного. В принципе, любое редактирование исходного текста несёт в себе определённый риск, ведь людям свойственно ошибаться, и программисты знают это лучше, чем кто-либо ещё.

Вполне естественное «решение», применяемое начинающими программистами во избежание потерь при редактировании исходных текстов, состоит в том, что все исходные тексты периодически архивируются или просто копируются в отдельную директорию; имена таких архивов и директорий обычно включают в том или ином виде текущую дату. По мере набора негативного опыта программист начинает создавать такие копии всё чаще и чаще; сначала исходники копируются после завершения реализации очередной подсистемы, потом — после каждой новой дописанной функции, затем, возможно, после каждого сеанса редактирования. Иногда за один день появляется два, три архива, а то и больше.

Проблема тут, впрочем, не в количестве архивов: объёмы дисков нынче достаточно большие, исходные тексты программ обычно имеют совсем скромный размер, а при должной аккуратности в именовании можно даже не запутаться, где что лежит. Хуже другое: обнаружив, что в какой-то момент — возможно, достаточно давно — вы испортили какой-то фрагмент своего исходника, вы вряд ли сможете вспомнить дату, когда было внесено «ухудшающее» изменение (программисты называют неприятности подобного рода английским словом *regression*; общепринятого русскоязычного термина пока нет, так что часто так и говорят — ***регрессия***). Поиск нужного «среза» среди нескольких десятков, а то и сотен архивов может отнять много времени и сил; при попытке соединить найденный фрагмент старого исходника с теми полезными изменениями, которые вы внесли позже, часто получается совершенно нежизнеспособный гибрид, поскольку вы можете забыть, для каких целей вносились те или иные правки, упустить какую-то часть связанных между собой изменений и т. п. Несколько десятков чуть-чуть разных версий одного файла — это уже прекрасная возможность заблудиться, когда же проект сам по себе состоит из десятков файлов (модулей), в которые вносятся изменения, как связанные между собой, так и самостоятельные, и всё это существует в десятках копий от разных дат, положение постепенно становится отчаянным, а там и вовсе безнадёжным.

Ситуация резко обостряется, когда над программой работает не один человек, а хотя бы двое или трое, не говоря уже о коллективах из десяти-пятнадцати разработчиков. Уследить за своими собственными действиями ещё как-то можно; но представьте себе поток модификаций,

внесённых в нетривиальный текст программы на протяжении хотя бы года несколькими программистами, *как-то* обменивающимися между собой очередными версиями, и нагромождения архивов исходных текстов, которые каждый из них сделал для себя в соответствии со своей собственной логикой — и вы поймёте, что задача поиска нужного варианта файла (или, хуже того, набора файлов) в таком месиве неразрешима принципиально.

Системы контроля версий позволяют не то чтобы полностью решить все эти проблемы, но хотя бы снизить их остроту. Такие системы на протяжении всего процесса разработки программы позволяют зарегистрировать каждое изменение исходных текстов, снабдить его комментарием, узнать, кто, когда и какие изменения внёс, а при необходимости — получить комплект исходных текстов или какую-то его часть в том состоянии, в котором файлы исходников существовали на определённую дату. Значимые моменты разработки, например, официальный выпуск очередной версии программы или просто «заработавшая» частная возможность, могут быть помечены специальными метками для удобства доступа к ним.

Больше того, все системы контроля версий позволяют поддерживать *одновременно* несколько вариантов истории изменений для одного и того же проекта. Например, часть разработчиков может заниматься одним направлением развития, часть — другим, а некоторые из разработчиков могут помогать то одной, то другой группе своих коллег, и до поры до времени эти группы могут действовать совершенно независимо друг от друга, обращая внимание только на свои изменения; состыковать результаты работы они смогут, когда придёт время — например, когда обе группы достигнут положительных результатов. Такие независимые фрагменты истории изменений называются ***ветками*** (англ. *branches*); они дают возможность проводить эксперименты, не боясь «сломать» основную версию программы. В некоторых случаях изменения, внесённые в отдельной ветке, так никогда и не включаются в основную версию программы; так происходит, если эксперимент признан неудавшимся, а также, например, когда приходится исправлять ошибки в старых версиях программы, причём с выпуском новой версии такие исправления по тем или иным причинам оказываются неактуальны.

При объединении изменений, внесённых разными людьми (в разных ветках или просто при одновременной работе), могут возникать ***конфликты***, возникающие, когда участники разработки изменяют один и тот же фрагмент программы, внося *разные* изменения. Системы контроля версий изрядно облегчают процесс разрешения таких конфликтов. Так, при «ручном» объединении изменений участникам разработки достаточно было бы отредактировать один и тот же файл, чтобы столкнуться со сложностями; системы контроля версий обычно достаточно «интеллектуальны», чтобы понять, что изменения внесены

в *разные места* одного файла, и в этом случае никакого конфликта не возникнет — система сама создаст новую версию файла, содержащую и те, и другие изменения. Конечно, автоматически подобные ситуации могут быть разрешены далеко не всегда, ведь изменения, внесённые разными людьми, могут коснуться одного и того же фрагмента текста; но и в этом случае система контроля версий оказывает существенную помощь, поместив конфликтующие варианты текста в один файл и пометив их специальными знаками, так что программистам остаётся только решить, как должен выглядеть итоговый текст.

База данных, в которой хранится вся история зарегистрированных изменений исходных текстов, называется ***репозиторием***. Обычно репозиторий представляет собой обычное дерево директорий с файлами, так что его можно, например, скопировать или перенести в другое место, заархивировать обычным архиватором, создать резервную копию и т. п. Один из основных принципов работы с репозиторием состоит в том, что **единожды внесённая в репозиторий информация остаётся в нём навсегда**. Вы можете, например, удалить файл из проекта, и система контроля версий этот факт зарегистрирует, но в репозитории при этом останется и сам этот файл, и вся история его изменений, и дата, когда он был удалён. Если бы из репозитория можно было что-то удалять «по-настоящему», то после мы не смогли бы восстановить исходные тексты в состоянии, предшествующем удалению; между тем возможность восстановить *любую* версию наших исходников, существовавшую когда-либо на протяжении всей разработки — это одна из ключевых функций системы контроля версий.

Необратимость внесения информации в репозиторий имеет одно важное следствие: **перед регистрацией того или иного файла в системе контроля версий следует хорошо подумать**. Для текстовых файлов это обычно не создаёт проблем, но в некоторых (достаточно редких) случаях в репозиторий приходится вносить двоичные файлы, например, изображения, которые программа должна выводить на экран. Размеры двоичных файлов могут оказаться существенными, так что наш репозиторий «распухнет» и его станет трудно обслуживать, например, производить резервное копирование. Сказанное не означает, что двоичные файлы вообще нельзя регистрировать в системе; напротив, если вашу программу невозможно собрать без того или иного файла, то этот файл должен быть помещён в репозиторий в обязательном порядке. Просто следует сначала думать, а потом делать; этот принцип, впрочем, можно считать общезначимым.

Системы контроля версий делятся на ***централизованные*** и ***распределённые***. Централизованная система контроля версий подразумевает использование одного репозитория; все участники проекта перед началом сеанса работы получают из репозитория версию исходных тек-

стов, содержащую свежие изменения, внесённые другими участниками, а в конце сеанса отправляют в репозиторий свои собственные изменения.

Распределённая система контроля версий работает по другому принципу. Для *каждой рабочей копии* исходных текстов создаётся собственный репозиторий. Регистрация новых изменений, восстановление старых версий, переключение между ветками и другие штатные операции производятся с использованием локального репозитория; система позволяет синхронизировать содержимое разных репозиториев между собой.

Основной недостаток централизованных систем состоит в том, что для работы с ними вам необходим доступ к центральному репозиторию. Пока вы со всеми вашими коллегами, участвующими в том же проекте, сидите в одном офисе, проблем не возникает. Если программисты физически находятся в разных местах, доступ к репозиторию приходится предоставлять через Интернет, а это уже требует определённых мер по обеспечению информационной безопасности; кроме того, всем участникам проекта для работы требуется постоянный доступ в Интернет, что тоже не всегда возможно. Если использовать распределённую систему контроля версий, то ваш репозиторий будет сопровождать вас везде, где вы захотите, и вы, даже не имея доступа в Интернет, сможете воспользоваться всеми возможностями системы, за исключением получения свежих изменений от других участников и отправки им своих результатов. С другой стороны, распределённые системы устроены сложнее, и их намного сложнее освоить.

Ранние системы контроля версий, такие как RCS или SCCS, в наше время встречаются крайне редко. В течение сравнительно долгого периода — приблизительно с начала 1990-х годов до середины 2000-х — среди разработчиков программного обеспечения была популярна система CVS (*Concurrent Versioning System*), предполагающая использование центрального репозитория. Последний официальный релиз CVS вышел в 2008 году. Новые проекты, предпочитающие централизованный репозиторий, сейчас обычно используют систему Subversion (SVN), которая появилась в 2000 году и активно поддерживается до сих пор. Среди распределённых систем контроля версий наибольшей популярностью пользуется созданная Линусом Торвальдсом в 2005 году программа `git`. К распределённым относятся также Darcs (2003), Mercurial (2005), Bazaar (2005) и многие другие; в целом, судя по всему, распределённая модель организации репозиториев постепенно вытесняет централизованную.

Так или иначе, с наибольшей вероятностью вам могут встретиться именно `git` и, как ни странно, CVS, несмотря на то, что эту систему сегодня следует считать морально устаревшей; эпоха её господства тянулась так долго, что запас проектов, начатых в тот период и пока не сменивших CVS на что-то другое, исчерпается ещё не скоро. Эти две системы мы и рассмотрим, причём начнём с CVS как с более простой.

Система CVS

Все действия, предусмотренные в системе CVS, можно выполнить с помощью одной программы, которая так и называется `cvs`. Для начала вам потребуется создать репозиторий, в качестве которого CVS использует директорию с файлами; эту директорию обычно называют «CVS Root» (*корневая для CVS*). Если вы работаете в одиночку, проще всего создать репозиторий прямо в вашей домашней директории — например, если ваше имя пользователя «`vasya`», а домашняя директория — `/home/vasya`, то в роли репозитория вам вполне подойдёт `/home/vasya/cvsroot`. При работе в группе такое решение неудобно, но об этом позже.

CVS узнаёт о местонахождении репозитория из переменной окружения `CVSROOT`, так что вам придётся эту переменную задать. Если бы нам нужно было дать всего одну команду, мы могли бы задать переменную прямо в ней, например:

```
vasya@host:~$ CVSROOT=/home/vasya/cvsroot echo $CVSROOT
/home/vasya/cvsroot
```

(напомним, что вместо `$CVSROOT` наш командный интерпретатор подставляет значение переменной `CVSROOT`, а команда `echo` печатает свои аргументы). Но в нашем случае одной командой дело не ограничится, так что удобнее будет по меньшей мере обеспечить существование нужной переменной на весь сеанс работы:

```
vasya@host:~$ export CVSROOT=/home/vasya/cvsroot
```

Ещё лучше будет сконфигурировать вашу систему так, чтобы переменная получала своё значение автоматически в каждом вашем сеансе работы. Интерпретатор командной строки `bash`, который мы используем[78], при запуске в качестве «лидера сеанса» пытается найти в вашей домашней директории файл с именем `.bash_profile`, а при любом другом запуске (например, в качестве интерпретатора командно-скриптового файла) — файл `.bashrc`, и если соответствующий файл найден, выполняет все записанные в таком файле команды. Для верности лучше вставить строку

```
export CVSROOT=/home/vasya/cvsroot
```

в оба этих файла. Чтобы проверить, что всё в порядке, правильнее всего будет завершить свой сеанс работы, зайти в систему снова, дать команду «`echo $CVSROOT`» и убедиться, что она печатает то, что нужно.

Создав директорию и установив переменную `CVSROOT`, сообщим системе, что мы хотим начать использование нового репозитория. Это

[78] Если вы используете другой интерпретатор командной строки, обратитесь к его документации.

делается командой «`cvs init`», которая создаст нужные для работы служебные файлы.

В одном репозитории система CVS позволяет размещать произвольное количество проектов (в терминах CVS — «модулей»). Традиционно каждый проект рассматривается как директория с файлами (и, возможно, поддиректориями), и в репозитории система тоже создаёт отдельную директорию для каждого проекта; это полезно знать, если будет нужно перенести проект со всей его историей изменений из одного репозитория в другой.

Для создания в репозитории нового проекта используется команда «`cvs import`». Будьте внимательны с этой командой! Она предполагает, что все файлы, содержащиеся в текущей директории (то есть в той, где вы находитесь, давая эту команду), как раз составляют импортируемый проект, и прилежно вносит их все (со всеми поддиректориями) в репозиторий. Если вы ошибётесь и дадите эту команду, например, в вашей домашней директории, CVS попытается скопировать в репозиторий вообще все ваши файлы, что вряд ли соответствует вашим желаниям. Впрочем, ничего фатального при этом не случится, можно будет просто зайти в ваш репозиторий и удалить оттуда ошибочно созданный проект с помощью команды «`rm -r`». Так или иначе, если вы начинаете проект с нуля, для его внесения в репозиторий вам понадобится пустая директория.

В непустых директориях команду `cvs import` дают, когда нужно начать регистрировать историю изменения исходных текстов, которые уже существуют — например, их написал другой программист, а вы хотите продолжить работу над его проектом. Очевидно, создатели CVS именно эту ситуацию и имели в виду, но в реальности так случается очень редко; чаще в репозитории создают именно пустой проект, в который затем вносят файлы.

Команда `cvs import` требует указания трёх параметров, первый из которых — это имя вашего проекта, под которым он будет известен системе, а два других — так называемые «метка производителя» и «метка выпуска» (*vendor tag* и *release tag*). Скорее всего, по замыслу создателей CVS эти два параметра должны были как-то идентифицировать происхождение исходных текстов, импортируемых в новый проект, но в реальности они *никак и ни для чего не используются*, несмотря на то, что система требует их обязательного указания и даже тщательно проверяет их соответствие неким условиям. Можно написать в этих параметрах любые слова, состоящие из латинских букв, обязательно разные, хотя бы и представляющие собой произвольную белиберду; но правильнее будет, например, в качестве «метки производителя» указать своё имя, а в качестве «метки выпуска» — `start` или что-нибудь подобное. Итак, создаём пустую директорию, заходим в неё и там даём команду `cvs import`:

```
vasya@host:~$ mkdir empty
```

```
vasya@host:~$ cd empty
vasya@host:~/empty$ cvs import myprogram vasya start
```

Программа `cvs` на каждое действие с контролируемыми файлами требует указания комментария, для чего запускает редактор текстов (скорее всего, `vim`, но если в вашей системе по умолчанию настроен другой редактор текстов, то его), давая вам возможность прокомментировать свои действия. Так она поступит и в этот раз. Запуска текстового редактора можно избежать, указав комментарий прямо в командной строке с помощью флажка `-m` (от слова *message*), примерно так:

```
cvs import -m "Starting the project" myprogram vasya start
```

Наконец, приняв команду, система ответит фразой «`No conflicts created by this import`»; теперь в вашем репозитории есть проект, который называется `myprogram`. Интересно, что в репозитории-то он есть, а вот рабочей директории, в которой можно редактировать файлы и отправлять изменения в систему, пока что нет — её нужно создать, получив из репозитория проект. Это делается командой `cvs get`, в данном случае — «`cvs get myprogram`». Учтите, что при этом в текущей директории будет создана директория `myprogram`, которая как раз и станет рабочей директорией для вашего проекта. Имеет смысл давать эту команду в том месте, где вы хотите работать, а не там, где вы делали `cvs import`, например:

```
vasya@host:~/empty$ cd
vasya@host:~$ cd work
vasya@host:~/work$ cvs get myprogram
cvs checkout: Updating myprogram
vasya@host:~/work$ cd myprogram
vasya@host:~/work/myprogram$ ls -F
CVS/
vasya@host:~/work/myprogram$
```

Как видим, в рабочей директории система создала поддиректорию `CVS` для хранения служебной информации. Там же хранится информация о расположении репозитория, так что **когда вы даёте команды `cvs`, находясь в рабочей директории, содержимое переменной `CVSROOT` игнорируется**; система использует тот репозиторий, из которого была изначально получена данная рабочая копия. Учтите это, если будете переносить репозиторий в другое место.

Чтобы получить представление о том, как протекает работа с CVS, создадим в нашей директории файл — например, `hello.c`, содержащий всё ту же программу «Hello, world». О появлении в проекте нового файла нужно сообщить системе с помощью команды `cvs add`:

```
vasya@host:~/work/myprogram$ ls -F
CVS/  hello.c
vasya@host:~/work/myprogram$ cvs add hello.c
cvs add: scheduling file ‘hello.c’ for addition
cvs add: use ‘cvs commit’ to add this file permanently
vasya@host:~/work/myprogram$
```

Система следит за изменениями только тех файлов, о появлении которых вы ей сообщили. Это позволяет избежать случайного попадания в репозиторий всевозможных исполняемых, объектных, временных и прочих файлов, которые не относятся к числу ваших исходных текстов; напомним, что передача любой информации в репозиторий необратима, так что этот момент очень важен.

Отправим наш файл в репозиторий; как можно догадаться из предыдущих сообщений системы, это делается командой `cvs commit`:

```
vasya@host:~/work/myprogram$ cvs commit -m "Wrote hello"
cvs commit: Examining .
/home/vasya/cvsroot/myprogram/hello.c,v  <--  hello.c
initial revision: 1.1
vasya@host:~/work/myprogram$
```

Если не указать параметр `-m`, программа запустит текстовый редактор для ввода комментария; кстати, в некоторых случаях это даже удобнее, поскольку во временном файле, который будет открыт в редакторе, вы увидите заботливо собранную системой информацию о том, какие файлы были добавлены, изменены или удалены; во многих случаях это помогает вспомнить, что́ реально было сделано, и написать более осмысленный комментарий. Сразу дадим довольно очевидный совет: **пишите комментарии по-английски**. CVS не содержит на этот счёт никаких ограничений, но в будущем вам, возможно, захочется показать историю своих изменений кому-то, кто не говорит по-русски.

Типичный сеанс работы с контролируемыми исходными текстами начинается с команды `cvs update`, которая извлекает из репозитория изменения, внесённые из других рабочих копий, и отражает их в ваших рабочих файлах. Полезно помнить, что по умолчанию `cvs update` не создаёт директории, так что если в другой рабочей копии была создана новая директория, вы её у себя не получите; исправить это можно указанием флага `-d`, причём некоторые программисты предпочитают этот флаг указывать при каждом вызове `update`. После приведения своей рабочей копии в соответствие репозиторию вы можете приступить к редактированию файлов и, возможно, созданию новых; каждый создаваемый файл, если он относится к числу ваших исходных текстов, следует зарегистрировать в системе с помощью `cvs add`.

Удаление файлов из проекта производится командой `cvs remove`, причём она отказывается удалять файл, если он всё ещё находится

в вашей рабочей директории, так что сначала файл нужно убрать обычной командой `rm`:

```
vasya@host:~/work/myprogram$ rm file12.c
vasya@host:~/work/myprogram$ cvs remove file12.c
cvs remove: scheduling ‘file12.c’ for removal
cvs remove: use ‘cvs commit’ to remove this file permanently
vasya@host:~/work/myprogram$
```

Здесь вы можете заметить, что действовать так не совсем удобно, поскольку на момент выполнения второй команды файла уже нет и автодополнение его имени не работает, так что приходится его имя набирать целиком. Можно, конечно, воспользоваться менее известными возможностями командного интерпретатора и набрать что-то вроде «`cvs remove !:1`», но помнить такие тонкости не обязательно: эффект двух команд можно совместить, указав флаг `-f`:

```
vasya@host:~/work/myprogram$ cvs remove -f file12.c
cvs remove: scheduling ‘file12.c’ for removal
cvs remove: use ‘cvs commit’ to remove this file permanently
vasya@host:~/work/myprogram$
```

До команды `commit` вы можете передумать и отменить удаление файла, добавив его обратно командой `cvs add`; в этом случае файл будет получен обратно из репозитория, и всё будет выглядеть так, будто никто этот файл не удалял. Впрочем, даже выполнение `commit`, как мы уже знаем, на самом деле лишь пометит в репозитории ваш файл как удалённый, но физически файл удалён не будет; вы в любой момент можете получить его назад и снова внести в проект.

Отметим, что CVS не поддерживает переименование файлов; единственный способ переименовать файл в проекте — это удалить его и добавить обратно под другим именем. CVS при этом не будет знать, что это тот же самый файл, так что история изменений для него начнётся с нуля. Отсутствие средств переименования называют едва ли не самым значительным недостатком CVS; старайтесь продумывать принципы именования файлов так, чтобы их не приходилось переименовывать.

Когда вы достигнете той или иной «логической точки» или просто решите, что пришло время зафиксировать внесённые изменения, дайте команду `cvs commit`. Даже если вы намерены продолжить работу, полезно считать, что отдельно взятый сеанс на этом завершён и начать новый сеанс с самого начала, то есть с команды `cvs update`. Если вы работаете в группе, то за время вашего предыдущего сеанса кто-то из ваших коллег мог отправить в репозиторий свои изменения; чем раньше вы их получите, тем меньше вероятность возникновения конфликтов, при которых система не может совместить имеющиеся изменения автоматически и требует ручного вмешательства.

Команда `cvs update` позволяет получить из репозитория не только актуальную (самую последнюю) версию контролируемых файлов, но и *любую* их версию. Так,

```
cvs update -D "Apr 12, 2015 12:30"
```

приведёт рабочую копию в соответствие тому состоянию исходных текстов, которое было зарегистрировано в репозитории 12 апреля 2015 года в 12:30; команда

```
cvs update -D "15 days ago"
```

восстановит версию пятнадцатидневной давности. Кроме того, той или иной версии исходников можно дать *имя* (*tag* — если угодно, метку или тег) с помощью команды `cvs tag`, например:

```
cvs tag as_released_for_mr_petrov
```

Тегами обычно помечают версии, по той или иной причине выделяющиеся из общего ряда — например, когда начинает работать та или иная возможность или когда делается официальный релиз программы. Версию исходников, помеченную таким именем-тегом, можно в любой момент восстановить, используя имя тега:

```
cvs update -r as_released_for_mr_petrov
```

Полезно помнить, что команда «`cvs update -A`» снова приведёт вашу рабочую копию в соответствие с самыми поздними изменениями, зафиксированными в репозитории.

Для изучения истории изменений и текущего положения вещей очень полезны команды `cvs log` и `cvs status`. Первую обычно запускают с указанием конкретного имени файла, и она выдаёт список всех версий заданного файла вместе с комментариями, которыми сопровождалось каждое зарегистрированное в репозитории изменение этого файла. Вторая команда позволяет узнать, как соотносится с репозиторием та версия файла, которая находится в рабочей директории. Подробно описывать эти команды — задача неблагодарная, будет лучше, если вы сами поэкспериментируете с ними; практически сразу вам всё станет понятно.

Как уже говорилось, системы контроля версий умеют поддерживать одновременно несколько независимых *веток* («бранчей») истории изменений. Если вы собираетесь вносить в проект изменения, в полезности или успешности которых сомневаетесь, либо просто по той или иной причине не хотите, чтобы ваши изменения затрагивали основной ход истории разработки, вы можете потребовать от системы создать отдельную ветку для регистрации ваших изменений; в будущем вы

при необходимости сможете объединить ваши изменения, сделанные в отдельной ветке, с основным ходом изменений или с другой веткой. Для создания ветки используется `cvs tag` с флажком `-b` и указанием имени для новой ветки, примерно так:

```
cvs tag -b doubtful_feature
```

Интересно, что такая команда создаст ветку `doubtful_feature`, но не переведёт вашу рабочую копию в режим работы с этой веткой. Последнее достигается командой `update`:

```
cvs update -r doubtful_feature
```

После этого все изменения, которые вы из вашей рабочей директории отправите в репозиторий с помощью `cvs commit`, будут отражаться именно в ветке `doubtful_feature`. Вернуться в основную ветку можно с помощью уже знакомой нам команды «`cvs update -A`».

Если изменения, зарегистрированные в отдельной ветке, будут признаны достойными включения в основную версию (или в другую ветку, так тоже бывает), вы можете воспользоваться командой `update` с флагом `-j` (от слова *join*) для присоединения изменений из другой ветки к текущей версии. Например, последовательность команд

```
cvs update -A
cvs update -j doubtful_feature
cvs commit
```

зарегистрирует в основной ветке все изменения, ранее внесённые в ветку `doubtful_feature`.

Репозиторий не обязан находиться на той же машине, где расположена ваша рабочая директория. Чаще всего репозиторий располагают на компьютере, к которому можно подключиться удалённо с помощью программы `ssh` (*secure shell*). При этом потребуется задействовать две переменные окружения — уже знакомую нам `CVSROOT`, в которой указать доменное имя машины с репозиторием, имя вашей учётной записи и расположение репозитория, и `CVS_RSH`, в которой задаётся имя программы, используемой для удалённого доступа (`ssh`). Например, если репозиторий расположен на машине `cvsserv.example.com` в директории `/home/cvs/CVSRoot`, а ваша учётная запись на этой машине называется `vasya`, то переменные окружения нужно установить следующим образом:

```
export CVSROOT=vasya@cvsserv.example.com:/home/cvs/CVSRoot
export CVS_RSH=ssh
```

Если ваша локальная учётная запись называется так же, как и запись на машине с репозиторием, то указывать её не обязательно (в нашем примере можно убрать из `CVSROOT` слово `vasya` вместе с символом «`@`»). Порядок работы с CVS при использовании удалённого доступа практически не меняется, за исключением того, что при каждом обращении к серверу система будет требовать ввести пароль. Этого можно избежать, если организовать удалённый доступ с использованием криптографических ключей, но рассказ об этом мы оставим за рамками нашей книги; в Интернете имеется достаточное количество руководств и описаний на эту тему, в том числе на русском.

Возможность доступа к удалённому (то есть находящемуся на другом компьютере) репозиторию открывает путь к организации групповой работы; собственно, для этого системы контроля версий изначально и предназначены. Организовать доступ группы программистов к одному репозиторию технически довольно просто. Для этого на машине, где располагается репозиторий, необходимо завести учётные записи для каждого участника проекта, объединить их в группу (обычно эту группу так и называют — `cvs`), а сам репозиторий сделать доступным для группы на чтение и запись, например, вот так:

```
chown :cvs -R /home/cvs/CVSRoot
chmod ug+rwX -R /home/cvs/CVSRoot
```

Когда над одним проектом работает несколько человек, иногда возникает ситуация *конфликта изменений* — попросту говоря, когда один и тот же фрагмент контролируемых файлов разные участники проекта изменили по-разному. Надо сказать, что CVS в этом плане ведёт себя достаточно стойко; простым внесением изменений в один и тот же файл систему не смутить, для возникновения конфликта нужно, чтобы изменённые фрагменты перекрывались или хотя бы располагались друг от друга на расстоянии менее чем в три строки. Тем не менее рано или поздно вы с конфликтной ситуацией столкнётесь. Между прочим, конфликт не всегда возникает именно при групповой работе: например, при слиянии двух веток (бранчей) система может заявить о конфликте даже когда вы работаете в гордом одиночестве, хотя, конечно, в этом случае организационно ситуация выглядит гораздо проще: конфликт у вас возник *с самим собой*, ну а с собой договориться обычно проще, чем с другими.

Так или иначе, если конфликт уже возник, пугаться не надо: система делает всё возможное, чтобы облегчить выход из создавшегося положения. Обычно конфликт обнаруживается при попытке отправить свои изменения в репозиторий; выглядит это примерно так:

```
cvs commit: Up-to-date check failed for ‘file1.c’
cvs [commit aborted]: correct above errors first!
```

Появление такого сообщения само по себе не означает, что возник конфликт; просто в репозитории в настоящий момент имеется версия файла `file1.c`, которая *свежее* той, в которую вы вносили изменения в своей рабочей копии. Иначе говоря, вы в какой-то момент получили из репозитория версию этого файла и начали её редактировать, а в это время кто-то из ваших коллег этот

файл тоже успел отредактировать и отправить результат в репозиторий. Сообщение «Up-to-date check failed» означает, что вам необходимо запустить cvs update; при этом система постарается совместить в вашем локальном файле file1.c как те изменения, которые внесли в него вы, так и те, которые зарегистрированы в репозитории. В большинстве случаев системе это удастся, так что вам останется только повторить команду cvs commit. И лишь в ситуации, когда в автоматическом режиме система ничего сделать не может, в ответ на команду update вы увидите что-то вроде следующего:

```
Merging differences between 1.1 and 1.2 into file1.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in file1.c
C file1.c
```

Впрочем, даже в этой ситуации всё на самом деле не так уж и плохо, просто система требует ручного объединения изменений. Всё необходимое для этого уже помещено в ваш локальный файл, так что вам остаётся открыть его в редакторе текстов, найти все конфликтные фрагменты (которые отмечены весьма заметными знаками) и решить, какое итоговое содержание оставить. В файле размеченные конфликтующие версии выглядят примерно так:

```
<<<<<<< file1.c
    printf("Hello, world\n");
=======
    printf("Hello, wonderful world!\n");
>>>>>>> 1.2
```

Пометки <<<<<<<, ======== и >>>>>>>> вставлены системой CVS и обозначают соответственно начало конфликтного фрагмента, разделитель (отделяющий вашу версию от той, которая взята из репозитория) и конец фрагмента. В данном случае можно предположить, что изначально строковая константа содержала фразу «Hello, world!», но вы решили из неё убрать восклицательный знак, а ваш коллега в это время добавил в ту же константу слово «wonderful». Технически самый простой способ ликвидации конфликта — попросту выкинуть версию, написанную коллегой, стереть служебные пометки, вставленные системой, и в таком виде сохранить файл и отправить его в репозиторий с помощью cvs commit; но если вы часто будете так делать, рано или поздно коллеги устроят вам «тёмную» в пыльном углу и будут совершенно правы. Правильнее будет при обнаружении конфликта связаться с коллегой, который внёс слово wonderful, и спросить, очень ли это слово ему дорого. Вполне возможно, именно вариант, написанный другим программистом, сильнее приближает проект к успеху, а вас — к премии; упираться на манер барана далеко не всегда осмысленно.

Система git

Обсуждая CVS, мы рассказали практически обо всех основных возможностях этой программы, хотя и очень кратко. К сожалению, с git такой номер не пройдёт: даже самое краткое описание всех её основных

возможностей и приёмов использования займёт добрую сотню страниц. В принципе, то же самое в большей или меньшей степени относится ко всем системам контроля версий, использующим распределённую модель. Не пытаясь объять необъятное, мы покажем, как начать работу с `git` и как в этой системе делать всё то, что мы уже умеем делать с помощью CVS, плюс ещё немного; с остальными возможностями вы либо разберётесь сами, когда они вам понадобятся, либо они вам вообще не понадобятся — так тоже часто бывает, когда речь идёт о достаточно сложных инструментах.

Все основные действия, предусмотренные системой, можно выполнить с помощью одной программы, которая называется `git` и, как и `cvs`, получает команду через свой (обычно первый) параметр командной строки.

Как уже говорилось, распределённые системы контроля версий предполагают поддержку своего репозитория при каждой рабочей директории. Этим обусловлено одно довольно важное концептуальное упрощение в сравнении с централизованными системами: отдельно взятый репозиторий здесь обслуживает только один проект. Теоретически это можно обойти, используя уже знакомые нам *ветки*, но гораздо проще и естественнее будет для каждого проекта завести собственный репозиторий. Заодно отметим, что репозиторий в таких условиях не нуждается в имени и идентифицируется только своим местоположением.

Превратить произвольную директорию в рабочую, имеющую свой репозиторий, можно одной командой «`git init`» без дополнительных параметров. Можно не беспокоиться о наличии в этой директории файлов, в том числе таких, которые мы не собираемся помещать в репозиторий: `git init` в любом случае создаст репозиторий *пустым*. На первый взгляд выполнение команды `init` ничего не меняет в окружающей действительности, но так кажется лишь потому, что в ОС Unix, как мы знаем, файлы, имена которых начинаются с точки, считаются *невидимыми*. Дав сразу после `git init` команду `ls -a`, мы обнаружим, что в текущей директории появилась поддиректория с именем «`.git`»; это и есть созданный репозиторий.

После создания репозитория желательно указать системе, как вас (как автора вносимых изменений) следует идентифицировать. Создатели `git` предлагают использовать для этого имя и адрес электронной почты, например, так:

```
git config user.name "Vasya Ivanov"
git config user.email "vasya.ivanov@example.com"
```

но с таким указанием адреса могут возникнуть определённые проблемы. Программы, предназначенные к свободному распространению, часто публикуют *в виде git-репозитория*, в котором, естественно, все внесённые вами изменения будут помечены с использованием ваших имени

и адреса. Адрес электронной почты, попавший в Интернет, рано или поздно оказывается в базах данных у спамеров со всеми вытекающими последствиями в виде медленно, но верно растущего потока непрошеных писем. С другой стороны, совсем не указывать свой адрес хотя и можно (технически это не создаст больших проблем, разве что в том же проекте обнаружится ваш полный тёзка), но не совсем правильно, ведь кому-то может потребоваться связаться с вами как с автором того или иного фрагмента программы, чтобы обсудить какие-то технические детали. Поэтому правильнее будет, наверное, указать адрес в таком виде, в котором распознать его может только человек, например:

```
git config user.email "vasya.ivanov at example dot com"
```

Если вы работаете с большим количеством репозиториев и не хотите каждый раз настраивать свою идентификацию, можно сэкономить время, сделав параметры «глобальными» с помощью флага `--global`:

```
git config --global user.name "Vasya Ivanov"
git config --global user.email "vasya.ivanov at example dot com"
```

В этом случае `git` запишет значения параметров в файл `.gitconfig` в вашей домашней директории и будет использовать их, если для данного конкретного репозитория не установлены свои собственные значения. Отметим, что идентифицирующую информацию лучше не менять и использовать (во всяком случае, в рамках одного проекта) каждый раз одну и ту же; если вы используете несколько репозиториев в работе над одним проектом — например, на разных машинах — позаботьтесь о том, чтобы имя и адрес были установлены одинаково с точностью до буквы. Нередки ситуации, когда нужно, например, найти все изменения, внесённые в проект конкретным человеком, и в таких случаях возникают определённые сложности, если в разное время этот человек использовал разные (хотя, быть может, и похожие) имена.

Поместить файлы под контроль `git` можно с помощью команды `git add`, параметрами которой служат имена файлов. Отметим, что сходство с аналогичной командой системы CVS тут кажущееся: на самом деле `git add` добавляет не файлы, а *их изменения*, и добавляет их не «в систему вообще», а конкретно в тот набор изменений, который будет зарегистрирован в репозитории ближайшей командой `commit`. Сам этот набор изменений в документации называется «индексом». Иначе говоря, если мы отредактируем файл, сделаем на него `git add`, потом снова отредактируем его и после этого сделаем `git commit`, то *в репозиторий попадёт версия, имевшая место на момент последнего* `git add`, а все изменения, внесённые в файл между командами `add` и `commit`, система проигнорирует, поскольку они не внесены в индекс. Некоторые программисты считают такой подход удобным; другие предпочитают

снабжать команду `commit` флагом `-a`, который сначала добавляет в индекс все изменения, *внесённые в известные системе файлы* (включая, кстати, удаления файлов, которые в норме следует выполнять командой `git rm`). Если с каждым `commit`'ом использовать `-a`, роль `git add` сводится к добавлению в систему новых файлов подобно тому, как мы делали это в CVS.

Пока вы не наберёте достаточно опыта, такая трёхступенчатая схема, состоящая из рабочей директории, индекса и репозитория, может казаться запутанной. «Распутать» её вам поможет, во-первых, команда `git status`, показывающая, какие файлы внесены в индекс (то есть уйдут в репозиторий, если сделать `commit` прямо сейчас), какие изменились, но не внесены в индекс, а какие файлы присутствуют в вашей рабочей директории, но не находятся под контролем `git`; во-вторых, команда `git diff`, показывающая разницу между текущим содержимым отслеживаемых файлов и той их версией, которая помещена в индекс.

Удаление файла производится командой `git rm`; файл не должен содержать изменений ни в индексе, ни в рабочей директории, иначе команда выдаст ошибку (впрочем, это ограничение можно обойти указанием флажка `-f`, но лучше этого не делать). Команда удаляет файл из рабочей директории и помечает его как удалённый в индексе, так что следующий `commit` зарегистрирует удаление в репозитории. Пока вы не сделали `commit`, у вас остаётся возможность передумать; вернуть свой файл обратно вы можете в два шага:

```
git reset HEAD -- file21.c
git checkout -- file21.c
```

Команда `reset` приведёт индекс в соответствие с последней версией, отражённой в репозитории (эта версия обозначается словом `HEAD`); команда `checkout` предназначена для приведения *рабочей директории* в соответствие с той или иной версией из репозитория, если же версию не указать — то из индекса, что и имеет место в данном случае. Эту команду вы можете использовать не только при ошибочном удалении, но и при ошибочном изменении файла.

Система `git` позволяет переименовать файл; это делается командой `git mv` с двумя параметрами — старым и новым именем файла. В рабочей директории файл при этом переименовывается, в индексе помечается как переименованный, а ближайший `commit`, как обычно, фиксирует факт переименования в репозитории; вся история изменений файла при этом сохраняется. Автору доводилось видеть программистов, называвших возможность переименования файлов главной причиной их перехода с CVS на `git`.

Отдельного обсуждения заслуживают уже знакомые нам по системе CVS *ветки* (бранчи). Пользователи централизованных систем контроля версий обычно рассматривают создание ветки как некое серьёзное

действие, которое нуждается в обдумывании, обсуждении с коллегами, возможно, даже в получении разрешения у руководителя разработки. Это можно понять, ведь репозиторий там один на всех, так что участники работы опасаются засорять его лишней информацией. Пользователи распределённых систем воспринимают ветки совершенно иначе, поскольку **ветки по умолчанию локальны в репозитории**, никто не заставляет передавать их в другие репозитории и вообще кому-то показывать. Иными словами, в распределённой системе контроля версий ветки представляют собой ваш личный локальный рабочий инструмент, использование которого остаётся целиком на ваше усмотрение и никак не касается других участников работы, если только вы сами не захотите им предложить использовать ваши ветки. При передаче изменений между репозиториями вы не обязаны передавать в другой репозиторий свои ветки, вы даже можете не сообщать никому об их существовании — но, с другой стороны, при желании вы можете и передать информацию о своих ветках, и создать в удалённых репозиториях ветки под теми же (или другими) именами.

В системе `git` ветка — это просто некое подмножество из всех изменений, которые зарегистрированы в вашем репозитории. Две разные ветки могут существовать абсолютно независимо, но могут и пересекаться, накладываться, сливаться — всё зависит от ваших действий и ваших желаний. Если веток стало слишком много, любую из них можно в любой момент удалить (ещё одна возможность, практически недоступная пользователям того же CVS, в котором каждое изменение привязано к своей ветке), только желательно, чтобы входящие в удаляемую ветку изменения к этому моменту были отражены в какой-то другой ветке; в противном случае, хотя изменения и не исчезнут, доставать их из репозитория придётся по одному, что довольно противно.

Репозиторий с несколькими ветками можно рассматривать как несколько репозиториев в одном; в самом деле, ведь никто не мешает вам создать несколько рабочих директорий, каждую со своим репозиторием, и при необходимости пересылать изменения между ними. Существуют и такие распределённые системы контроля версий, где вообще не поддерживаются ветки, а вместо них предлагается создавать отдельные репозитории (один из самых известных примеров такой системы — Darcs); но `git` ветки поддерживает, и делает это настолько эффективно, что время, затрачиваемое на «смену декораций» при переходе от одной ветки к другой, совершенно неощутимо. Пользователи, перешедшие на `git` с централизованных систем, обычно довольно быстро входят во вкус активной работы с ветками.

При создании нового (пустого) репозитория в нём автоматически заводится ветка с именем `master`; узнать, какие ветки у вас есть и какая из них сейчас активна, можно с помощью команды `git branch`:

```
vasya@host:~/work/prog1$ git branch
* master
vasya@host:~/work/prog1$
```

Символ «*» отмечает активную ветку — ту, с которой вы сейчас работаете. Создать новую ветку можно с помощью той же команды `branch`:

```
vasya@host:~/work/prog1$ git branch doubtful_changes
vasya@host:~/work/prog1$ git branch
  doubtful_changes
* master
vasya@host:~/work/prog1$
```

Как видим, ветка появилась, но не стала активной. Переключиться из одной ветки в другую можно с помощью `git checkout`:

```
vasya@host:~/work/prog1$ git checkout doubtful_changes
Switched to branch 'doubtful_changes'
vasya@host:~/work/prog1$ git branch
* doubtful_changes
  master
vasya@host:~/work/prog1$
```

В принципе эти два действия — создание ветки и переключение в неё — можно совместить, дав команду `checkout` с флажком `-b`; чаще всего так и делают:

```
vasya@host:~/work/prog1$ git checkout -b experiments
Switched to a new branch 'experiments'
vasya@host:~/work/prog1$ git branch
  doubtful_changes
* experiments
  master
vasya@host:~/work/prog1$
```

Как уже говорилось, `git checkout` работает очень быстро — гораздо больше времени тратится на то, чтобы эту команду набрать. В разных ветках у вас могут быть разные файлы и директории, один файл может в разных ветках иметь совершенно разное содержание и т. д.; систему этим не смутить.

В некоторых случаях переключение ветки могло бы привести к потере информации; в таких случаях `git` отказывается выполнять `checkout`. Простейший пример такой ситуации — если у вас в рабочей директории имеются изменённые файлы, *причём их версия в текущей ветке и в той, куда вы хотите перейти, отличается*. В этом случае вы получите примерно такое сообщение:

```
error: You have local changes to 'first.txt'; cannot switch branches.
```

Отметим, что внесённые в файл изменения не станут препятствием к смене ветки, если версия этого файла в обеих ветках одинакова; в этом случае система просто считает, что вы решили зарегистрировать ваши изменения в другой ветке.

Аналогичным образом, хотя и с другой диагностикой, `git` отказывается менять ветку, если в вашем текущем индексе присутствуют изменения, которые вы ещё не зарегистрировали в репозитории. Самый простой способ избежать всех этих проблем — это сначала полностью завершить сеанс работы с текущей веткой, ненужные изменения файлов отменить, нужные — отдать в репозиторий с помощью `commit`, и только после этого переключаться в другую ветку. К сожалению, иногда это оказывается неудобно — например, вы ещё не довели работу до логической точки, когда хотелось бы делать `commit`, но для продолжения работы вам необходимо заглянуть в какой-то файл, который есть только в другой ветке. Даже в такой странной ситуации `git` предоставляет довольно удобное решение: с помощью команды `git stash` вы можете временно сохранить «где-то» (на самом деле в репозитории, но это не важно, поскольку в общую историю такие сохранения не входят) все изменения, которые успели внести в рабочую директорию и индекс. Рабочая директория и индекс при этом очищаются, то есть приводятся в соответствие с последней версией текущей ветки, так что вы можете сменить ветку, поработать в другой ветке, затем перейти назад в исходную ветку и восстановить состояние рабочей директории и индекса, которое раньше сохранили с помощью `stash`; делается это командой «`git stash pop --index`». Если флаг `--index` не указать, восстановлены будут только файлы рабочей директории, а индекс будет считаться пустым; в большинстве случаев это приемлемо, ведь локальные файлы тоже содержат те изменения, которые вы отразили в индексе.

Пожалуй, самое нетривиальное в распределённой модели репозиториев — это обмен информацией между ними. Отметим для начала, что новый репозиторий может быть создан не только пустым, но и как копия другого (уже существующего) репозитория; для этого служит команда `git clone`. Если копируется репозиторий, находящийся на той же машине, достаточно указать путь к нему, то есть полное имя рабочей директории; создать копию репозитория, находящегося на другой машине, к которой имеется доступ через `ssh`, можно примерно так:

```
git clone ssh://vasya@gitserv.example.com/home/vasya/work/prog1
```

Существуют и другие способы указания репозитория, но их мы рассматривать не будем.

Прежде чем двигаться дальше, нам придётся разобраться с двумя концепциями, которые использует `git` при взаимодействии репозиториев между собой. Во-первых, для обмена данными с другим репозиторием

этот другой репозиторий (называемый *удалённым*, англ. *remote*) должен быть зарегистрирован в настройках нашего (локального) репозитория и снабжён именем. Когда мы создаём пустой репозиторий, никаких удалённых репозиториев он изначально не знает; но когда репозиторий создаётся как копия (клон) другого, в создаваемом репозитории его исходный репозиторий (то есть тот, копией которого он является) автоматически регистрируется в качестве удалённого под именем `origin`.

Манипулировать списком зарегистрированных удалённых репозиториев позволяет команда `git remote`; в частности, если дать её без дополнительных параметров, она выдаст список зарегистрированных удалённых репозиториев, но покажет при этом только их имена, что не всегда удобно:

```
vasya@devserv:~/TMP/prog1$ git remote
devserv
origin
vasya@devserv:~/TMP/prog1$
```

Более подробные результаты можно получить, если добавить флажок «`-v`» (от слова *verbose*):

```
vasya@vbook:~/TMP/prog1$ git remote -v
devserv ssh://devserv.example.com/home/vasya/prog1 (fetch)
devserv ssh://devserv.example.com/home/vasya/prog1 (push)
origin  /home/vasya/work/prog1 (fetch)
origin  /home/vasya/work/prog1 (push)
vasya@vbook:~/TMP/prog1$
```

Здесь уже можно предположить, что данный репозиторий был создан как копия репозитория, находящегося на этой же машине и принадлежащего, судя по всему, тому же пользователю `vasya`, а позже здесь был зарегистрирован ещё один удалённый репозиторий, на сей раз находящийся на сервере `devserv.example.com`. Загадочные слова `fetch` и `push` — это команды, позволяющие получить новые изменения из удалённого репозитория и, наоборот, отправить свои изменения «на тот конец». Дело здесь в том, что `git` позволяет использовать разные способы доступа к одному и тому же репозиторию, и здесь в выдаче отражён тот факт, что для каждого из удалённых репозиториев один и тот же адрес используется как для получения информации, так и для её отправки.

Добавить новый удалённый репозиторий в этот список можно с помощью команды `git remote add`; например, `devserv` из предыдущего примера мог быть добавлен так:

```
git remote add devserv ssh://devserv.example.com/home/vasya/prog1
```

Указать, где находится добавляемый репозиторий, можно теми же способами, которые применяются для `git clone` (два из которых мы рассмотрели). Удалённый репозиторий можно переименовать с помощью `git remote rename` и удалить из списка с помощью `git remote rm`; есть и другие команды для манипуляции этим списком.

Вторая концепция, которую нужно понять — это ***удалённые ветки*** (*remote-tracking branches*). Для каждого удалённого репозитория `git`, когда у него есть такая возможность (то есть при получении «оттуда» информации), создаёт (локально) ветки, соответствующие состоянию веток удалённого репозитория. Эти ветки имеют «хитрые» имена, записываемые через слэш, что-то вроде «`remotes/origin/master`» или «`remotes/devserv/fixup21`»; первый вариант соответствует ветке `master` из удалённого репозитория `origin`, второй — ветке `fixup21` из репозитория `devserv`. Удалённые ветки не показываются командой `git branch`, если её об этом специально не попросить, а попросить её можно флажком `-a` (от слова *all*):

```
vasya@host:~/TMP/prog1$ git branch -a
* master
  remotes/devserv/HEAD -> devserv/master
  remotes/devserv/fixup21
  remotes/devserv/fixup15
  remotes/devserv/master
  remotes/origin/HEAD -> origin/experiments
  remotes/origin/doubtful_changes
  remotes/origin/experiments
  remotes/origin/master
vasya@host:~/work/prog1$
```

Как ни странно, в ходе обычной работы слово `remotes` в названии удалённых веток не пишут, сокращая название до имени репозитория и имени ветки в нём, записанных через слэш; зачем там это слово вообще появилось — вопрос из области философии.

Удалённые ветки отличаются от обычных тем, что в них нельзя ничего регистрировать, то есть делать в них `commit`. К удалённой ветке можно применить `git checkout`, чтобы посмотреть, как выглядит рабочая директория в той или иной ветке удалённого репозитория, только при этом лучше не пытаться вносить изменения в файлы[79] и регистрировать их в репозитории, поскольку на самом деле *текущей* удалённая ветка не становится, вместо этого у вас вообще исчезает

[79] Конечно, никто не может помешать вам редактировать файлы. Регистрация внесённых изменений при этом теоретически возможна, но имеет целый ряд особенностей, подробное обсуждение которых заняло бы много места. Поскольку всё равно так обычно не делают, мы эту возможность обсуждать не будем. Если очень интересно, поищите нужные материалы самостоятельно по ключевой фразе «`detached HEAD state`».

текущая ветка. Впрочем, основная функция удалённых веток скорее идентификационная, то есть состоит в том, чтобы мы могли при необходимости сослаться на набор изменений, входящих в ту или иную ветку удалённого репозитория.

Ещё одно понятие, создающее изрядную путаницу, по-английски звучит как *tracking branch*, что можно перевести примерно как «отслеживающая ветка». Из-за слова *tracking* эти ветки часто путают с удалёнными ветками, которые мы только что обсуждали, поскольку в их английском названии это слово тоже есть[80]. На самом деле «отслеживающие ветки» никого и ничего не отслеживают. Всё их отличие от обычных веток состоит в том, что для такой ветки в конфигурации репозитория указана некая удалённая ветка, связанная с данной, или, если угодно, «вышестоящая» (англ. *upstream*) для данной. Проявляется эта связь исключительно в том, что в некоторых командах, подразумевающих обмен данными с удалёнными репозиториями, можно опустить имя ветки, которое должно использоваться для «того конца».

При клонировании репозитория локально создаётся, как уже говорилось, всего одна ветка, имя которой соответствует имени *активной* ветки клонируемого оригинала (чаще всего это ветка `master`). Введя понятия удалённых и отслеживающих веток, мы к этому утверждению можем добавить, что единственная создаваемая ветка сразу же по умолчанию становится «отслеживающей» для той ветки, которая послужила ей оригиналом, то есть чаще всего (а именно — если в оригинальном репозитории активной была ветка `master`), при клонировании получается ветка `master`, для которой `origin/master` выступает в качестве «вышестоящей». Можно создать локальные аналоги и для других веток исходного репозитория, например, так:

```
vasya@host:~/TMP/prog1$ git branch --track experiments origin/experiments
Branch experiments set up to track remote branch experiments from origin.
vasya@host:~/TMP/prog1$
```

Узнать, для каких веток заданы «вышестоящие» ветки (и какие), можно с помощью команды «`git branch -vv`» (две буквы `v` означают «очень подробно»).

Для получения свежей информации из удалённых репозиториев служит команда `git fetch`. У этой команды много всевозможных флажков, параметров и опций, но в большинстве случаев достаточно просто указать имя удалённого репозитория (дать команду вроде «`git fetch origin`») или флаг `--all`, чтобы скачать всё новое из *всех* удалённых репозиториев, указанных в локальных настройках. Важно понимать, что **команда** `git fetch` **только скачивает сведения о**

[80] Мы перевели термин «*remote-tracking branch*» словосочетанием «удалённая ветка»; этот перевод, разумеется, неточен, поскольку вообще никак не учитывает слово «*tracking*», но более точный перевод, который при этом не был бы до крайности косноязычным, нам ни придумать, ни найти не удалось.

свежих изменениях из удалённых репозиториев в локальный и больше ничего не делает. Она не меняет ни файлы в вашей рабочей директории, ни содержимое вашего индекса, ни содержимое локальных веток, в том числе «отслеживающих». Иначе говоря, в вашем репозитории появляется информация об изменениях, зарегистрированных в других репозиториях, но вы вполне можете эту информацию *не заметить*. Конечно, `git fetch` меняет состояние *удалённых веток* (то есть веток с именами наподобие `origin/master` или `devserv/fixup21`), но на эти ветки в ходе обычной работы вполне можно не обращать внимания.

Чтобы увидеть изменения, «приехавшие» из других репозиториев, потребуется в явном виде затребовать их внесение в рабочую копию; соответствующее действие нам уже знакомо по описанию системы CVS и называется *слиянием* (англ. *merge*). Команда, выполняющая это действие, так и называется `git merge`. Возможности этой команды довольно широки, но в подавляющем большинстве случаев используют её ровно одним способом. Для начала нужно убедиться в том, что ни в вашей рабочей директории, ни в индексе нет изменений, которые ещё не зарегистрированы в репозитории; если такие изменения есть (`git status` сообщает об их наличии), их следует либо зарегистрировать (сделать `commit`), либо отменить. Затем, если вы находитесь не в той ветке, в которую собираетесь «вливать» изменения, нужно сделать соответствующий `git checkout`, чтобы нужная ветка стала активной (текущей). После этого вы можете присоединить к текущей ветке изменения, имеющиеся в другой ветке, *в том числе удалённой*. Выглядит это примерно так:

```
vasya@host:~/TMP/prog1$ git fetch --all
Fetching origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From /home/vasya/work/prog1
   f23c5f2..908687f  experiments -> origin/experiments
vasya@host:~/TMP/prog1$ git branch
* experiments
  master
vasya@host:~/TMP/prog1$ git merge origin/experiments
Updating f23c5f2..908687f
Fast-forward
 hello.c    |    2 +-
 second.txt |    1 +
 2 files changed, 2 insertions(+), 1 deletions(-)
vasya@host:~/TMP/prog1$
```

Эта процедура может показаться довольно сложной для очевидно рутинного действия, и здесь как раз может пригодиться рассмотренное ранее свойство *отслеживающих* веток. Если ветка, в которой мы работаем, настроена как отслеживающая ту удалённую ветку, из которой мы хотим получить изменения, то мы можем не делать ни `git fetch`, ни `git merge`, для которой необходимо довольно громоздкое указание имени удалённой ветки; со всей этой работой справится одна команда `git pull`. Как водится, эта команда позволяет указывать всевозможные сложные комбинации параметров, чтобы потом просто вызвать `git fetch` и `git merge`, но когда такие параметры нужны, пожалуй, проще будет вызвать эти две команды в явном виде, нежели вникать, как именно их вызовет `git pull`; совсем другое дело — `git pull` без параметров при условии, что текущая ветка настроена как отслеживающая. В этом случае `git pull` для начала затребует свежие данные из репозитория, в котором находится вышестоящая ветка, а затем вызовет `merge` для внесения изменений в нашу текущую ветку и заодно в рабочую копию.

Для привыкших к системе CVS отметим, что `git pull` делает именно то, чего вы привыкли ожидать от `cvs update` — отражает в рабочей копии изменения, которые кто-то другой внёс в одну с вами ветку.

Команда `git push` позволяет передать данные в обратную сторону, то есть отправить изменения, накопленные в локальном репозитории, в один из удалённых репозиториев. Проще всего пользоваться этой командой, если вы работаете в отслеживающей ветке и хотите отправить изменения в вышестоящую ветку; в этом случае `git push` можно запустить без параметров. Впрочем, полезно знать, что у вас есть возможность отправить текущую версию любой из локальных веток в любой из удалённых репозиториев, причём «на том конце» воспользоваться любой из имеющихся веток или создать новую. Например, команда

```
git push boss_computer crazy_experiment:stable_version
```

отправит содержимое вашей локальной ветки с именем `crazy_experiment` в удалённый репозиторий `boss_computer`, где это содержимое будет отражено в ветке, имеющей имя `stable_version`; если такой ветки там пока нет, она будет создана.

С командой `push` связано одно довольно неочевидное ограничение: `git` по умолчанию отказывается выполнять `push` в ветку, которая на удалённом репозитории является текущей. Обусловлено это тем, что в результате такого `push` версия файлов, находящихся в рабочей директории удалённого репозитория, окажется устаревшей в сравнении с тем, что зарегистрировано в самом репозитории, а это может повлечь довольно странные эффекты (особенно странные для человека, который не знает или не понимает, что́ в действительности произошло). Решений здесь возможно несколько. Если у вас есть два репозитория и вы из

одного в другой часто передаёте информацию с помощью `push`, можно в этом втором репозитории завести специальную неиспользуемую ветку, назвав её, например, `dummy`, `fake`, `parking` или как-то ещё, и *всегда оставлять репозиторий переключённым на эту ветку.* С тем же успехом можно оставлять репозиторий в состоянии «detached HEAD», например, с помощью команды `git checkout HEAD^0` (к сожалению, у нас нет возможности подробно объяснить, что и почему при этом происходит; если будете так делать, постарайтесь не забыть, что данный репозиторий находится в «безголовом» состоянии, и перед очередным сеансом работы обязательно верните его в одну из существующих веток).

Более «штатное» решение состоит в том, чтобы в настройках репозитория, куда делается `push`, в явном виде разрешить принимать изменения в текущую (активную) ветку. Делается это так:

```
git config receive.denyCurrentBranch warn
```

После этого отправка изменений в активную ветку такого репозитория вызовет предупреждение, но благополучно сработает. Предупреждение будет выглядеть примерно так:

```
remote: warning: updating the current branch
```

Увидев его, не забудьте исправить некорректное положение вещей; сделать это довольно просто: нужно зайти в рабочую директорию удалённого репозитория и дать там страшновато выглядящую команду «`git reset --hard HEAD`». На самом деле ничего особо страшного в этой команде нет, она предписывает `git`'у привести файлы в рабочей директории в соответствие с состоянием `HEAD`, не обращая внимания на любые несоответствия. Впрочем, здесь тоже есть о чём беспокоиться: необходимо точно знать, что в рабочей директории не осталось изменений, не зарегистрированных в репозитории, но при этом нужных — их-то вы, очевидно, потеряете.

«Совсем штатное» решение — завести для обмена данными репозиторий специального типа, так называемый `bare repository`, который, кратко говоря, представляет собой репозиторий без рабочей директории. Этот подход тоже имеет свои ограничения, к тому же такие репозитории — это одна из тех возможностей `git`, которые хотелось бы оставить за рамками обсуждения в нашей книге. Так или иначе, возможны достаточно сильно различающиеся варианты построения рабочего процесса с использованием `git`, и от избранного варианта зависит, как правильнее будет решить техническую заминку с `push`.

Следует отметить, что для выполнения команды `push` у вас должен быть доступ к удалённому репозиторию на запись. Если это ваш собственный репозиторий, проблем с записью в него, скорее всего, не будет, но у вас вряд ли получится отправить свои изменения в репозиторий, принадлежащий кому-то другому, пусть даже это ваш коллега по команде разработки и вообще лучший

друг. Теоретически это можно обойти, если установить на файлы и директории, составляющие репозиторий, соответствующие права доступа, но, как показывает опыт, работать в репозитории, в который в любой момент кто-то из коллег может что-то залить, оказывается довольно неудобно.

Некоторые коллективы решают эту проблему, создав один центральный «разделяемый» репозиторий (обычно как раз `bare`, то есть не имеющий своей рабочей директории), который используется всеми участниками разработки для обмена данными между собой. Делается это примерно так же, как при создании централизованного репозитория для CVS и других централизованных систем (см. описание на стр. 474). Обычно в таких случаях участники проекта создают свои рабочие репозитории в виде клонов центрального репозитория, а рабочий процесс в целом начинает напоминать работу с централизованной системой контроля версий.

Мы возьмём на себя смелость порекомендовать другую схему организации рабочего процесса, в которой участникам разработки нет нужды отправлять изменения в *чужие* репозитории. Для реализации этой схемы потребуется Unix-машина, доступная по `ssh` всем участникам разработки; иметь такую машину в любом случае очень удобно и, как мы видели, при использовании схемы с центральным репозиторием наличие такой системы также весьма желательно. Здесь и далее мы будем называть эту машину «сервером разработки».

Каждый участник работы использует по меньшей мере *два* репозитория: один — на сервере, второй — на своей рабочей машине; здесь важно то, что именно он сам выступает в роли *владельца* обоих своих репозиториев, то есть репозиторий на сервере он заводит под своей учётной записью и в своей домашней директории. Конечно, ограничиваться этим разработчик не обязан, репозиториев на своих машинах он может наделать сколько угодно, да и на сервере, в принципе, может создать их больше одного (хотя от этого польза уже весьма сомнительна).

Репозиторий на сервере заводится как клон какого-то другого репозитория на сервере — например, как клон репозитория, принадлежащего руководителю разработки. Права на репозитории выставляются таким образом, чтобы сделать их доступными для чтения всем участникам разработки (например, можно объединить их в одну группу и дать права на чтение этой группе); доступ на запись в свой репозиторий каждый разработчик оставляет только для себя. Репозиторий на личной рабочей машине заводится клонированием *своего* репозитория с сервера.

В репозитории, расположенном на личной машине, создаётся список удалённых репозиториев, в который включаются репозитории других разработчиков, находящиеся на сервере. Свои результаты работы каждый разработчик отправляет с помощью `git push` в *свой* репозиторий на сервере, откуда их могут забрать другие разработчики. Как-то специально организовывать обмен данными между репозиториями на сервере не обязательно, поскольку все изменения в любом случае в итоге попадут в каждый из них; в самом деле, «втянув» изменения из серверных репозиториев, принадлежащих коллегам, в свой рабочий репозиторий, разработчик рано или поздно решит отправить *свои* изменения в свой репозиторий на сервере, но при этом туда будут переданы также и те изменения, которые были ранее загружены из репозиториев других разработчиков. Впрочем, поскольку в такой схеме репозитории на сервере вполне могут быть обычны-

ми, то есть иметь свои рабочие директории, при желании разработчики могут «таскать» изменения друг у друга также и на сервере.

Несомненное достоинство этой схемы состоит в отсутствии необходимости давать права записи в один и тот же репозиторий всем разработчикам сразу. Содержимое разделяемой директории всегда находится под угрозой порчи, пусть и непреднамеренной; в нашей схеме каждый может испортить только свой собственный репозиторий, что далеко не столь страшно, поскольку вся информация, содержащаяся в нём, также имеется в репозиториях других разработчиков, причём как в серверных, так и в рабочих.

В заключение обсуждения групповой работы с `git` дадим один, возможно, неожиданный совет. В Интернете можно найти множество сайтов, предлагающих бесплатный сервис хостинга git-репозиториев — BitBucket, GitHub, SourceForge и многие другие. Несмотря на их популярность, мы возьмём на себя смелость рекомендовать воздержаться от использования этих и любых других подобных сервисов, насколько это возможно. Если у вас нет возможности получить в своё распоряжение unix-машину с глобально-видимым ip-адресом, арендуйте для своей команды VPS (виртуальный частный сервер); стоимость аренды VPS начинается от двух-трёх долларов в месяц, а обслуживать такой сервер может не один десяток проектов, то есть вы можете пригласить к себе всех знакомых, занимающихся разработкой программ, сколько бы их у вас ни было, и мощности одного сервера хватит на всех.

Если такое предостережение покажется вам странным, поищите в Интернете статью автора книги, озаглавленную «Осторожно, частный сервис»; там вы найдёте подробное изложение причин, по которым следует относиться с осторожностью к любым «бесплатным сервисам», предлагаемым в Интернете.

Рассмотренных возможностей `git` вам хватит, чтобы начать работу с этой системой, но для серьёзного использования желательно знать о ней гораздо больше. Так, мы не рассмотрели ни средства просмотра журналов изменений, ни разнообразные способы идентификации отдельных «коммитов», ни загадочную на первый взгляд, но очень важную и нужную команду `git rebase`; остались за кадром и многие другие возможности. К счастью, в Интернете в изобилии представлены всевозможные руководства, обучающие тексты, примеры и прочие материалы, посвящённые системе `git`; не пренебрегайте возможностями самостоятельного изучения!

Литература

[1] Э. Танненбаум. Архитектура компьютера. 4-е издание. СПб.: Питер, 2003.

[2] Керниган Б., Ритчи Д. Язык программирования Си. 3-е издание. СПб.: Невский диалект, 2000.

[3] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling. Numerical Recipes in C: The Art of Scientific Computing. New York: Cambridge University Press, 1992 (2nd ed.)

Предметный указатель

ГЛАВНЫЕ СПОНСОРЫ ПРОЕКТА

список наиболее крупных пожертвований

I: 45763 (19972+25791), *спонсор пожелал остаться неизвестным*

II: 41500, *спонсор пожелал остаться неизвестным*

III: 25000, **unDEFER**

IV: 21600, *спонсор пожелал остаться неизвестным*

V: 15000 (3000+7000+5000), *спонсор пожелал остаться неизвестным*

VI: 12000 (2000+10000), **Masutacu**

VII: 10524 (5262+5262), **os80**

VIII: 10000, **Аня «canja» Ф.**

IX: 10000 (2000+8000), **Сергей Сетченков**

X: 8080, **Дергачёв Борис Николаевич**

XI: 8072, *спонсор пожелал остаться неизвестным*

XII: 8000, **Смирнов Денис**

СТОЛЯРОВ Андрей Викторович

ПРОГРАММИРОВАНИЕ: ВВЕДЕНИЕ В ПРОФЕССИЮ
II: НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ
Учебно-методическое издание

В оформлении обложки использован рисунок
Олеси Фроловой
Дизайн обложки Елены Доменновой
Корректор Екатерина Ясиницкая

Напечатано с готового оригинал-макета

Подписано в печать 09.06.2016 г.
Формат 60x90 1/16. Усл.печ.л. 31. Тираж 300 экз. Изд. № 158.

Издательство ООО “МАКС Пресс”
Лицензия ИД № 00510 от 01.12.99 г.

11992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891

Отпечатано в ППП «Типография «Наука»
121099, Москва, Шубинский пер., 6
Заказ № 999

Андрей Викторович Столяров (род. 1974) — кандидат физико-математических наук, кандидат философских наук, доцент; работает на кафедре алгоритмических языков факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова.

Второй том учебника А. В. Столярова посвящён языку ассемблера и языку Си, с которыми традиционно связывают термин «низкоуровневое программирование». Кроме того, в книге даются сведения об инструментах программиста, включая средства отладки, утилиту автоматизированной сборки и системы контроля версий.

http://www.stolyarov.info