Стефан Кочан
Патрик Вуд

4-е издание

Программирование командных оболочек в Unix, Linux и OS X

Программирование командных оболочек в Unix, Linux и OS X

4-е издание

Shell Programming in Unix, Linux and OS X

Fourth Edition

Stephen G. Kochen
Patrick Wood

Addison
Wesley

800 East 96th Street,
Indianapolis, Indiana 46240

Программирование командных оболочек в Unix, Linux и OS X

4-е издание

Стефан Кочан и Патрик Вуд

Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75
К75
УДК 681.3.07

Компьютерное издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Кочан, Стефан, **Вуд**, Патрик.
К75 Программирование командных оболочек в Unix, Linux и OS X, 4-е изд. : Пер. с англ. — СпБ. : ООО "Альфа-книга", 2017. — 432 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909445-3-4 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Стефан Кочан, Патрик Вуд

Программирование командных оболочек в Unix, Linux и OS X
4-е издание

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

Подписано в печать 12.07.2017. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 34,8. Уч.-изд. л. 17,8.
Тираж 300 экз. Заказ № 4866.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9909445-3-4 (рус.)

ISBN 978-0-13-449600-9 (англ.)

Оглавление

Содержание

Об авторах

Стефан Кочан — автор нескольких популярных книг по ОС Unix и языку C, включая *Programming in C*, *Programming in Objective-C*, *Topics in C Programming* и *Exploring the Unix System*. Прежде он работал консультантом по программному обеспечению в компании T&T Bell Laboratories, где составил и вел курсы по Unix и программированию на C.

Патрик Вуд работает техническим директором в филиале компании Electronics for Imaging, находящемся в штате Нью-Джерси. Он входил в состав инженерно-технического персонала компании Bell Laboratories, где и познакомился со Стефаном Кочаном в 1985 году. Совместно они основали консультационную компанию Pipeline Associates, Inc. по ОС Unix, где Патрик занимал пост вице-президента. Кроме того, они совместно написали ряд книг, в том числе *Exploring the Unix System*, *Unix System Security*, *Topics in C Programming* и *Unix Shell Programming*.

Введение

Не секрет, что семейство Unix и Unix-подобных операционных систем стало за последние несколько десятилетий самым распространенным и широко употребляемым в современной вычислительной технике. Для программистов, пользующихся Unix многие годы, в этом нет ничего удивительного, ведь система Unix предоставляет изящную и эффективную среду для программирования. Именно такую среду и стремились создать Деннис Ритчи (Dennis Ritchie) и Кен Томпсон (Ken Thompson), разрабатывая Unix в компании Bell Laboratories в конце 1960-х годов.

Примечание

На протяжении всей книги термином *Unix*, как правило, обозначается обширное семейство основанных на Unix операционных систем (ОС), включая такие ОС типа Unix, как Solaris, а также Unix-подобные ОС вроде Linux и Mac OS X.

К числу самых сильных сторон системы Unix относится обширный ряд программ. В стандартной распространяемой версии ОС Unix насчитывается более 200 основных команд, число которых в обычной версии Linux достигает 700–1000! Эти команды, называемые также *инструментами*, выполняют самые разные действия: от подсчета количества строк в файле до отправки электронной почты и отображения календаря на любой год. Истинный потенциал системы Unix кроется не только в большом наборе команд, но и в изяществе и простоте, с которыми можно их сочетать для решения намного более сложных задач.

Стандартным пользовательским интерфейсом в Unix служит командная строка, которая, по существу, является командным процессором или *оболочкой* — программой, выполняющей роль посредника между пользователем и более низкими уровнями самой системы, образующими ее *ядро*. Проще говоря, оболочка — это программа, читающая вводимые пользователем команды и преобразующая их в форму, более понятную для системы. В нее входят также программные конструкции, позволяющие делать выбор, организовывать циклы и сохранять значения в переменных.

Стандартная оболочка, распространяемая вместе с системами Unix, происходит от версии, которая распространялась компанией AT&T и постепенно развилась в версию, первоначально написанную Стивеном Борном (Stephen Bourne) из компании Bell Labs. С тех пор в IEEE были созданы стандарты на основе оболочки Борна и ряда других более современных оболочек. Текущая версия (на момент написания данной книги) стандарта на оболочку называлась Shell and Utilities (Оболочка и утилиты), том 1003.1-2001 стандарта IEEE, известного также под названием стандарта POSIX. Именно эта оболочка и послужит основанием для изложения остального материала данной книги.

Все примеры применения, приведенные в данной книге, были проверены на компьютере Macintosh, работающем под управлением ОС Mac OS X 10.11, Ubuntu Linux 14.0, а также на рабочей станции Sparcstation Ultra-30 со старой версией SunOS 5.7. За исключением некоторых примеров применения в оболочке Bash из главы 14, все примеры выполнялись в оболочке Korn, хотя все они вполне работоспособны и в оболочке Bash.

Оболочка предоставляет интерпретируемый язык программирования, позволяющий быстро и просто писать, видоизменять и отлаживать программы. Мы прибегаем к оболочке как к первому избранному нами варианту языка программирования после того, как научились искусно программировать на языке оболочки. Надеемся, что и вы последуете нашему примеру.

Структура книги

В этой книге предполагается, что вы знакомы с основами системы Unix и режимом работы в командной строке, т.е. знаете, как входить в систему, создавать в ней файлы, редактировать и удалять их и как работать с каталогами. А на тот случай, если вы давно не пользовались системой Unix или Linux, мы изложим самые основы в главе 1 "Краткий обзор основ", где также поясняется порядок подстановки имен файлов, переадресация ввода-вывода и каналы.

В главе 2 "Назначение оболочки" поясняется, что собой представляет оболочка, как она действует и каким образом становится в конечном итоге основным средством взаимодействия с самой операционной системой. Из этой главы вы узнаете также, что происходит при входе в систему, каким образом программа оболочки запускается на выполнение, как в ней производится синтаксический анализ и автоматическое выполнение других программ. Главное внимание в данной главе уделяется тому обстоятельству, что оболочка является не более чем обычной программой.

В главе 3 "Рабочие инструменты" представлен учебный материал по инструментам, полезным для написания программ в оболочке. В частности, рассматриваются команды `cut`, `paste`, `sed`, `grep`, `sort`, `tr` и `uniq`. И хотя их выбор носит субъективный характер, тем не менее, он подготавливает почву для разработки программ, рассматриваемых в остальной части книги. Кроме того, подробно рассматриваются регулярные выражения, применяемые во многих командах Unix, в том числе `sed`, `grep` и `ed`.

Из глав 4–9 вы узнаете, как пользоваться оболочкой на практике для написания программ. В частности, вы научитесь писать свои команды; пользоваться переменными; писать программы, принимающие аргументы; делать выбор; пользоваться командами оболочки `for`, `while` и `until` для организации циклов; а также читать данные из терминала или файла по команде `read`. Глава 5

"Заключение в кавычки" полностью посвящена интерпретации кавычек — одной из самых захватывающих (и зачастую сбивающих с толку) особенностей оболочки. Этими главами завершается изложение всех основных программных конструкций в оболочке, и вы сможете писать в ней свои программы для решения конкретных задач.

В главе 10 "Рабочая среда" поясняется, насколько важно ясно понимать *среду*, в которой действует оболочка. В этой главе рассматриваются локальные и экспортируемые переменные, подоболочки, специальные переменные оболочки, в том числе `HOME`, `PATH` и `CDPATH`, а также порядок установки файла параметров пользователя с расширением `.profile`.

В главе 11 "Дополнительные сведения о параметрах" и главе 12 "Невыясненные вопросы" разрешаются некоторые не до конца выясненные вопросы. А в главе 13 "Возвращение к программе `rolo`" представлен окончательный вариант программы `rolo`, реализующей телефонный справочник и разрабатывавшейся на протяжении предыдущих глав книги.

В главе 14 "Интерактивные и нестандартные средства оболочки" рассматриваются те средства оболочки, которые формально не являются частью стандарта IEEE POSIX на оболочку. Тем не менее они вполне доступны в большинстве оболочек Unix и Linux и применяются, главным образом, интерактивно, а не в программах.

В приложении А "Краткое изложение оболочки" сведены все средства оболочки по стандарту IEEE POSIX. А в приложении Б "Дополнительные источники информации" приводится список первоисточников и ресурсов, включая те вебсайты, откуда могут быть загружены различные оболочки.

Данная книга составлена по принципу обучения на конкретных примерах. Мы считаем, что надлежащим образом подобранные примеры намного нагляднее демонстрируют порядок применения отдельных средств, чем их многословное описание. Старое изречение "Одна картинка лучше тысячи слов" как нельзя лучше применимо к программированию.

Настоятельно рекомендуем набирать каждый пример, рассматриваемый в данной книге, чтобы проверить его в вашей системе. Ведь только практика поможет вам научиться искусно программировать на языке оболочки. Не бойтесь экспериментировать. Пробуйте менять команды в примерах программ из данной книги, чтобы увидеть результат, а также вводить разные параметры и средства, чтобы сделать свои программы более полезными и надежными.

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

Краткий обзор основ

В этой главе дается краткий обзор ОС Unix, включая ее файловую систему, основные команды, подстановку имен файлов, переадресацию ввода-вывода и каналы.

Некоторые основные команды

Отображение даты и времени: команда `date`

Команда `date` предписывает системе вывести дату и время, как показано ниже.

```
$ date
Thu Dec 3 11:04:09 MST 2015
$
```

По команде `date` выводится день недели, месяц, день, время (в 24-часовом формате и часовой пояс, в котором действует система), а также год. Все, что в коде примеров, представленных в данной книге, выделяется **полужирным моноширинным шрифтом**, пользователь должен ввести в режиме командной строки. А обычным моноширинным шрифтом выделяется то, что выводит система Unix на экран терминала. И, наконец, *наклонным моноширинным шрифтом* выделяются комментарии в интерактивных последовательностях команд.

Выявление пользователей, зарегистрированных в системе: команда `who`

По команде `who` можно получить сведения обо всех пользователях, зарегистрированных в настоящее время в системе:

```
$ who
pat     tty29   Jul   19 14:40
ruth    tty37   Jul   19 10:54
steve   tty25   Jul   19 15:52
$
```

В данном примере зарегистрированными в системе оказываются следующие пользователи: `pat`, `ruth` и `steve`. Наряду с идентификатором пользователя в выводимом результате перечисляется номер `tty` данного пользователя, а также день и время его входа в система. Номер `tty` — это однозначный идентификационный номер, который система Unix присваивает каждому терминалу или сетевому устройству, с которого пользователь входит в систему.

По команде who пользователь может также получить сведения о самом себе следующим образом:

```
$ who am i
pat tty29 Jul 19 14:40
$
```

Команды who и who am i, по существу, являются одинаковыми. В последнем случае am и i являются *аргументами* команды who. (И хотя это не совсем удачный пример для демонстрации принципа действия аргументов, тем не менее, он раскрывает любопытные особенности команды who.)

Эхоотображение символов: команда echo

По команде echo на терминал выводится (или *отображается эхом*) все, что пользователь вводит в командной строке, как показано ниже, хотя из этого правила имеется ряд исключений, о которых речь пойдет далее.

```
$ echo this is a test
this is a test
$ echo why not print out a longer line with echo?
why not print out a longer line with echo?
$ echo
      Отображается пустая строка
$ echo one      two three four five
one two three four five
$
```

Как следует из последнего из приведенных выше примеров применения команды echo, она удаляет лишние пробелы между вводимыми словами. Дело в том, что для системы Unix слова важнее, чем пробелы, которые только разделяют слова. Как правило, лишние пробелы игнорируются системой Unix (подробнее об этом — в следующей главе).

Манипулирование файлами

В системе Unix распознаются только следующие типы файлов: *обыкновенные* файлы, *каталоги* и *специальные* файлы. Обыкновенным в системе считается любой файл, содержащий данные, текст, инструкции программы и практически все, что угодно. Каталоги (или папки) рассматриваются далее в этой главе. И, наконец, специальный файл, как подразумевает его название, имеет специальное назначение в системе Unix, и, как правило, связан с определенной формой ввода-вывода.

Имя файла может состоять из всех символов, непосредственно доступных, а иногда и не доступных, для ввода с клавиатуры, при условии, что их общее количество не превышает **255**. Если же в имени файла указано больше **255** символов, система Unix проигнорирует лишние символы.

В системе Unix предоставляется немало инструментов, упрощающих манипулирование файлами. Ниже вкратце описываются некоторые из основных команд манипулирования файлами.

Перечисление файлов: команда `ls`

Чтобы выяснить, какие именно файлы хранятся в текущем каталоге, достаточно ввести команду `ls` следующим образом:

```
$ ls
READ_ME
names
tmp
$
```

Как следует из результата выполнения команды `ls` в данном примере, в текущем каталоге находятся три файла, называемых `READ_ME`, `names` и `tmp`. (Результат, выводимый по команде `ls`, может отличаться в разных системах.) Например, в большинстве систем Unix результат выполнения команды `ls` выводится в нескольких столбцах, если он направляется на терминал, а в некоторых системах разные типы файлов могут выделяться отдельными цветами. Чтобы вывести результат в одном столбце, достаточно указать в команде `ls` параметр **`-1`**, обозначающий количество столбцов.

Отображение содержимого файла: команда `cat`

Содержимое файла можно отобразить по команде `cat`, название которой сокращенно обозначает "сцепление". В качестве аргумента команды `cat` указывается имя файла, содержимое которого требуется исследовать, как показано ниже.

```
$ cat names
Susan
Jeff
Henry
Allan
Ken
$
```

Подсчет количества слов в файле: команда `wc`

Выполнив команду `wc`, можно получить подсчитанное в итоге количество строк, слов и символов, содержащихся в файле. И в этом случае в качестве аргумента команды `wc` предполагается имя файла:

```
$ wc names
5 7 27 names
$
```

По команде wc выводится список из трех чисел, а вслед за ними — имя файла. Первое число обозначает количество строк (**5**), второе — количество слов (**7**), третье — количество символов в файле (**27**).

Дополнительные параметры команд

В большинстве команд Unix допускается дополнительно указывать *параметры*, используемые во время выполнения команды. Как правило, они указываются в следующем формате:

-буква

Это означает, что параметр команды указывается одной буквой сразу же после знака "минус". Например, чтобы подсчитать количество строк в файле, достаточно указать параметр **-l** (т.е. букву **l** после знака "минус") в команде wc, как показано ниже.

```
$ wc -l names
5 names
$
```

А для того чтобы подсчитать количество символов в файле, достаточно указать параметр **-c** следующим образом:

```
$ wc -c names
27 names
$
```

И, наконец, для подсчета количества слов в файле можно указать параметр **-w** таким образом:

```
$ wc -w names
7 names
$
```

В некоторых командах требуется, чтобы дополнительные параметры указывались прежде аргументов, обозначающих имена файлов. Например, команда `sort names -r` вполне допустима, тогда как команда `wc names -l` недопустима. Тем не менее первый вариант указания дополнительных параметров редок, и большинство команд Unix рассчитано на то, чтобы дополнительные параметры указывались прежде аргументов, как, например, в команде `wc -l names`.

Создание копии файла: команда cp

Для создания копии файла служит команда cp. В качестве первого аргумента этой команды указывается имя копируемого файла, называемого *исходным*, а в качестве второго аргумента — имя того файла, в котором должна быть размещена копия, называемого *целевым*. Например, создать копию файла names и назвать ее saved_names можно следующим образом:

```
$ cp names saved_names
$
```

В результате выполнения этой команды содержимое файла names копируется в новый файл saved_names. Как и при выполнении многих других команд Unix, вывод приглашения в командной строке вместо результата после ввода команды cp означает, что она была выполнена успешно.

Переименование файла: команда mv

Файл можно переименовать по команде mv (т.е. "переместить"). Аргументы команды mv указываются в том же самом формате, что и аргументы команды cp. В качестве первого аргумента команды cp указывается имя переименовываемого файла, а в качестве второго аргумента — новое имя файла. Например, чтобы сменить имя файла с saved_names на hold_it, достаточно выполнить следующую команду:

```
$ mv saved_names hold_it
$
```

Будьте внимательны, выполняя команду mv или cp! Ведь систему Unix не интересует, существует ли уже файл, указанный в качестве второго аргумента. Если он существует, его содержимое теряется. Так, если файл old_names существует, то в результате выполнения команды

```
cp names old_names
```

файл names будет скопирован в файл old_names, разрушив в ходе этого процесса прежнее содержимое файла old_names. Аналогично в результате выполнения команды

```
mv names old_names
```

файл будет переименован с names на old_names, даже если файл old_names существовал до выполнения данной команды.

Удаление файла: команда rm

Для удаления файла из системы служит команда rm. В качестве аргумента команды rm достаточно указать имя удаляемого файла следующем образом:

```
$ rm hold_it
$
```

Чтобы удалить больше одного файла по команде rm, достаточно указать все подобные файлы в командной строке. Например, в следующей команде можно удалить три файла wb, collect и mon:

```
$ rm wb collect mon
$
```

Манипулирование каталогами

Допустим, имеется один ряд файлов, содержащих различные заметки, предложения и письма, а также другой ряд файлов, содержащих компьютерные программы. Было бы вполне логично сгруппировать первый ряд файлов в каталоге `documents`, а второй — в каталоге `programs`. Подобная организация каталогов приведена на рис. 1.1.

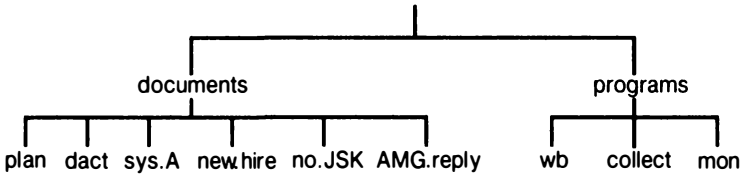


Рис. 1.1. Пример организации структуры каталогов

В каталоге `documents` содержатся файлы `plan`, `dact`, `sys.A`, `new.hire`, `no.JSK` и `AMG.reply`, а в каталоге `programs` — файлы `wb`, `collect` и `mon`. В какой-то момент может быть принято решение разделить файлы по категориям в каталоге. С этой целью можно создать подкаталоги, разместив в них соответствующие файлы. Например, в каталоге `documents` можно создать подкаталоги `memos`, `proposals` и `letters`, как показано на рис. 1.2.

В каталоге `documents` содержатся подкаталоги `memos`, `proposals` и `letters`, а в каждом из них — по два файла. Так, подкаталог `memos` содержит файлы `plan` и `dact`, подкаталог `proposals` — файлы `sys.A` и `new.hire`, а подкаталог `letters` — файлы `no.JSK` и `AMG.reply`.

И хотя у каждого файла в отдельном каталоге должно быть свое особенное имя, это совсем не обязательно для файлов в других каталогах. Так, в каталоге `programs` может находиться файл `dact`, несмотря на то, что файл с таким же именем находится в подкаталоге `memos`.

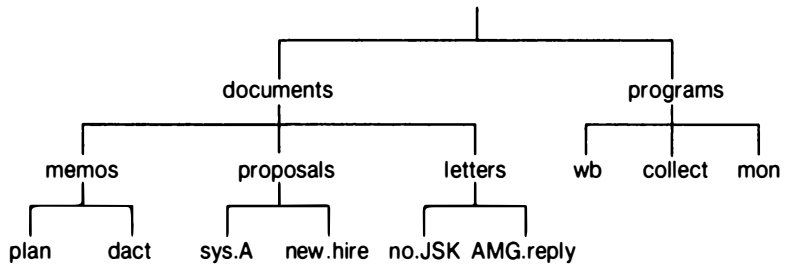


Рис. 1.2. Каталоги, содержащие подкаталоги

Начальный каталог и пути к файлам

Каждый пользователь системы Unix привязывается в ней к конкретному каталогу. При входе в систему пользователь автоматически направляется в свой каталог, называемый *начальным*.

Несмотря на то что местоположение начальных каталогов пользователей может отличаться в разных системах, допустим, что ваш начальный каталог называется `steve` и содержит подкаталог `users`. Следовательно, если у вас имеются каталоги `documents` и `programs`, то общая структура каталогов будет выглядеть так, как показано на рис. 1.3. На вершине дерева этой структуры находится специальный каталог, обозначаемый знаком косой черты (**/**) и называемый *корневым*.

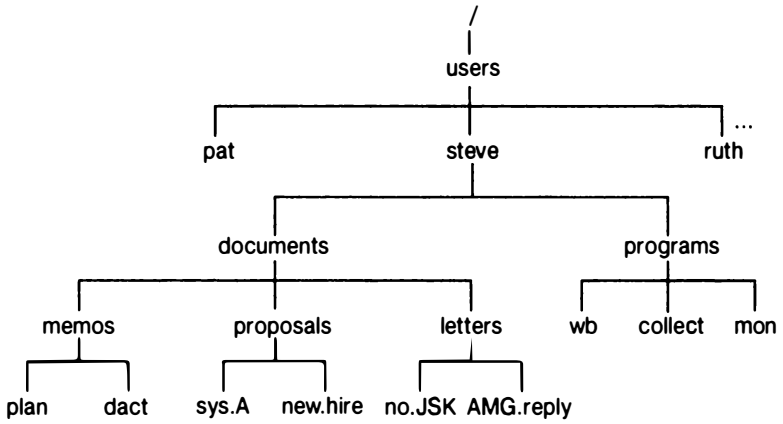


Рис. 1.3. Иерархическая древовидная структура каталогов

Когда вы находитесь в конкретном каталоге, называемом *текущим рабочим каталогом*, файлы из этого каталога доступны вам непосредственно и не требуют указывать путь к ним. Если же вам требуется доступ к файлу из другого каталога, вы можете выдать команду для смены текущего каталога на требующийся и получить доступ к конкретному файлу или указать *полное имя* конкретного файла для доступа к нему по указанному *пути*.

Полное имя, или путь к файлу, однозначно определяет конкретный файл в системе Unix. При указании пути к файлу соответствующие каталоги последовательно разделяются знаками косой черты (**/**). Путь к файлу, начинающийся со знака косой черты, называется *полным* или *абсолютным*, поскольку он обозначает весь путь от корневого каталога к нужному файлу. Например, путь `/users/steve` обозначает каталог `steve`, содержащийся в каталоге `users`. Аналогично путь `/users/steve/documents` обозначает каталог `documents`, содержащийся в каталоге `steve`, который, в свою очередь, находится в каталоге `users`. И, наконец, путь `/users/steve/documents/letters/AMG.reply` обозначает файл `AMG.reply`, находящийся в указанной структуре каталогов.

Чтобы упростить ввод путей к файлам, в системе Unix предоставляются некоторые удобства их обозначения. В частности, путь, не начинающийся со знака косой черты, называется *относительным*, т.е. он указывается относительно текущего рабочего каталога. Так, если вы только что вошли в систему и оказались в своем начальном каталоге `/users/steve`, для непосредственного обращения к каталогу `documents` вы можете просто ввести его имя **`documents`**. Аналогично для доступа к файлу `mon` в каталоге `programs` достаточно ввести относительный путь `programs/mon` к этому файлу.

Условное обозначение `..` всегда означает каталог, находящийся на один уровень выше текущего каталога и называемый *родительским*. Так, если вы находитесь в своем начальном каталоге `/users/steve`, для обращения к каталогу `users` вам достаточно ввести путь `..` (т.е. два знака точки подряд). А если вы выдали соответствующую команду для смены своего рабочего каталога на каталог `documents/letters`, то путь `..` будет обозначать обращение к каталогу `documents`, путь `../..` — обращение к каталогу `steve`, а путь `../proposals/new.hire` — обращение к конкретному файлу в каталоге `proposals`. Как правило, указать путь к конкретному файлу можно несколькими способами, что весьма характерно для системы Unix.

Еще одним условным обозначением является одиночный знак точки (`.`), который всегда обозначает текущий каталог. Такое обозначение приобретет большее значение далее в книге, когда речь пойдет об указании сценария оболочки в текущем каталоге, а не в переменной окружения `PATH`. Ниже мы поясним это более подробно.

Отображение рабочего каталога: `pwd`

Указывая имя вашего текущего рабочего каталога, команда `pwd` призвана помочь вам сориентироваться в структуре каталогов, приведенной на рис. 1.3. Если вы вошли в систему под именем пользователя `steve`, то окажетесь в своем начальном каталоге `/users/steve`. Чтобы убедиться в этом, введите команду `pwd` (ее название обозначает "напечатать рабочий каталог") следующим образом:

```
$ pwd
/users/steve
$
```

Как показывает результат выполнения этой команды, текущим для пользователя `steve` является рабочий каталог `/users/steve`.

Смена каталогов: команда `cd`

По команде `cd` можно сменить текущий рабочий каталог. В качестве аргумента эта команда принимает имя целевого каталога.

Допустим, вы вошли в систему под именем пользователя `steve` и оказались в своем начальном каталоге `/users/steve`. На рис. 1.4 ваше местонахождение в структуре каталогов обозначено стрелкой.

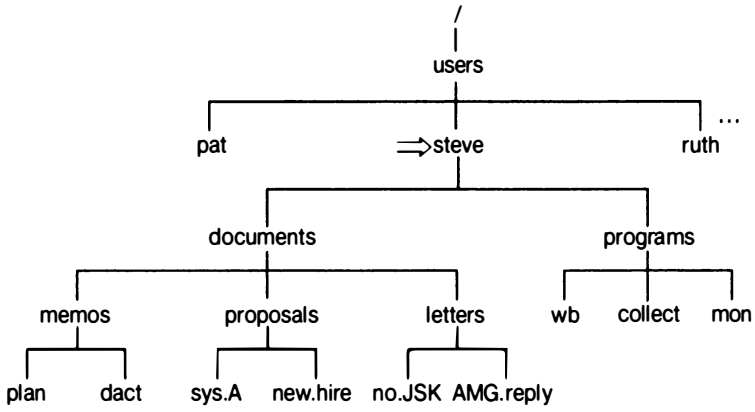


Рис. 1.4. Текущим является рабочий каталог `steve`

Вам известно, что ниже вашего начального каталога `steve` расположены еще два каталога, `documents` и `programs`. В этом нетрудно убедиться, введя команду `ls`, как показано ниже. По команде `ls` каталоги `documents` и `programs` перечисляются таким же образом, как и обыкновенные файлы в предыдущих примерах.

```
$ ls
documents
programs
$
```

Чтобы сменить текущий рабочий каталог, достаточно ввести команду `cd`, указав после нее имя нового каталога:

```
$ cd documents
$
```

Выполнив эту команду, вы окажетесь в каталоге `documents`, как показано на рис. 1.5.

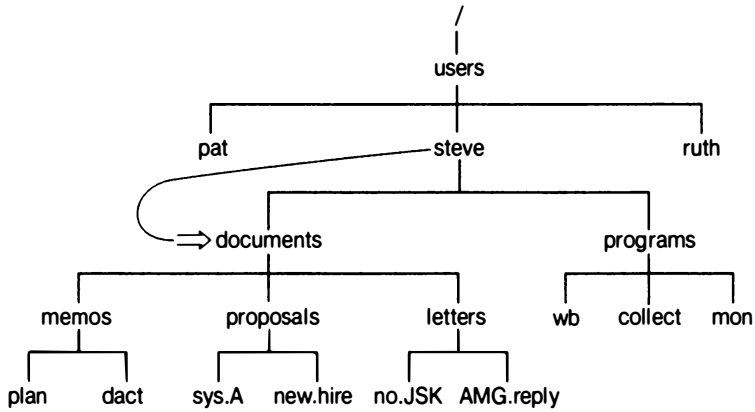


Рис. 1.5. Переход к каталогу `documents` по команде `cd`

Результат смены рабочего каталога можно проверить по команде pwd следующим образом:

```
$ pwd
/users/steve/documents
$
```

Чтобы перейти на один уровень вверх, проще всего ввести команду **cd ..**, поскольку условное обозначение **..** всегда обозначает каталог, находящийся на один уровень выше в структуре каталогов (рис. 1.6):

```
$ cd ..
$ pwd
/users/steve
$
```

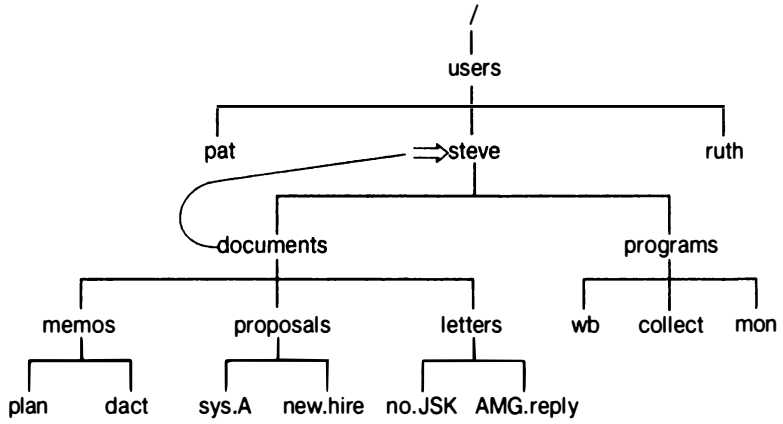


Рис. 1.6. Переход на один уровень выше в структуре каталогов по команде cd ..

Если же требуется перейти к каталогу letters, это можно также сделать по команде cd, указав относительный путь documents/letters, как показано ниже и на рис. 1.7.

```
$ cd documents/letters
$ pwd
/users/steve/documents/letters
$
```

Чтобы вернуться в свой начальный каталог, введите приведенную ниже команду cd, по которой осуществляется переход на два каталога вверх.

```
$ cd ../..
$ pwd
/users/steve
$
```

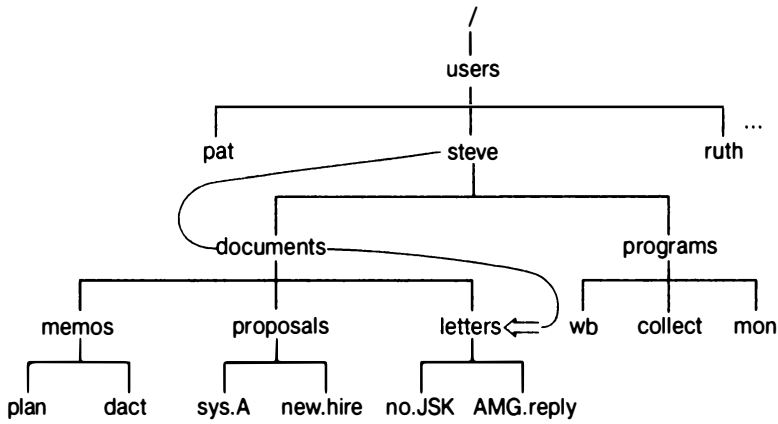


Рис. 1.7. Переход к каталогу `letters` по команде `cd documents/letters`

С другой стороны, вы можете вернуться в свой начальный каталог, указав полный путь к нему вместо относительного следующим образом:

```
$ cd /users/steve
$ pwd
/users/steve
$
```

И, наконец, третий и самый простой способ вернуться в свой начальный каталог состоит в том, чтобы ввести команду `cd` *без* аргумента, как показано ниже. По этой команде *всегда* происходит возврат в начальный каталог независимо от вашего местоположения в файловой системе.

```
$ cd
$ pwd
/users/steve
$
```

Дополнительные сведения о команде `ls`

В результате выполнения введенной команды `ls` обычно перечисляется содержимое текущего каталога. С помощью команды `ls` можно также перечислить содержимое других каталогов, предоставив ей соответствующий аргумент. Но прежде вернитесь в начальный каталог следующим образом:

```
$ cd
$ pwd
/users/steve
$
```

А теперь просмотрите содержимое текущего рабочего каталога, как показано ниже.

```
$ ls
documents
programs
$
```

Если в качестве аргумента команды `ls` указать имя одного из перечисленных выше каталогов, то можно получить список файлов, содержащихся в данном каталоге. Например, чтобы выяснить, что же находится в каталоге `documents`, достаточно ввести команду `ls documents`, как показано ниже.

```
$ ls documents
letters
memos
proposals
$
```

А для просмотра содержимого подкаталога `memos` достаточно выполнить аналогичную процедуру следующим образом:

```
$ ls documents/memos
dact
plan
$
```

Если же в качестве аргумента команды `ls` указать путь к файлу, то в конечном итоге на экран терминала будет выведено имя этого файла:

```
$ ls documents/memos/plan
documents/memos/plan
$
```

Если вас смущает такой результат, не позволяющий отличить отдельный файл от каталога, укажите в команде `ls` параметр **`-l`**, чтобы получить более подробное описание файлов в каталоге. Так, если вы находитесь в начальном каталоге пользователя `steve`, то, введя команду `ls` с параметром **`-l`**, вы получите следующий результат:

```
$ ls -l
total 2
drwxr-xr-x   5   steve   DP3725   80 Jun 25 13:27   documents
drwxr-xr-x   2   steve   DP3725   96 Jun 25 13:31   programs
$
```

В первой строке приведенного выше результата отображается подсчитанное общее количество блоков памяти, содержащих по 1024 байта в каждом и используемых перечисленными файлами, в данном случае — каталогами. А в каждой

последующей строке, отображаемой по команде `ls -l`, представлены подробные сведения о файле в каталоге. В частности, первый символ обозначает тип файла: `d` — каталог, `-` — файл, `b`, `c`, `l` или `p` — специальный файл.

Следующие девять символов в выводимой строке обозначают права доступа к конкретному файлу в каталоге. Это *режимы доступа*, распространяемые на владельца файла (первые три символа), других пользователей из той же самой *группы*, что и владелец файла (следующие три символа), и, наконец, на всех остальных пользователей системы (последние три символа). Как правило, они обозначают, разрешено ли пользователю указанного класса читать содержимое файла, записывать в него новую информацию или выполнять его содержимое, если это файл программы или сценария оболочки.

Далее в рассматриваемой здесь строке, выводимой по команде `ls -l`, отображается подсчитанное количество *ссылок* на файл (подробнее об этом — далее, в разделе "Связывание файлов: команда `ln`"), имя владельца файла, имя группового владельца файла, размер файла (т.е. количество содержащихся в нем символов), а также дата последней модификации файла. И последним в рассматриваемой здесь строке отображается имя самого файла, в данном случае — каталога.

Примечание

Во многих современных системах Unix больше не используются группы пользователей. Поэтому при отображении прав доступа к файлу или каталогу по команде **`ls`** сведения о его групповом владельце зачастую опускаются.

Таким образом, из результатов, выводимых по команде `ls -l`, можно извлечь немало полезных сведений о каталоге, заполненном файлами, как показано ниже.

```
$ ls -l programs
total 4
-rwxr-xr-x   1 steve   DP3725   358   Jun 25 13:31   collect
-rwxr-xr-x   1 steve   DP3725  1219   Jun 25 13:31   mon
-rwxr-xr-x   1 steve   DP3725    89   Jun 25 13:30   wb
$
```

Знак дефиса в первом столбце приведенного выше результата обозначает, что в данном каталоге присутствуют три обыкновенных файла, `collect`, `mon` и `wb`, но не каталоги. Сумеете ли вы теперь выяснить, насколько они велики?

Создание каталога: команда `mkdir`

Для создания каталогов служит команда `mkdir`. А в качестве ее аргумента указывается имя создаваемого каталога. Допустим, вы по-прежнему работаете со структурой каталогов, приведенной на рис. 1.7, и вам требуется создать новый каталог `misc` на *том же самом* уровне, что и каталоги `documents` и `programs`. Если вы находитесь теперь в своем начальном каталоге, то можете добиться желаемого результата, введя команду `mkdir misc`:

```
$ mkdir misc
$
```

Если вы затем выполните команду `ls`, то в списке текущих каталогов увидите вновь созданный каталог:

```
$ ls
documents
misc
programs
$
```

Теперь структура каталогов выглядит так, как показано на рис. 1.8.

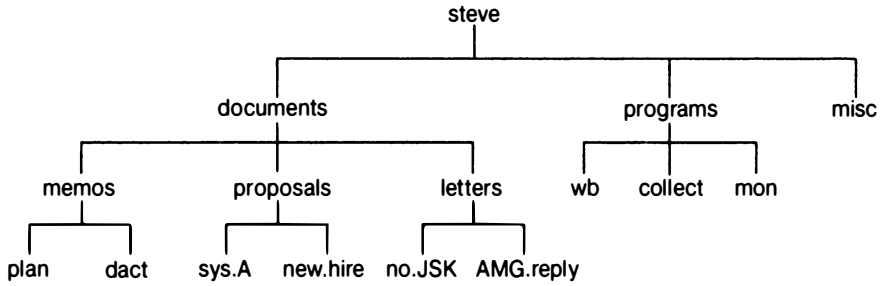


Рис. 1.8. Структура каталогов со вновь созданным каталогом `misc`

Копирование файла из одного каталога в другой

Команда `cp` служит для копирования файла из одного каталога в другой. Например, файл `wb` можно скопировать из каталога `programs` в файл `wbx`, размещаемый в каталоге `misc`, следующим образом:

```
$ cp programs/wb misc/wbx
$
```

Если оба файла находятся в разных каталогах, они могут вполне носить одно и то же имя:

```
$ cp programs/wb misc/wb
$
```

При выполнении приведенной выше команды система Unix распознает каталог в качестве второго аргумента данной команды и копирует исходный файл в этот каталог. Новому файлу присваивается такое же имя, как и у исходного файла.

В указанный каталог можно скопировать не один, а несколько файлов, перечислив их перед именем целевого каталога. Так, если вы находитесь в каталоге `programs`, то по команде

```
$ cp wb collect mon ../misc
$
```

сможете скопировать три файла, wb, collect, и mon, в каталог misc под теми же самыми именами.

Чтобы скопировать файл из другого каталога в текущий каталог и присвоить новому файлу то же самое имя, воспользуйтесь удобным сокращенным обозначением (.) текущего каталога. Так, по указанной ниже команде cp файл collect копируется из каталога ../programs в текущий каталог (/users/steve/misc).

```
$ pwd
/users/steve/misc
$ cp ../programs/collect .
$
```

Перемещение файлов из одного каталога в другой

Напомним, что файл можно переименовать по команде mv. На самом деле в системе Unix нет специальной команды для переименования файлов. Но если оба аргумента команды mv обозначают разные каталоги, то файл фактически перемещается из первого каталога во второй.

Чтобы продемонстрировать порядок перемещения файлов из одного каталога в другой, необходимо сначала перейти из начального каталога в каталог documents по следующей команде:

```
$ cd documents
$
```

А теперь допустим, что файл plan требуется переместить из каталога memos в каталог proposals, поскольку он в действительности содержит предложение, а не заметку. Это можно сделать следующим образом:

```
$ mv memos/plan proposals/plan
$
```

Как и в команде cp, в команде mv достаточно указать только имя целевого каталога, если имена исходного и целевого файлов одинаковы. Следовательно, файл plan проще всего переместить по следующей команде:

```
$ mv memos/plan proposals
$
```

Аналогично команде cp, по команде mv можно также переместить в другой каталог целую группу файлов, перечислив их перед именем целевого каталога, как показано ниже. В итоге три файла, wb, collect и mon, будут перемещены в каталог misc.

```
$ pwd
/users/steve/programs
$ mv wb collect mon ../misc
$
```

С помощью команды mv можно даже изменить имя каталога. Например, по следующей команде каталог programs переименовывается в bin:

```
$ mv programs bin
$
```

Связывание файлов: команда ln

При рассмотрении команд манипулирования файлами речь до сих пор шла о том, что отдельный набор данных носит одно и только одно имя файла, где бы он ни находился в файловой системе. Но оказывается, что в системе Unix допускается не только это, но и возможность присваивать несколько имен файлов одному и тому же набору данных.

Основой для создания дубликатов имени отдельного файла служит команда ln. Ниже приведена общая форма этой команды, где файл *откуда* связывается с файлом *куда*.

```
ln откуда куда
```

Как следует из структуры каталогов пользователя steve, приведенной на рис. 1.8, в каталоге programs хранится файл программы wb. Допустим, пользователю steve требуется также назвать этот файл writeback. Очевидно, что с этой целью он может создать копию файла wb под именем writeback следующим образом:

```
$ cp wb writeback
$
```

Недостаток такого подхода заключается в том, что файл программы занимает теперь в два раза больше места на диске. Более того, внеся изменения в файл wb, пользователь steve может забыть продублировать эти изменения в файле writeback, в результате чего у него окажутся две несинхронизированные копии одной и той же, как ему кажется, программы. Нехорошо получается, Стив!

Этих осложнений можно избежать, связав файл wb с новым именем следующим образом:

```
$ ln wb writeback
$
```

Теперь вместо двух копий одного файла существует только одна его копия под двумя именами: wb и writeback. В итоге оба файла оказываются логически связанными в системе Unix.

Складывается такое впечатление, будто теперь имеются два *разных* файла программы. В самом деле, если выполнить программу ls, то в конечном итоге будут показаны два отдельных файла — wb и writeback:

```
$ ls
collect
```

```
mon
wb
writeback
$
```

Но самое любопытное оказывается, если выполнить команду `ls -l`, как показано ниже.

```
$ ls -l
total 5
-rwxr-xr-x   1 steve   DP3725    358 Jun 25 13:31 collect
-rwxr-xr-x   1 steve   DP3725   1219 Jun 25 13:31 mon
-rwxr-xr-x   2 steve   DP3725     89 Jun 25 13:30 wb
-rwxr-xr-x   2 steve   DP3725     89 Jun 25 13:30 writeback
$
```

Обратите особое внимание на второй столбец приведенного выше результата. Для файлов `collect` и `mon` в нем указано число **1**, а для файлов `wb` и `writeback` — число **2**. Эти числа обозначают количество ссылок на файл. Так, число **1**, как правило, указывается для несвязанных обыкновенных файлов. Но поскольку файлы `wb` и `writeback` связаны вместе, это число равно для них **2**. А точнее, оно означает, что у данного файла имеются два имени.

Любой их двух связанных вместе файлов можно удалить, когда угодно, как показано ниже. При этом другой файл не удаляется. Как видите, количество ссылок на файл `wb` теперь стало равным **1**.

```
$ rm writeback
$ ls -l
total 4
-rwxr-xr-x   1 steve   DP3725    358 Jun 25 13:31 collect
-rwxr-xr-x   1 steve   DP3725   1219 Jun 25 13:31 mon
-rwxr-xr-x   1 steve   DP3725     89 Jun 25 13:30 wb
$
```

Чаще всего команда `ln` применяется для того, чтобы один файл мог одновременно появиться в нескольких каталогах. Допустим, пользователю `pat` требуется доступ к файлу программы `wb` пользователя `steve`. Вместо того чтобы сделать себе копию этого файла (и тем самым навлечь на себя описанные выше осложнения, возникающие в связи с синхронизацией копий одинаковых файлов) или включить каталог `programs`, где находится файл программы `wb`, в переменную окружения `PATH`, что связано с риском нарушения безопасности, как поясняется в главе 10, пользователь `pat` может просто связать данный файл с каталогом своих программ следующим образом:

```
$ pwd
/users/pat/bin              каталог программ пользователя pat
$ ls -l
total 4
```

```
-rwxr-xr-x   1 pat    DP3822    1358 Jan 15 11:01 lcat
-rwxr-xr-x   1 pat    DP3822     504 Apr 21 18:30 xtr
$ ln /users/steve/wb .          Связать файл wb с каталогом bin
                                пользователя pat
$ ls -l
total 5
-rwxr-xr-x   1 pat      DP3822   1358   Jan 15 11:01  lcat
-rwxr-xr-x   2 steve    DP3725     89   Jun 25 13:30  wb
-rwxr-xr-x   1 pat      DP3822    504   Apr 21 18:30  xtr
$
```

Обратите внимание на то, что пользователь `steve` по-прежнему перечислен как владелец файла `wb` — даже при просмотре содержимого каталога пользователя `pat`. И в этом есть свой смысл, ведь на самом деле имеется лишь одна копия файла, которой владеет пользователь `steve`.

В отношении связывания файлов необходимо сделать следующую оговорку: связываемые вместе обыкновенные файлы должны *непременно* находиться в одной и той же файловой системе. В противном случае при попытке связать файлы по команде `ln` будет выдана ошибка. (Чтобы выявить разные файловые системы в своей системе, выполните команду `df`. В первом поле каждой строки результата выполнения этой команды указывается наименование файловой системы.)

Чтобы связать файлы в разных файловых системах (или в разных сетевых системах), достаточно указать параметр **`-s`** в команде `ln`. В итоге будет образована *символическая* ссылка. Своим поведением символические ссылки очень похожи на обычные ссылки, за исключением того, что символическая ссылка указывает на первоначальный файл. Если же первоначальный файл удаляется, символическая ссылка больше не действует.

Выясним, каким образом действуют символические ссылки, на следующем примере:

```
$ rm wb
$ ls -l
total 4
-rwxr-xr-x   1 pat    DP3822    1358   Jan 15 11:01  lcat
-rwxr-xr-x   1 pat    DP3822     504   Apr 21 18:30  xtr
$ ln -s /users/steve/wb ./symwb      символическая ссылка на файл wb
$ ls -l
total 5
-rwxr-xr-x   1 pat    DP3822    1358 Jan 15 11:01  lcat
lrwxr-xr-x   1 pat    DP3822      15 Jul 20 15:22  symwb ->
                                                   /users/steve/wb
-rwxr-xr-x   1 pat    DP3822     504 Apr 21 18:30  xtr
$
```

Как видите, пользователь перечислен в данном примере как владелец файла `symwb`, а тип этого файла обозначен самым первым символом `l` в результате,

выводимом по команде ls. Этот символ обозначает символическую ссылку, размер которой равен **15** (соответствующий файл фактически содержит символьную строку /users/steve/wb). Но если попытаться получить доступ к содержимому этого файла, то будет предоставлено содержимое связанного с ним файла /users/steve/wb:

```
$ wc symwb
        5        9       89  symwb
$
```

Вместе с параметром **-l** в команде **ls** можно также указать дополнительный параметр **-L**, чтобы получить подробный список сведений о файле, на который указывает символическая ссылка:

```
$ ls -Ll
total 5
-rwxr-xr-x   1 pat      DP3822   1358   Jan 15 11:01  lcat
-rwxr-xr-x   2 steve    DP3725     89   Jun 25 13:30  wb
-rwxr-xr-x   1 pat      DP3822    504   Apr 21 18:30 xtr
$
```

Если удалить файл, на который указывает символическая ссылка, она станет недействительной, как следует из приведенного ниже примера. Ведь символические ссылки сохраняются как имена файлов, но не удаляются.

```
$ rm /users/steve/wb          предположим, что пользователь pat
                              может удалить этот файл
$ ls -l
total 5
-rwxr-xr-x   1 pat   DP3822   1358   Jan 15 11:01  lcat
lrwxr-xr-x   1 pat   DP3822     15   Jul 20 15:22  wb ->
                                                  /users/steve/wb
-rwxr-xr-x   1 pat   DP3822    504   Apr 21 18:30  xtr
$ wc wb
Cannot open wb: No such file or directory
(Не удается открыть файл wb: такого файла или
 каталога не существует)
$
```

Файл такого типа называется *висячей символической ссылкой* и должен быть удален, если только нет особых причин сохранить его (например, для того, чтобы заменить удаленный файл).

И прежде чем завершить рассмотрение команды ln, необходимо заметить, что она придерживается такой же формы, как и команды cp и mv. Это означает, что ссылки на группу файлов в конкретном целевом каталоге можно создавать, используя следующую форму команды ln:

```
ln файлы каталог
```

Удаление каталога: команда rmdir

Удалить каталог можно по команде rmdir. Но чтобы не удалить случайно десятки или даже сотни файлов, команда rmdir не позволит сделать это до тех пор, пока указанный каталог не будет полностью освобожден от файлов и подкаталогов. Например, для удаления каталога /users/pat можно было бы ввести следующую команду:

```
$ rmdir /users/pat
rmdir: pat: Directory not empty
(rmdir: pat: каталог не пустой)
$
```

Но это было бы ошибкой! Вместо этого можно попытаться удалить созданный ранее каталог misc следующим образом:

```
$ rmdir /users/steve/misc
$
```

И в этом случае команда rmdir будет выполнена лишь при условии, что в каталоге misc не окажется ни файлов, ни подкаталогов. В противном случае произойдет то же самое, что и прежде, как показано ниже. Если каталог misc требуется все-таки удалить, необходимо сначала удалить из него все файлы, а затем выдать команду rmdir еще раз.

```
$ rmdir /users/steve/misc
rmdir: /users/steve/misc: Directory not empty
$
```

В качестве альтернативы удалению каталога и его содержимого можно воспользоваться командой rm с параметром **-r** в следующей простой форме:

```
rm -r каталог
```

где *каталог* — имя удаляемого каталога. Команда rm удаляет указанный каталог и *все* файлы в нем, включая и подкаталоги. Поэтому пользуйтесь этой мощной командой очень аккуратно.

Чтобы избавиться от лишних хлопот, связанных с необходимостью реагировать каждый раз на приглашение удалить содержимое каталога, достаточно указать дополнительный параметр **-f** в команде rm. Но подобным образом можно по неосторожности легко опустошить всю систему, и поэтому многие системные администраторы избегают пользоваться командой rm -rf!

Подстановка имен файлов

В последующих разделах описываются различные механизмы подстановки имен файлов.

Знак звездочки

К числу самых эффективных средств системы Unix, употребляемых в оболочке, относится *подстановка имен файлов*. Допустим, в текущем каталоге находятся следующие файлы:

```
$ ls
chapt1
chapt2
chapt3
chapt4
$
```

Допустим также, что требуется отобразить все содержимое этих файлов. Это нетрудно сделать по команде `cat`, позволяющей отображать содержимое стольких файлов, сколько указано в командной строке:

```
$ cat chapt1 chapt2 chapt3 chapt4

$
```

Но вводить целый ряд файлов в командной строке неудобно. Вместо этого можно выгодно воспользоваться подстановкой их имен, просто введя следующее:

```
$ cat *

$
```

Оболочка автоматически *подставит* имена всех файлов в текущем каталоге, совпадающих с шаблоном `*`. Такая же подстановка происходит и в том случае, если воспользоваться знаком звездочки (`*`) в какой-нибудь другой команде, например `echo`. Подстановка выполняется оболочкой *везде*, где появляется знак `*` в командной строке, как показано в следующем примере:

```
$ echo * : *
chapt1 chapt2 chapt3 chapt4 : chapt1 chapt2 chapt3 chapt4
$
```

Знак `*` относится к развитому языку подстановки имен файлов. Он может употребляться в сочетании с другими знаками для ограничения, накладываемого на совпадение имен файлов.

Допустим, в текущем каталоге имеются не только файлы `chapt1-chapt4`, но и файлы `a`, `b` и `c`:

```
$ ls
a
b
c
chapt1
```

```
chapt2
chapt3
chapt4
$
```

Чтобы отобразить содержимое только тех файлов, имена которых начинаются на chap, достаточно ввести следующую команду:

```
$ cat chap*



$
```

Шаблон chap* совпадает с любым именем файла, начинающимся на chap. Имена всех подобных файлов подставляются в командной строке еще до вызова указанной команды.

Знак * можно указывать не только в конце, но и в начале или в средине имени файла, как показано в следующем примере:

```
$ echo *t1
chapt1
$ echo *t*
chapt1 chapt2 chapt3 chapt4
$ echo *x
*x
$
```

В первой из приведенных выше команд echo шаблон *t1 обозначает имена всех файлов, оканчивающихся на t1. Во второй команде echo первый знак * обозначает совпадение со всеми символами до буквы t, а второй знак * — совпадение со всеми символами после буквы t. Таким образом, выводятся имена всех файлов, содержащие букву t. В текущем каталоге отсутствуют файлы, имена которых оканчиваются на букву x, поэтому в последней команде echo совпадение не происходит, и она просто отображает свой шаблон *x.

Совпадение с одиночными символами

Знак звездочки (*) обозначает совпадение с нулевым и более количеством символов. В частности, с шаблоном x* совпадают имена файлов x, x1, x2, xabc и т.д. А знак вопроса (?) обозначает совпадение только с одним символом. Так, по команде cat ? отобразятся все файлы, имена которых состоят только из одной буквы, а по команде cat x? — все файлы, имена которых состоят из двух символов и начинаются на букву x. Такой режим совпадения с одиночными символами наглядно демонстрируется в следующем примере применения команды echo:

```
$ ls
a
aa
aax
alice
b
bb
c
cc
report1
report2
report3
$ echo ?
a b c
$ echo a?
aa
$ echo ??
aa bb cc
$ echo ??*
aa aax alice bb cc report1 report2 report3
$
```

В данном примере с шаблоном **??** совпадают два символа, а с шаблоном ***** — нулевое и большее количество символов до самого конца имени файла. В итоге с шаблоном **??*** совпадают все файлы, имена которых состоят из двух или больше символов.

Указать совпадение с одиночным символом можно и по-другому, предоставив список сопоставляемых символов в квадратных скобках (**[]**). Например, с шаблоном `[abc]` совпадает буква `a`, `b` или `c`. Этот шаблон действует подобно знаку **?**, но позволяет выбрать конкретные символы для сопоставления.

Через дефис можно также указать логический ряд или диапазон символов, что очень удобно. Например, с шаблоном `[0-9]` совпадают символы от **0** до **9**. На указание *диапазона* символов накладывается единственное ограничение: первый символ должен следовать раньше в алфавитном порядке, чем последний. Следовательно, диапазон символов `[z-f]` оказывается недействительным, тогда как диапазон символов `[f-z]` — действительным.

Сочетая диапазоны и одиночные символы в шаблонах, можно выполнять сложные подстановки. Например, с шаблоном `[a-np-z]*` совпадают все файлы, имена которых начинаются с буквы `a` и до `n` или `p` и вплоть до `z`, а проще говоря, все файлы, имена которых *не* начинаются на букву `o`.

Если первым после открывающей квадратной скобки (**[**) следует знак восклицания (**!**), то смысл сопоставления с шаблоном меняется на обратный. Это означает, что с шаблоном совпадает любой символ, *кроме* указанных в квадратных скобках. Таким образом, с шаблоном

```
[!a-z]
```

совпадает имя файла с любым символом, кроме строчных букв, а с шаблоном

```
*[!o]
```

имя любого файла, не оканчивающееся на строчную букву o.

Некоторые характерные примеры подстановки имен файлов приведены в табл. 1.1.

Таблица 1.1. Примеры подстановки имен файлов

Команда	Описание
`echo a*`	Вывести *имена* всех файлов, начинающиеся на букву **a**
`cat *.c`	Вывести содержимое всех файлов, имена которых оканчиваются на букву **c**
`rm *.*`	Удалить все файлы, в именах которых содержится знак точки (.)
`ls x*`	Перечислить все файлы, имена которых начинаются на букву **x**
`rm *`	Удалить *все* файлы из текущего каталога (пользуйтесь этой командой очень аккуратно)
`echo a*b`	Вывести имена всех файлов, имена которых начинаются на букву **a** и оканчиваются на букву **b**
`cp ../programs/*`	Скопировать все файлы из каталога **../programs** в текущий каталог
`ls [a-z]*[!0-9]`	Перечислить файлы, имена которых начинаются со строчной буквы и не оканчиваются на цифру

Особенности указания имен файлов

В последующих разделах поясняются особенности указания имен файлов.

Пробелы в именах файлов

Рассмотрение команд и имен файлов было бы неполным без обсуждения пробелов в именах файлов — пагуба для традиционных пользователей системы Unix и вполне обыденное явление для современных пользователей систем Linux, Windows и Mac OS.

Дело в том, что в оболочке пробелы служат в качестве разделителей слов. Иными словами, фраза `echo hi mom` надлежащим образом интерпретируется как вызов команды `echo` с двумя аргументами, `hi` и `mom`.

А теперь допустим, что имеется файл `my test document`. Как обратиться к нему из командной строки? Как просмотреть или отобразить его содержимое по команде `cat`?

```
$ cat my test document
cat: my: No such file or directory
```

```
cat: test: No such file or directory
cat: document: No such file or directory
```

Почему не удается выполнить приведенную выше команду `cat`? А потому что ей требуется имя файла, и вместо одного файла она обнаруживает целых три: `my`, `test` и `document`.

Имеются два стандартных выхода из этого затруднительного положения: экранировать каждый пробел знаком обратной косой черты или заключить все имя файла в кавычки, чтобы оболочка интерпретировала его как одно слово с пробелами, а не ряд слов, разделяемых пробелами. Оба эти варианта демонстрируются в следующем примере:

```
$ cat "my test document"
This is a test document and is full
of scintillating information to edify
and amaze.
$ cat my\ test\ document
This is a test document and is full
of scintillating information to edify
and amaze.
(Это тестовый документ, наполненный блестящей
 информацией для назидания и изумления.)
```

Приведенные выше варианты выхода из положения следует знать, работая с файловыми системами, где может присутствовать немало каталогов и файлов с пробелами в их именах.

Другие необычные символы

Если пробелы могут оказаться самыми неудобными и неприятными специальными символами в именах файлов, то иногда в них встречаются и другие необычные символы, способные стать настоящей помехой для работы в режиме командной строки.

Как, например, обращаться с именем файла, содержащим знак вопроса? В следующем разделе поясняется, что знак вопроса (**?**) имеет особое назначение в оболочке. Большинство современных оболочек обладают достаточно развитой логикой, чтобы избежать двойного назначения специальных символов, но и в этом случае рекомендуется заключать имена файлов в кавычки или экранировать специальные символы знаками обратной косой черты для указания на то, что они являются частью имени файла, как демонстрируется в следующем примере:

```
$ ls -l who\ me\?
-rw-r--r--   1 taylor   staff    0 Dec    4 10:18   who me?
```

Но самая интересная ситуация возникает, когда в имени файла присутствует сам знак обратной косой черты или кавычки. Это может произойти неумышленно, особенно в файлах, создаваемых в графически ориентированных программах,

работающих в системе Linux или Mac OS. В таком случае для экранирования имен файлов, содержащих двойные кавычки, следует употребить одиночные кавычки, и наоборот:

```
$ ls -l "don't quote me" 'She said "yes"'
-rw-r--r--   1 taylor   staff    0 Dec   4 10:18 don't quote me
-rw-r--r--   1 taylor   staff    0 Dec   4 10:19 She said "yes"
```

Мы еще вернемся к этому вопросу далее в книге, но теперь вы уже знаете, как справиться с именами файлов или каталогов, содержащими пробелы и другие необычные символы.

Стандартный ввод-вывод и его переадресация

В последующих разделах описывается организация стандартного ввода-вывода данных в системе Unix.

Стандартный ввод и вывод

Большинство команд в системе Unix принимают данные, вводимые с клавиатуры, и посылают результат на экран. В терминологии Unix экран и клавиатура называются *терминалом*, отсылающим к ранней эпохе развития вычислительной техники. Ныне это, вероятнее всего, терминальная *прикладная программа*, выполняемая в графической среде, будь то оконный диспетчер в Linux, ПК под Windows или система Mac.

Как правило, команда читает данные из *стандартного ввода*, которым по умолчанию является клавиатура на вашем компьютере. Это особый способ обозначить набор вводимых данных. Аналогичным образом команда, как правило, записывает результат своего выполнения в *стандартный вывод*, которым по умолчанию также служит терминал или терминальная прикладная программа. Такой принцип ввода-вывода в командах системы Unix наглядно показан на рис. 1.9.

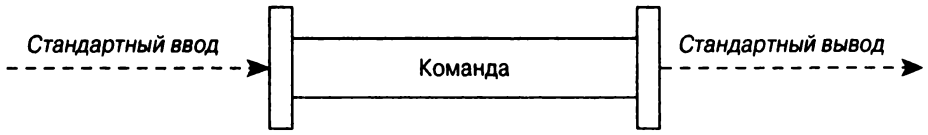


Рис. 1.9. Типичная организация ввода-вывода в командах системы Unix

В качестве примера рассмотрим выполнение команды who, приводящее к отображению всех пользователей, зарегистрированных в настоящий момент в системе. Более формально команда who направляет список зарегистрированных пользователей в стандартный вывод, как показано на рис. 1.10.

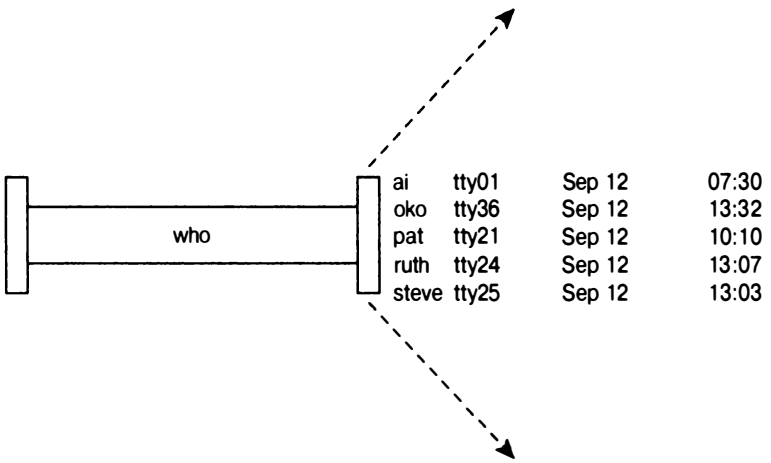


Рис. 1.10. Порядок выполнения команды `who`

Оказывается, что практически каждая одиночная команда Unix может принять результат, выводимый командой, или файл в качестве своего ввода и даже послать результат своего выполнения другой команде или программе. Этот принцип ввода-вывода очень важен для понимания сильных сторон командной строки и причин, по которым полезно знать все рассматриваемые здесь команды, даже если у вас есть возможность пользоваться графическим интерфейсом.

Но прежде рассмотрим следующее: если команда `sort` вызывается *без* имени файла в качестве аргумента, она принимает свои входные данные из стандартного ввода. Как и стандартным выводом, по умолчанию стандартным вводом служит терминал (или клавиатура).

Когда данные вводятся подобным образом, последовательность окончания файла должна быть указана после ввода последней строки. В системе Unix с этой целью принято нажимать комбинацию клавиш <Ctrl+d>, т.е. нажать одновременно клавиши <Ctrl> и <d>.

В качестве примера воспользуемся командой `sort` для сортировки следующих имен: `Tony`, `Barbara`, `Harry`, `Dirk`. Вместо того чтобы вводить эти имена сначала в файл, введем их непосредственно с терминала следующим образом:

```
$ sort
Tony
Barbara
Harry
Dirk
<Ctrl+d>
Barbara
Dirk
Harry
Tony
$
```

В данном примере команде `sort` не было указано имя файла, и поэтому исходные данные взяты из стандартного ввода, которым служит терминал. После ввода четвертого имени была нажата комбинация клавиш <Ctrl+d>, обозначающая конец ввода данных. Вслед за этим команда `sort` отсортировала четыре введенных имени и отобразила результат сортировки на устройстве стандартного вывода, которым также служит терминал (рис. 1.11).

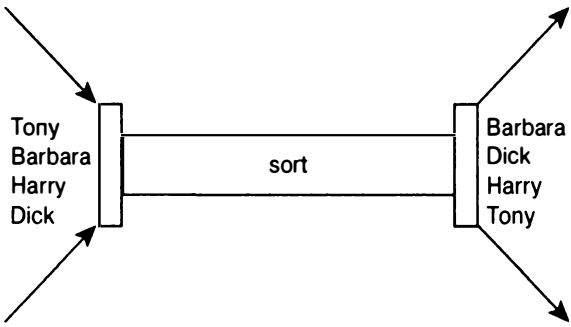


Рис. 1.11. Порядок выполнения команды `sort`

Еще одним примером команды, принимающей исходные данные из стандартного ввода, если в командной строке не указано имя файла, служит команда `wc`. В следующем примере демонстрируется применение этой команды для подсчета количества строк текста, введенных с терминала:

```
$ wc -l
This is text that
is typed on the
standard input device.
(Этот текст набран на устройстве
 стандартного ввода)
<Ctrl+d>
      3
$
```

Обратите внимание на то, что комбинация клавиш *<Ctrl+d>*, употребляемая для прекращения ввода данных, не считается отдельной строкой в команде `wc`, поскольку она интерпретируется оболочкой и не передается далее команде. Кроме того, в результате выполнения команды `wc` представлен только подсчет количества строк (**3**), поскольку в данной команде указан параметр **`-l`**.

Переадресация вывода

Как правило, результат выполнения команды, предназначенный для стандартного вывода, нетрудно перенаправить в файл. Такая возможность называется *переадресацией вывода* и не менее важна для уяснения истинного потенциала системы Unix.

Если обозначение > *файл* присоединяется к *любой* команде, она, как правило, направляет результат своего выполнения не в стандартный вывод, а в указанный *файл*, как показано в приведенном ниже примере.

```
$ who > users
$
```

В данном примере команда `who` направляет результат своего выполнения в файл `users`. Обратите внимание на отсутствие отображаемого результата. Дело в том, что выводимый результат был *переадресован* из устройства стандартного вывода (т.е. терминала) в указанный файл. В этом можно убедиться следующим образом:

```
$ cat users
oko     tty01   Sep  12  07:30
ai      tty15   Sep  12  13:32
ruth    tty21   Sep  12  10:10
pat     tty24   Sep  12  13:07
steve   tty25   Sep  12  13:03
$
```

Если команда переадресует результат своего выполнения в файл, который уже содержит некоторые данные, эти данные будут перезаписаны и утрачены. Рассмотрим следующий пример:

```
$ echo line 1 > users
$ cat users
line 1
$
```

Продолжим данный пример, помня, что в файле `users` уже содержится результат выполнения предыдущей команды `who`:

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Внимательно присмотревшись к приведенной выше команде `echo`, вы обнаружите, что в ней употребляется совсем другой тип переадресации вывода, обозначаемый знаками **>>**. При такой переадресации стандартный вывод из команды *присоединяется* к содержимому указанного файла. Прежнее содержимое файла не теряется, а новое содержимое (т.е. выводимый командой результат) просто присоединяется в его конце.

Указав в команде `cat` знаки **>>**, обозначающие переадресацию с присоединением, содержимое одного файла можно присоединить в конце другого файла следующим образом:

```
$ cat file1
This is in file1.
(Это содержимое файла file1)
$ cat file2
This is in file2.
(А это содержимое файла file2)
$ cat file1 >> file2            Присоединить файл file1 к файлу file2
$ cat file2
This is in file2.
This is in file1.
$
```

Напомним, что если указать в команде `cat` несколько файлов, то их содержимое отобразится одно за другим. И это означает, что того же самого результат можно добиться другим способом:

```
$ cat file1
This is in file1.
$ cat file2
This is in file2.
$ cat file1 file2
This is in file1.
This is in file2.
$ cat file1 file2 > file3       Переадресовать оба файла в третий
$ cat file3
This is in file1.
This is in file2.
$
```

В данном примере команда `cat` вполне оправдывает свое название. Если в ней указано несколько файлов, то она *сцепляет* их вместе.

Переадресация ввода

Ввод для команды *из* файла может быть переадресован таким же образом, как и вывод из команды. И если для обозначения переадресации вывода служит знак **>**, то для обозначения переадресации ввода — знак **<**. Безусловно, переадресация ввода из файла допускается только в тех командах, которые обычно принимают свои исходные данные из стандартного ввода.

Чтобы переадресовать ввод, достаточно указать знак **<** и имя файла, откуда производится ввод. Например, для того чтобы подсчитать количество строк в файле `users`, достаточно, как вам должно быть уже известно, выполнить команду `wc` с параметром **`-l`**:

```
$ wc -l users
      2 users
$
```

Оказывается, что для подсчета количества строк в файле достаточно переадресовать стандартный ввод из него в команду wc таким образом:

```
$ wc -l < users
      2
$
```

Обратите внимание на отличие в результатах, выводимых в обеих формах команды wc. В первом случае имя файла users перечисляется вместе с подсчетом количества строк, а во втором случае оно отсутствует.

Это указывает на едва заметное отличие в выполнении обеих форм данной команды. В первом случае команде wc известно, что она читает свои исходные данные из файла users. А во втором случае она только обнаруживает исходные данные, поступающие через стандартный ввод. Оболочка переадресует ввод таким образом, чтобы он поступал из файла users, а не с терминала (подробнее об этом — в следующей главе). Что же касается команды wc, то ей неизвестно, откуда поступают ее входные данные: с терминала или из файла, и поэтому она не может сообщить имя файла!

Каналы

Если помните, созданный ранее файл users содержит список всех пользователей, зарегистрированных в настоящий момент в системе. А поскольку известно, что в этом файле на каждого зарегистрированного пользователя приходится отдельная строка, то нетрудно определить *количество* сеансов регистрации, подсчитав количество строк в файле users:

```
$ who > users
$ wc -l < users
      5
$
```

Как следует из приведенного выше результата, в настоящий момент в системе зарегистрировано пять пользователей или произведено пять сеансов регистрации. Единственное отличие заключается в том, что пользователи, особенно администраторы, нередко регистрируются неоднократно. Итак, имеется последовательность команд, с помощью которой можно в любой момент выяснить, сколько пользователей зарегистрировано в системе.

Другой способ определить количество зарегистрированных пользователей состоит в том, чтобы обойтись без промежуточного файла. Как упоминалось ранее, в системе Unix допускается соединять команды вместе. Такое соединение называется *каналом* и позволяет направить выход из одной команды непосредственно на вход другой команды. Канал обозначается знаком |, указываемым между

двумя командами. Например, чтобы установить канал между командами who и wc -l, достаточно ввести следующее:

```
$ who | wc -l
      5
$
```

Канал, устанавливаемый между этими двумя командами, наглядно показан на рис. 1.12.

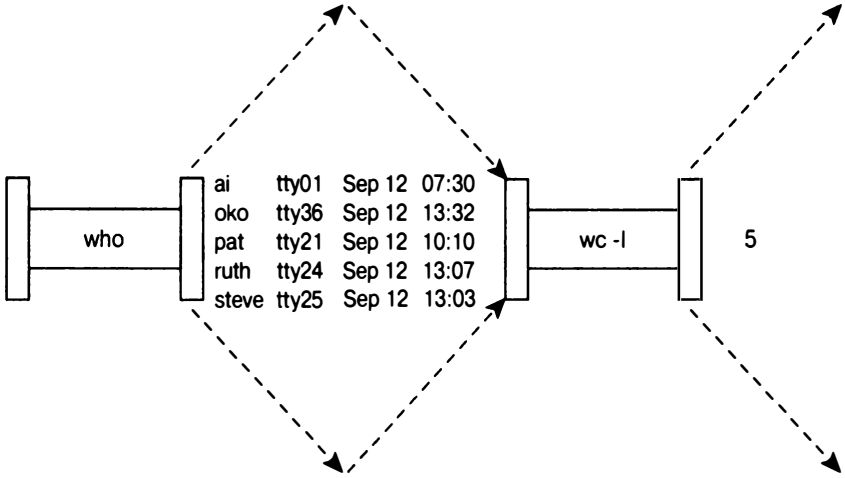


Рис. 1.12. Процесс организации канала между командами who и wc -l

Когда между двумя командами устанавливается канал, стандартный вывод из первой команды непосредственно соединяется со стандартным вводом второй команды. Как вам должно быть уже известно, команда who направляет составленный ею список зарегистрированных пользователей в стандартный вывод. Вы знаете также, что команда wc берет свои исходные данные из стандартного ввода, если в ней не указано имя файла в качестве аргумента. Следовательно, список зарегистрированных пользователей, выводимый из команды who, автоматически становится входными данными для команды wc. Обратите внимание на то, что команда who не выводит результат своего выполнения на терминал, поскольку он направляется конвейером по каналу непосредственно команде wc (рис. 1.13). Канал можно установить между *любыми* двумя программами, при условии, что первая программа направляет результат своего выполнения в стандартный вывод, а вторая — читает свои входные данные из стандартного ввода.

В качестве еще одного примера допустим, что требуется подсчитать количество файлов в каталоге. Зная, что команда ls выводит файлы из каталога построчно, можно воспользоваться каналом, как показано ниже. В текущем каталоге, как следует из выведенного результата, находится десять файлов.

```
$ ls | wc -l
      10
$
```

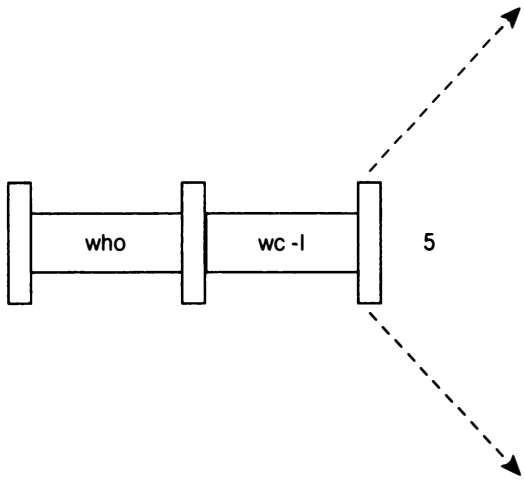


Рис. 1.13. Конвейерный процесс

Имеется также возможность организовать сложный *конвейер*, состоящий из более чем двух программ, чтобы направлять результат выполнения одной программы следующей далее программе. Став более искушенным пользователем режима командной строки, вы обнаружите немало случаев, когда конвейеры оказываются необычайно эффективными.

Фильтры

Термин *фильтр* нередко употребляется в терминологии Unix для обозначения любой программы, которая способна принимать входные данные из стандартного ввода, выполнять какую-нибудь операцию над этими данными и направлять полученный результат в стандартный вывод. Короче говоря, фильтром может служить любая программа, с помощью которой можно видоизменить результат выполнения других программ в конвейере. Так, в конвейере из предыдущего примера команда `wc` считается фильтром, а команда `ls` таковой не считается, поскольку она не читает свои исходные данные из стандартного ввода. В качестве других примеров фильтров можно привести команды `cat` и `sort`, тогда как команды `who`, `date`, `cd`, `pwd`, `echo`, `rm`, `mv` и `cp` *не* являются фильтрами.

Стандартный вывод ошибок

Помимо стандартного ввода и вывода данных, имеется еще одно виртуальное устройство *стандартного вывода ошибок*. Именно на это устройство большинство команд в системе Unix выводят свои сообщения об ошибках. И как

стандартный ввод и вывод, стандартный вывод ошибок связан по умолчанию с терминалом или терминальной прикладной программой. Зачастую отличия стандартного вывода ошибок от стандартного вывод данных практически незаметны, как показано в следующем примере:

```
$ ls n*                 Список всех файлов, имена которых
                        начинаются с буквы n
n* not found
$
```

В данном примере сообщение "not found" (не найдено) фактически направляется командой ls в стандартный вывод ошибок. Чтобы убедиться в том, что данное сообщение не направляется в стандартный вывод данных, результат выполнения команды ls можно переадресовать в файл, как показано в приведенном ниже примере. Сообщение, как видите, по-прежнему выводится на терминал и не направляется в файл foo, несмотря на то, что стандартный вывод переадресовывается.

```
$ ls n* > foo
n* not found
$
```

В данном примере наглядно демонстрируется следующее разумное основание для существования стандартного вывода ошибок: вывод сообщений об ошибках на терминал, даже если стандартный вывод переадресовывается в файл или направляется по каналу другой команде.

Стандартный вывод ошибок можно также переадресовать в файл (например, том случае, если потенциальные ошибки в программе протоколируются в ходе ее длительного выполнения). Для этой цели служит следующая, немного более сложная форма:

```
команда 2> файл
```

Обратите внимание на то, что *пробел* между цифрой **2** и знаком **>** *не* допускается. Любые сообщения об ошибках, которые, как правило, направляются в стандартный вывод ошибок, перенаправляются в указанный *файл* подобно переадресации стандартного вывода данных, как показано в следующем примере:

```
$ ls n* 2> errors
$ cat errors
n* not found
$
```

Дополнительные сведения о командах

В последующих разделах представлены дополнительные возможности для применения команд в оболочке системы Unix.

Ввод нескольких команд в одной строке

В одной строке можно ввести несколько команд, при условии, что они разделяются точкой с запятой. Например, для того чтобы выяснить текущие время и рабочий каталог, достаточно ввести команды `date` и `pwd` в одной строке, как показано ниже. В одной строке можно ввести длинную цепочку команд, разделив их точкой с запятой.

```
$ date; pwd
Sat Jul 20 14:43:25 EDT 2002
/users/pat/bin
$
```

Передача команд на выполнение в фоновый режим

Как правило, вы вводите команду и ожидаете вывода результатов ее выполнения на экране. Во всех рассмотренных ранее примерах ждать приходилось недолго — доли секунды.

Но иногда приходится выполнять команды, для завершения которых требуется несколько минут, а то и больше. В подобных случаях приходится ждать завершения команды, прежде чем продолжить дальше, если только не выполнять ее в *фоновом режиме*.

На первый взгляд, системы Unix и Linux полностью сосредоточены на выполнении одной конкретной задачи, но на самом деле все они являются многозадачными системами, способными одновременно выполнять много задач в любой заданный момент времени. Например, в системе Ubuntu одновременно могут действовать оконный диспетчер, системные часы, монитор состояний и окно терминала. Из командной строки можно также выполнить несколько команд одновременно. Это делается по принципу передачи команд на выполнение в фоновый режим, что дает возможность работать над другими задачами до завершения команды.

Для условного обозначения передачи команды или последовательности команд на выполнение в фоновый режим служит знак амперсанда (**&**). Он указывает на то, что команда больше не связана с терминалом, что дает возможность перейти к другим задачам. Стандартный *вывод* из такой команды по-прежнему направляется на терминал, тогда как стандартный *ввод* не связывается с терминалом. Если команда все же попытается прочитать данные из стандартного ввода, она остановится в ожидании ее передачи на дальнейшее выполнение в приоритетный режим (подробнее об этом речь пойдет в главе 14). Ниже приведен пример передачи команды на выполнение в фоновый режим.

```
$ sort bigdata > out &          Передать команду sort на
                                выполнение в фоновый режим
[1] 1258                        Идентификационный номер процесса
$ date                          Терминал немедленно освобождается
                                для работы над другой задачей
Sat Jul 20 14:45:09 EDT 2002
$
```

Когда команда передается на выполнение в фоновый режим, система Unix автоматически отображает два числа. Первое число обозначает *номер задания* данной команды, а второе число — *идентификационный номер процесса* (PID). В приведенном выше примере число **1** обозначает номер задания, а число **1258** — идентификационный номер процесса. Номер задания служит в качестве сокращенного варианта обращения к конкретному фоновому заданию в некоторых командах оболочки (подробнее об этом — в главе 14). А идентификационный номер процесса однозначно определяет команду, переданную на выполнение в фоновом режиме. С его помощью можно получить сведения о состоянии команды, и это делается по команде ps (т.е. состояние процесса).

Команда ps

Команда ps предоставляет сведения о процессах, выполняемых в системе. Если эта команда вводится без дополнительных параметров, то она выводит только состояние процессов. Введя команду ps с терминала, можно получить в итоге несколько строк, описывающих выполняемые процессы, как показано в следующем примере:

```
$ ps
  PID    TTY          TIME  CMD
  13463  pts/16   00:00:09  bash
  19880  pts/16   00:00:00  ps
$
```

Как правило, хотя это зависит от конкретной системы, команда ps выводит результат в четырех столбцах: PID (Идентификационный номер процесса), TTY (Номер терминала, с которого процесс запущен на выполнение), TIME (Системное время в минутах и секундах, израсходованное процессом), а также CMD (Название процесса). В приведенном выше примере процесс bash является оболочкой, запущенной на выполнение при входе в систему. Этот процесс израсходовал 9 секунд системного времени. До тех пор, пока команда не завершится, она отображается в результатах, выводимых командой ps, как выполняемый процесс, и в данном примере процесс **19880** является самой командой ps.

Если команда ps вводится с параметром **-f**, она выводит дополнительные сведения о процессах, включая идентификационный номер *родительского* процесса (PPID), время запуска процесса на выполнение (STIME), а также аргументы команды:

```
$ ps -f
UID     PID     PPID   C  STIME  TTY     TIME      CMD
steve   13463   13355  0  12:12  pts/16  00:00:09  bash
steve   19884   13463  0  13:39  pts/16  00:00:00  ps -f
$
```

Сводка команд

В табл. 1.2 сведены все команды, рассмотренные в этой главе. В данной таблице *файл* обозначает указываемый файл, *файл(ы)* — один или несколько указываемых файлов, *каталог* — указываемый каталог, *каталог(и)* — один или несколько указываемых каталогов, *аргументы* — заданные аргументы команды.

Таблица 1.2. Сводка команд

Команда	Описание
`cat` ***файл(ы)***	Отобразить содержимое указанных ***файл(ов)***, а если они не указаны — стандартный ввод
`cd` ***каталог***	Заменить рабочий каталог на указанный ***каталог***
`cp` ***файл***$_1$ ***файл***$_2$	Скопировать ***файл***$_1$ в ***файл***$_2$
`cp` ***файл(ы) каталог***	Скопировать ***файл(ы)*** в ***каталог***
`date`	Отобразить дату и время
`echo` ***аргументы***	Отобразить заданные ***аргументы***
`ln` ***файл***$_1$ ***файл***$_2$	Связать ***файл***$_1$ и ***файл***$_2$
`ln` ***файл(ы) каталог***	Связать ***файл(ы)*** и ***каталог***
`ls` ***файл(ы)***	Перечислить ***файл(ы)***
`ls` ***каталог(и)***	Перечислить файлы, которые содержат указанные ***каталог(и)*** или текущий каталог, если ***каталог(и)*** не указаны
`mkdir` ***каталог(и)***	Создать указанные ***каталог(и)***
`mv` ***файл***$_1$ ***файл***$_2$	Переместить ***файл***$_1$ в ***файл***$_2$ (или переименовать файл, если оба указанных файла находятся в одном и том же каталоге)
`mv` ***файл(ы) каталог***	Переместить ***файл(ы)*** в указанный ***каталог***
`ps`	Вывести списком сведения об активных процессах
`pwd`	Отобразить путь к текущему рабочему каталогу
`rm` ***файл(ы)***	Удалить ***файл(ы)***
`rmdir` ***каталог(и)***	Удалить *пустые* ***каталог(и)***
`sort` ***файл(ы)***	Отсортировать строки, которые содержат указанные ***файл(ы)*** или стандартный ввод, если ***файл(ы)*** не указаны
`wc` ***файл(ы)***	Подсчитать количество строк, слов и символов, которые содержат указанные ***файл(ы)*** или стандартный ввод, если ***файл(ы)*** не указаны
`who`	Отобразить пользователей, зарегистрированных в системе

2

Назначение оболочки

В этой главе рассматривается основное назначение командной оболочки Unix, принцип ее действия и причины, по которым она является столь важным инструментом в арсенале средств всякого опытного пользователя системы Unix.

Ядро и утилиты

Система Unix логически разделяется на две области: *ядро* и *утилиты* (рис. 2.1). А если угодно, то ее можно представить как ядро и все остальное, обычно доступное через оболочку.

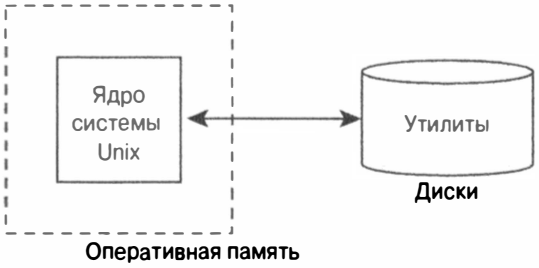


Рис. 2.1. Система Unix

Ядро образует сердцевину системы Unix и находится в оперативной памяти компьютера, начиная с того момента, когда компьютер включен, а система загружена, и кончая тем моментом, когда система закрывается, а компьютер выключается.

Различные инструменты и утилиты, обеспечивающие полноценное взаимодействие пользователя с системой Unix, находятся на дисках компьютера, загружаются в оперативную память и выполняются только по запросу. Буквально каждая известная вам команда Unix является утилитой, т.е. служебной программой, находящейся на диске и загружаемой в оперативную память только по запросу. Например, после ввода команды `date` в командной строке система Unix загружает служебную программу (или утилиту) под названием `date` в оперативную память с диска на компьютере и начинает читать ее код для выполнения указанного действия или нескольких действий.

Оболочка также является служебной программой, загружаемой в оперативную память в процессе входа в систему. В действительности стоит знать точную последовательность событий, происходящих в тот момент, когда первая оболочка запускается на терминале или в окне.

Исходная оболочка

Прежде терминалы были физическими устройствами, напрямую подключавшимися к системе Unix. В настоящее время терминальные программы дают пользователям возможность взаимодействовать с системой Unix по сети в управляемом окне, оставаясь в своей среде Linux, Mac OS или Windows. Как правило, пользователь запускает на выполнение программу вроде Terminal или xterm, а затем подключается к удаленным системам, используя такие служебные программы, как `ssh`, `telnet` или `rlogin`.

Для каждого физического терминала в системе Unix активизируется служебная программа `getty`. Этот процесс наглядно показан на рис. 2.2.

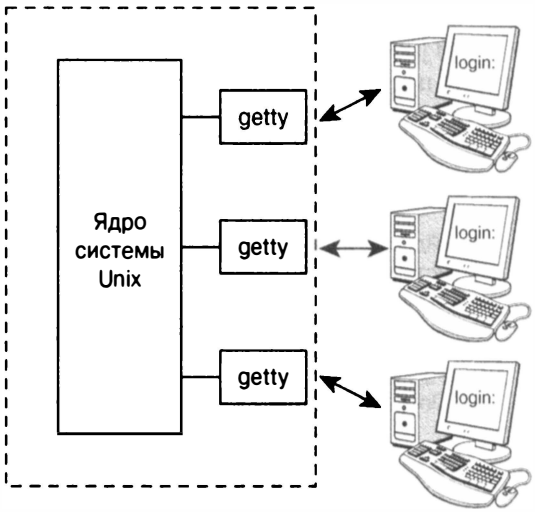


Рис. 2.2. Процесс активизации терминала по программе **`getty`**

Система Unix, а точнее — программа `init`, автоматически запускает программу `getty` в каждом порту терминала всякий раз, когда система разрешает пользователям войти в нее. По существу, программа `getty` служит драйвером устройства, позволяя служебной программе `login` вывести сообщение `login:` (вход в систему) на назначенный терминал и перейти в режим ожидания ввода регистрационных данных.

Если вы подключаетесь к системе через такую служебную программу, как `ssh`, для вас назначается псевдотерминал или pseudo-tty (ptty) в терминологии Unix. Именно поэтому вы обнаруживаете такие элементы, как `ptty3` или `pty1`, когда вводите команду `who`.

Так или иначе, имеется программа, читающая данные вашей учетной записи и пароля, а также программа, проверяющая эти данные на достоверность и вызывающая программы регистрации, с помощью которых вы можете войти в систему, если проверка введенных вами учетных данных пройдет успешно. Как только вы

введете свои учетные данные и нажмете клавишу <Enter>, программа `login` завершит процесс входа в систему (рис. 2.3).

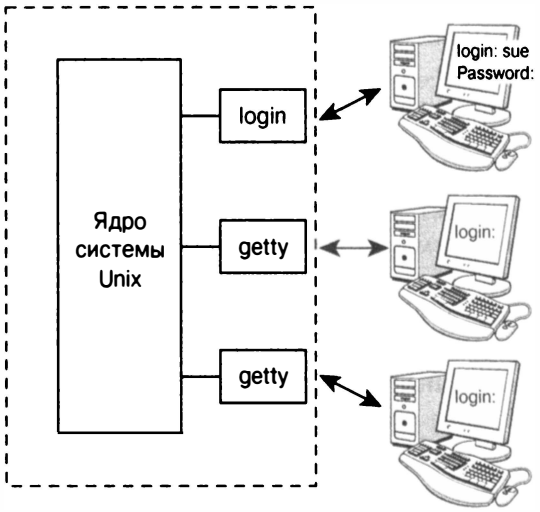


Рис. 2.3. Запуск служебной программы **`login`** на терминале пользователя **`sue`**

Когда программа `login` начинает свое выполнение, она отображает символьную строку `Password:` (Пароль:) и переходит в режим ожидания ввода вашего пароля. Как только вы введете свой пароль и нажмете клавишу <Enter> (при этом вы не увидите введенный вами пароль из соображений безопасности), программа `login` перейдет к проверке достоверности введенного вами регистрационного имени пользователя и пароля по соответствующей записи в файле `etc/passwd`. В этом файле содержатся записи, организуемые по учетной записи каждого пользователя, где, среди прочего, указано регистрационное имя, начальный каталог и программа, запускаемая при входе пользователя в систему. И последний фрагмент информации (исходная оболочка) сохраняется в каждой строке вслед за *последним* двоеточием. Если же вслед за последним двоеточием ничего не следует, то по умолчанию предполагается *стандартная* оболочка `/bin/sh`.

Если вы входите в систему через терминальную программу, в процесс обмена данными с подтверждением может быть также вовлечена служебная программа вроде `ssh` в вашей системе или `sshd` на сервере. И если вы откроете окно в своей системе Unix, то, скорее всего, сразу же войдете в систему, не вводя пароль снова, что очень удобно.

Но вернемся к файлу пароля. Ниже приведены типичные строки из файла `/etc/passwd` для регистрации трех пользователей системы: `sue`, `pat` и `bob`.

```
sue:*:15:47::/users/sue:
pat:*:99:7::/users/pat:/bin/ksh
bob:*:13:100::/users/data:/users/data/bin/data_entry
```

Как только программа login проверит на достоверность введенный вами пароль по зашифрованному паролю в указанной учетной записи, хранящейся в файле /etc/shadow, она перейдет к проверке имени выполняемой программы регистрации. В большинстве случаев это будет исходная оболочка /bin/sh, /bin/ksh или /bin/bash. А в других случаях это может быть специально разработанная программа или же программа /bin/nologin для тех учетных записей, которые не включают в себя доступ в диалоговом режиме, что характерно для управления владением файлами. Основной замысел всех этих программ состоит в том, чтобы настроить учетную запись для входа в систему на автоматическое выполнение любой программы, кто бы и когда бы ни входил в систему. В качестве такой программы чаще всего выбирается оболочка, потому что она является общей утилитой, хотя это и не единственный вариант выбора.

Но вернемся к пользователю sue. Как только он пройдет проверку на достоверность своих учетных данных, программа login, по существу, уничтожит себя, передав сначала управление стандартной оболочке для подключения терминала пользователя sue, а затем исчезнув из оперативной памяти (рис. 2.4).

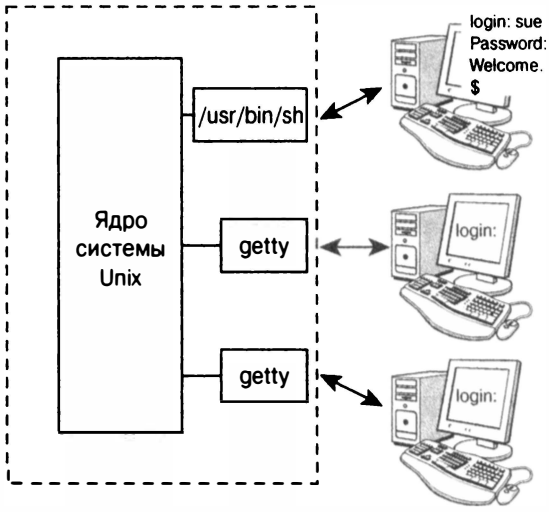


Рис. 2.4. Запуск стандартной оболочки **/usr/bin/sh** на выполнение из программы **login**

В соответствии с остальными приведенными выше записями из файла /etc/passwd пользователь pat получает доступ к программе ksh, хранящейся в каталоге /bin (это оболочка Корна), а пользователь bob — доступ к специальной программе data_entry (рис. 2.5).

Как упоминалось ранее, программа init выполняется аналогично программе getty для сетевых соединений. Например, программы sshd, telnetd и logind отвечают на запросы соединения через программы ssh, telnet и rlogin соответственно. Вместо конкретного физического терминала эти программы

подключают оболочки пользователей к *псевдотерминалам*. Можете убедиться в этом, войдя в систему через сеть, экран X Windows или программу подключения сетевого терминала и выполнив команду who, как показано ниже.

```
$ who
phw    pts/0    Jul 20 17:37          Этот пользователь вошел в систему
                                      с помощью программы rlogin
$
```

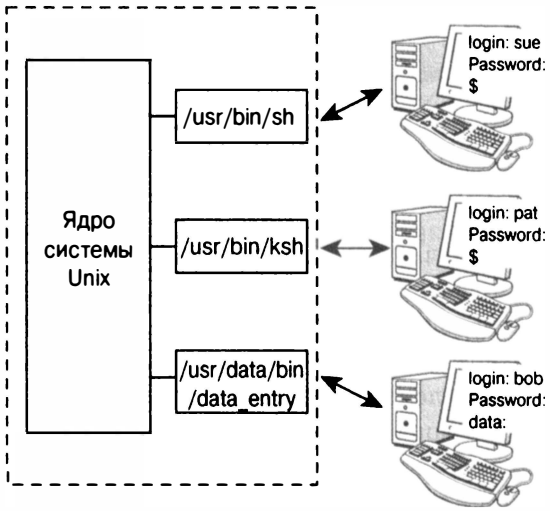


Рис. 2.5. Процесс входа трех пользователей в систему

Ввод команд в оболочке

Когда оболочка запускается на выполнение, она отображает на терминале приглашение командной строки — как правило, знак денежной единицы (**$**) — и затем переходит в режим ожидания, когда пользователь введет команду (см. шаги 1 и 2 цикла выполнения команд на рис. 2.6). Всякий раз, когда вы вводите команду и нажимаете клавишу <Enter> (шаг 3), оболочка анализирует то, что вы ввели, и переходит к выполнению запроса (шаг 4).

Если вы запросите оболочку вызвать конкретную программу, она приступит к поиску программы во всех каталогах на диске, указанных в переменной окружения PATH, до тех пор, пока не найдет указанную программу. Как только программа будет найдена, оболочка создаст свою копию, называемую *подоболочкой*, и запросит у ядра заменить подоболочку указанной программой. Затем исходная оболочка перейдет в режим ожидания до тех пор, пока не завершится запрашиваемая программа (шаг 5). Ядро скопирует указанную программу в оперативную память и приступит к ее выполнению. Эта копия программы называется *процессом*. Подобным образом проводится различие между программой, хранящейся в файле на диске, и процессом, находящимся в оперативной памяти и выполняемым построчно.

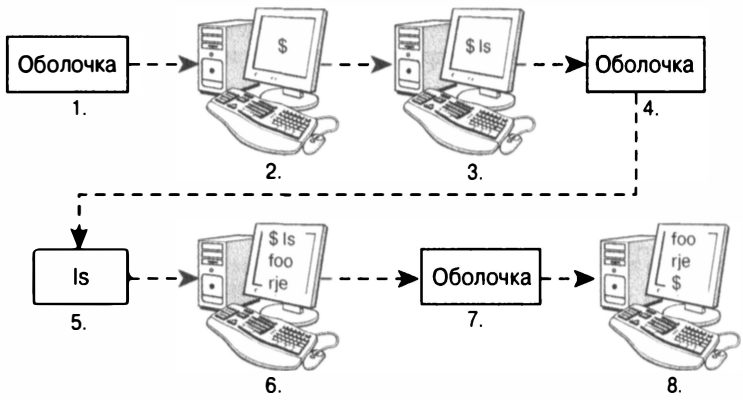


Рис. 2.6. Цикл выполнения команд

Если программа направляет результат своего выполнения в стандартный вывод, этот результат появится на вашем терминале, если, конечно, он не переадресовывается или направляется по каналу другой команде. Аналогично, если программа читает входные данные из стандартного ввода, она перейдет в режим ожидания ввода пользователем этих данных, если только их ввод не переадресовывается из файла или не направляется по каналу из другой команды (шаг 6). Когда же команда завершит свое выполнение, она исчезнет из оперативной памяти, и управление снова возвратится к исходной оболочке, которая выдаст приглашение на ввод следующей команды (шаги 7 и 8).

Следует, однако, иметь в виду, что описанный выше цикл выполнения команд продолжается до тех пор, пока вы не выйдете из системы. Как только вы выйдете из системы, выполнение оболочки прервется и система запустит новую программу `getty` (`rlogind` и т.д.), перейдя в режим ожидания входа в систему какого-нибудь другого пользователя. Этот цикл регистрации в системе наглядно показан на рис. 2.7.

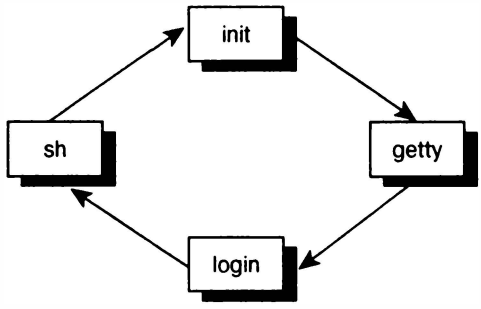


Рис. 2.7. Цикл регистрации в системе

Следует, однако, признать, что оболочка является обычной служебной программой. У нее нет никаких привилегий в системе, а это означает, что всякий

желающий, обладающий достаточным опытом и квалификацией, может создать свою оболочку. Именно поэтому существуют различные варианты или разновидности оболочек, включая прежнюю оболочку Bourne, разработанную Стивеном Борном; оболочку Korn, разработанную Дэвидом Корном; "возрожденную" оболочку Bourne, применяемую, главным образом, в системах Linux; оболочку C, разработанную Биллом Джоем. Все эти оболочки были предназначены для конкретных целей и обладают особыми возможностями и индивидуальными свойствами.

Обязанности оболочки

Теперь вы знаете, что оболочка осуществляет синтаксический анализ каждой вводимой вами строки и начинает выполнение избранной вами программы. Именно на стадии синтаксического анализа происходит интерпретация специальных символов расширения имен файлов вроде знака *. Но у оболочки имеются и другие обязанности, представленные на рис. 2.8.

Рис. 2.8. Обязанности оболочки

Выполнение программ

Оболочка отвечает за выполнение всех программ, которые запрашиваются с терминала. Всякий раз, когда вы вводите строку в оболочке, она анализирует эту строку и определяет порядок дальнейших действий. Что же касается самой

оболочки, то каждая строка в ней должна соответствовать следующему основному формату:

имя_программы аргументы

Строка, вводимая в оболочке, более формально называется *командной строкой*. Оболочка просматривает командную строку, определяя имя программы, которую следует выполнить, а также аргументы, которые необходимо передавать этой программе.

Чтобы выяснить, с чего начинается и чем оканчивается имя программы и каждый ее аргумент, в оболочке применяются специальные символы, которые называются *пробельными*. К их числу относится символ пробела, символ горизонтальной табуляции, а также символ окончания строки, формально называемый *символом новой строки*. Несколько следующих подряд пробельных символов игнорируются оболочкой. Когда вводится команда

```
mv    tmp/mazewars games
```

оболочка просматривает командную строку и интерпретирует все символы от начала строки и до первого пробельного символа как имя выполняемой программы (в данном случае — `mv`). Следующий далее пробел игнорируется, а ряд символов вплоть до последующего пробела интерпретируется как первый аргумент команды (в данном случае — `tmp/mazewars`). Все символы вплоть до последующего пробела, а им в рассматриваемом здесь примере оказывается символ новой строки, интерпретируются как второй аргумент команды (в данном случае — `games`). После синтаксического анализа командной строки оболочка переходит к выполнению команды `mv`, передавая ей указанные аргументы `tmp/mazewars` и `games` (рис. 2.9).

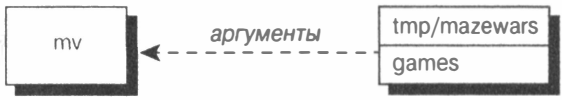


Рис. 2.9. Процесс выполнения команды **`mv`** с двумя аргументами

Как упоминалось ранее, несколько следующих подряд пробельных символов игнорируются оболочкой. Это означает, что когда оболочка обработает следующую командную строку:

```
echo        when    do    we    eat?
```

она передаст команде `echo` четыре аргумента: `when`, `do`, `we` и `eat?` (рис. 2.10).

Рис. 2.10. Процесс выполнения команды **echo** с четырьмя аргументами

Принимая свои аргументы, команда echo просто отображает их на терминале, и поэтому выделение каждого аргумента отдельным пробелом делает выводимый результат более удобочитаемым:

```
$ echo        when    do    we    eat?
when do we eat?
$
```

Оказывается, что команда echo вообще не замечает пустые пробелы, поскольку они "поглощаются" оболочкой в процессе синтаксического анализа командной строки. При обсуждении кавычек в главе 5 будет показано, как включать пустые пробелы в аргументы программ, но зачастую лишние пробелы требуется удалить.

Как упоминалось ранее, оболочка сначала осуществляет поиск указанной для выполнения программы до тех пор, пока не найдет ее на диске, а затем запрашивает у ядра Unix начало выполнения найденной программы. И, как правило, именно так и происходит. Но имеется ряд команд, в том числе cd, pwd и echo, которые фактически встроены в саму оболочку. Поэтому оболочка выясняет, является ли введенная команда встроенной, прежде чем искать ее на диске. И если это действительно встроенная команда, то она выполняется непосредственно.

Но оболочка выполняет еще кое-что, прежде чем вызвать отдельные программы. Поэтому обсудим далее, о каких действиях оболочки здесь идет речь.

Подстановка значений переменных и имен файлов

Как и более формальный язык программирования, оболочка позволяет присваивать значения переменным. Всякий раз, когда вы указываете одну из таких переменных в командной строке, предваряя ее знаком денежной единицы, оболочка подставляет значение, присвоенное переменной. Более подробно этот вопрос рассматривается в главе 4.

Кроме того, оболочка подставляет имена файлов в командной строке. В действительности оболочка просматривает командную строку в поисках символов *****, **?** или **[. . .]** для подстановки имени файла, прежде чем определить имя выполняемой программы и ее аргументы.

Допустим, в текущем каталоге содержатся следующие файлы:

```
$ ls
mrs.todd
prog1
shortcut
sweeney
$
```

А теперь воспользуемся подстановкой имен файлов (*) в команде echo следующим образом:

```
$ echo *                          Список всех файлов
mrs.todd prog1 shortcut sweeney
$
```

Сколько же аргументов было передано команде echo: один или четыре? Их должно быть четыре, поскольку оболочка выполняет подстановку имен файлов. Когда оболочка анализирует следующую строку:

```
echo *
```

она распознает специальный символ * и подставляет имена всех файлов из текущего каталога (и даже автоматически расставляет их в алфавитном порядке):

```
echo mrs.todd prog1 shortcut sweeney
```

Затем оболочка определяет аргументы, которые следует передать конкретной команде. Таким образом, команда echo вообще не замечает знак звездочки, т.е. тот факт, что в командной строке, по существу, были введены четыре аргумента (рис. 2.11).

Рис. 2.11. Процесс выполнения команды **echo**

Переадресация ввода-вывода

В обязанности оболочки входит также переадресация ввода и вывода. Она просматривает каждую введенную командную строку на наличие специальных символов **<**, **>** или **>>** для переадресации (любопытно, что для переадресации ввода имеется еще одна последовательность знаков, **<<**, как поясняется в главе 12).

Если ввести следующую команду:

```
echo Remember to record The Walking Dead > reminder
```

оболочка распознает специальный символ **>** для переадресации вывода и интерпретирует следующее слово в командной строке как имя файла, в который переадресовывается результат выполнения команды `echo` (в данном случае — это файл `reminder`). Если файл `reminder` существует и доступен для записи, то прежнее его содержимое перезаписывается. А если этот файл или его каталог *не* доступен для записи, то оболочка выдаст сообщение об ошибке.

Прежде чем начать выполнение требуемой программы, оболочка переадресовывает стандартный вывод из программы в указанный файл. Практически в каждом случае программе вообще неизвестно, что вывод результатов ее выполнения переадресовывается. Она просто направляет результаты своего выполнения в стандартный вывод, т.е. как обычно, на терминал, даже не подозревая, что оболочка переадресовывает эти данные в файл.

Рассмотрим еще один пример применения двух практически одинаковых команд:

```
$ wc -l users
      5 users
$ wc -l < users
      5
$
```

В первом случае оболочка определяет в ходе синтаксического анализа командной строки, что выполняемая программа носит имя `wc`, и передает ей два аргумента: **`-l`** и `users` (рис. 2.12).

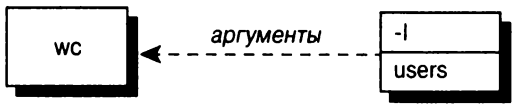


Рис. 2.12. Процесс выполнения команды **`wc -l users`**

Когда команда `wc` начинает свое выполнение, она обнаруживает, что ей переданы два аргумента. Первый аргумент, **`-l`**, предписывает ей подсчитать количество строк, а второй аргумент обозначает имя файла, в котором подсчитываются строки. Таким образом, команда `wc` открывает файл `users`, подсчитывает в нем количество строк и выводит полученный подсчет вместе с именем файла.

А во втором случае команда `wc` действует несколько иначе. Просматривая командную строку, оболочка обнаруживает символ **<** для переадресации ввода. Поэтому слово, которое следует после этого символа в командной строке, интерпретируется как имя файла, из которого переадресовывается ввод. Проанализировав выражение `< users` из командной строки, оболочка приступает к выполнению команды `wc`, переадресовывая ей стандартный ввод из файла `users` и передавая ей только один аргумент **`-l`** (рис. 2.13).

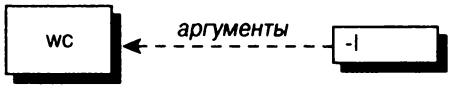


Рис. 2.13. Процесс выполнения команды **wc -l < users**

Когда команда wc начинает свое выполнение, она на этот раз замечает, что ей передан только один аргумент **-l**. А поскольку имя файла не указано, то команда wc решает, что строки, количество которых следует подсчитать, должны поступать не из файла, а из стандартного ввода. Таким образом, команда `wc -l` подсчитывает количество строк, даже не подозревая, что эти строки на самом деле поступают из файла `users`. Конечный результат выполнения этой команды отображается как обычно, но без имени файла, поскольку оно не было передано команде wc.

Очень важно уяснить отличия в выполнении обеих рассмотренных выше команд. Если эти отличия по каким-то причинам вам не ясны, внимательно прочитайте этот раздел еще раз, прежде чем перейти к следующему разделу.

Подключение конвейера

Оболочка просматривает командную строку в поисках не только символов для переадресации ввода-вывода, но и символа канала (**|**). При каждом совпадении с этим символом оболочка подключает стандартный вывод из предыдущей команды к стандартному вводу последующей команды, а затем начинает выполнение обеих программ.

Так, если ввести следующую строку:

```
who | wc -l
```

оболочка обнаружит символ канала, разделяющий команды who и wc. При этом она подключает стандартный вывод из первой команды к стандартному вводу второй команды, а затем начинает выполнение обеих команд. В результате своего выполнения команда who производит список зарегистрированных пользователей и направляет его в стандартный вывод, даже не подозревая, что это не терминал, а другая команда.

А когда выполняется команда wc, то она выясняет, что имя файла не указано, и поэтому подсчитывает строки из стандартного ввода, даже не подозревая, что стандартный ввод поступает не с терминала, а следует из результата, выводимого командой who.

Как станет ясно в дальнейшем, конвейер можно образовать не только из двух команд, но из сложной цепочки, состоящей из трех, четырех, пяти или большего числа команд. И хотя это не легко уяснить поначалу, тем не менее, едва ли не самый большой потенциал системы Unix кроется именно в конвейеризации команд.

Контроль окружения

Оболочка предоставляет такие команды, которые дают вам возможность специально настраивать свое окружение. В это окружение обычно входит ваш начальный каталог; символы, которые оболочка отображает, приглашая вас ввести команду; а также список каталогов, который следует искать всякий раз, когда вы запрашиваете выполнение программы. Подробнее о контроле своего окружения вы узнаете в главе 10.

Интерпретируемый язык программирования

У оболочки имеется свой встроенный язык программирования. Это *интерпретируемый* язык, а следовательно, оболочка анализирует каждый вводимый оператор своего языка и выполняет любые команды, которые она посчитает достоверными. Этим язык оболочки отличается от таких языков программирования, как C++ и Swift, где операторы, как правило, компилируются в исполняемый машинный код перед их выполнением.

Отлаживать и видоизменять программы, разработанные на интерпретируемых языках программирования, как правило, проще, чем компилируемые программы. Но в то же время они выполняются дольше, чем их компилируемые аналоги.

Для программирования на языке оболочки предоставляются средства, которые можно обнаружить в большинстве других языков программирования. В частности, язык оболочки является процедурно-ориентированным и предоставляет конструкции для организации циклов, операторы выбора, переменные и функции. В современных оболочках, основанных на стандарте IEEE POSIX, имеются и многие другие языковые средства, в том числе массивы, типы данных, встроенные арифметические операции.

3

Рабочие инструменты

В этой главе подробно описываются некоторые из наиболее употребительных инструментов для программирования на языке оболочки, в том числе команды `cut`, `paste`, `sed`, `tr`, `grep`, `uniq` и `sort`. Чем лучше вы освоите эти инструменты, тем проще вам будет писать эффективные сценарии оболочки.

Регулярные выражения

Прежде чем рассматривать инструменты для программирования на языке оболочки, необходимо разъяснить понятие *регулярных выражений*. Регулярные выражения применяются в самых разных командах системы Unix, включая `ed`, `sed`, `awk`, `grep`, и в меньше степени в редакторе `vi`. Они предоставляют удобный и согласованный способ указания *шаблонов* для сопоставления.

Как ни странно, но оболочка распознает ограниченную форму регулярных выражений в подстановке имен файлов. Напомним, что знак звездочки (**`*`**) обозначает совпадение с нулевым или бо́льшим количеством символов, знак вопроса (**`?`**) — с любым одиночным символом, а конструкция **`[...]`** — с любыми символами в квадратных скобках. Но подобные формы все же отличаются от более формальных регулярных выражений, рассматриваемых в этой главе. Например, оболочка интерпретирует знак **`?`** как совпадение с любым одиночным символом, тогда как в регулярном выражении для этой цели служит знак точки (**`.`**).

Настоящие регулярные выражения оказываются намного более сложными, чем те их формы, которые распознаются оболочкой, и поэтому вопросам составления по-настоящему сложных регулярных выражений посвящены целые книги. Впрочем, чтобы оценить истинный потенциал регулярных выражений, совсем не обязано быть их знатоком!

В этом разделе предполагается, что вы знакомы с такими строковыми редакторами, как `ex` или `ed`. Более подробные сведения об этих редакторах можно найти, обратившись за справкой к соответствующей оперативной странице руководства `man` по системе Unix.

Совпадение с любым одиночным символом: знак точки (.)

Знак точки (**`.`**) обозначает в регулярном выражении совпадение с любым одиночным символом, каким бы он ни был. Таким образом, регулярное выражение

`r.`

означает совпадение с буквой `r` и любым последующим символом.

А регулярное выражение

```
.x.
```

означает совпадение с буквой x, окруженной двумя любыми и необязательно одинаковыми символами.

Немало регулярных выражений можно продемонстрировать, используя ed — простой традиционный строковый редактор, появившийся вместе с системой Linux. Например, следующая команда редактора ed:

```
/ ... /
```

осуществляет прямой поиск в редактируемом файле первой строки, содержащей любые три символа, окруженные пробелами. Но прежде чем продемонстрировать применение данного выражения на конкретном примере, следует заметить, что сначала редактор ed отображает количество символов в файле (**248**) и что такие команды, как p (т.е. печатать), предваряются спецификатором диапазона, наиболее характерным примером которого служит последовательность символов `1,$`, обозначающая пределы от первой до последней строки в файле:

```
$ ed intro
248
1,$p                          Вывести все строки
The Unix operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the Unix system was to create an
environment that promoted efficient program
development.
```

Выше выведено содержимое нашего рабочего файла. А теперь опробуем некоторые регулярные выражения в следующем примере:

```
/ ... /                 Найти три символа, окруженные пробелами
The Unix operating system was pioneered by Ken
/                       Повторить последний поиск
Thompson and Dennis Ritchie at Bell Laboratories
1,$s/p.o/XXX/g          Заменить все комбинации символов p.o на XXX
1,$p                    Посмотреть, что из этого вышло
The Unix operating system was XXXneered by Ken
ThomXXXn and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the Unix system was to create an
environment that XXXmoted efficient XXXgram
development.
```

При первом поиске редактор ed начинает действовать с самого начала файла, обнаруживая в первой же строке последовательность символов " was ", совпадающую с указанным шаблоном, а затем выводя ее. При повторении поиска

(по команде / редактора ed) из файла выводится вторая строка, поскольку в ней с указанным шаблоном совпадает последовательность символов " and ".

В следующей далее команде подстановки s указывается, что все вхождения символа p, после которого следует любой одиночный символ и далее символ o, должны быть заменены комбинацией символов XXX. Спецификатор 1,$, предваряющий эту команду, указывает на то, что его следует применить ко всем строкам в файле, а подстановка обозначается в виде структуры *s/прежнее/новое/g*, где *s* — подстановка, знаки косой черты ограничивают *прежнее* и *новое* значения, а *g* — применение данной операции в каждой строке не один раз, а столько раз, сколько потребуется.

Совпадение с началом строки: знак вставки (^)

Если знак вставки (^) указывается в качестве первого символа в регулярном выражении, он обозначает совпадение с началом строки. Таким образом, с регулярным выражением

```
^George
```

символы George совпадают *лишь* в том случае, если они оказываются *в начале* строки. В терминологии регулярных выражений это называется *укоренением слева* по вполне очевидным причинам.

Рассмотрим следующий пример:

```
$ ed intro
248
/the/
>>in the late 1960s. One of the primary goals in
>>the design of the Unix system was to create an
/^the/                Найти строку, начинающуюся со слова the
the design of the Unix system was to create an
1,$s/^/>>/            Вставить знаки >> в начале каждой строки
1,$p
>>The Unix operating system was pioneered by Ken
>>Thompson and Dennis Ritchie at Bell Laboratories
>>in the late 1960s. One of the primary goals in
>>the design of the Unix system was to create an
>>environment that promoted efficient program
>>development.
```

В приведенном выше примере также демонстрируется, что регулярное выражение ^ может быть использовано для совпадения с началом строки. В данном случае оно используется для вставки знаков >> в начале каждой строки. А команда, подобная следующей:

```
1,$s/^/    /
```

обычно применяется для вставки пробелов в начале каждой строки (в данном случае — четырех пробелов).

Совпадение с концом строки: знак денежной единицы ($)

Подобно применению знака вставки (^) для совпадения с началом строки, знак денежной единицы ($) служит для совпадения с концом строки. Таким образом, со следующим регулярным выражением:

```
contents$
```

комбинация символов `contents` совпадает *лишь* в том случае, если они оказываются *в конце* строки. Как вы думаете, с чем совпадет приведенное ниже регулярное выражение?

```
.$
```

Совпадет ли с ним только знак точки в конце строки? Нет, не совпадет. Напомним, что знак точки обозначает совпадение с любым символом, и поэтому с ним совпадет любой символ в конце строки, включая и знак точки.

Как же добиться совпадения с самим знаком точки? В общем, если требуется добиться совпадения с любыми символами, имеющими специальное назначение в регулярном выражении, их следует предварить знаком обратной косой черты (\), чтобы переопределить их специальное назначение. Например, со следующим регулярным выражением:

```
\.$
```

совпадает любая строка, оканчивающаяся точкой, а с регулярным выражением

```
^\.
```

любая строка, начинающаяся с точки.

Если же в регулярном выражении требуется указать знак обратной косой черты как таковой, то его следует предварить еще одним знаком обратной косой черты (\\). Ниже приведены характерные примеры поиска символов на совпадение в конце строки.

```
$ ed intro
248
/\.$/                    Найти строку, оканчивающуюся точкой
development.
1,$s/$/>>/               Добавить знаки >> в конце каждой строки
1,$p
The Unix operating system was pioneered by Ken>>
Thompson and Dennis Ritchie at Bell Laboratories>>
in the late 1960s. One of the primary goals in>>
the design of the Unix system was to create an>>
environment that promoted efficient program>>
development.>>
1,$s/..$//               Удалить два символа из каждой строки
1,$p
```

```
The Unix operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the Unix system was to create an
environment that promoted efficient program
development.
```

Знаки **^** и **$** обычно применяются в следующем регулярном выражении:

```
^$
```

с которым совпадает любая строка, вообще не содержащая ни одного символа. Однако это регулярное выражение отличается от приведенного ниже регулярного выражения, с которым совпадает любая строка, состоящая из одного символа пробела.

```
^ $
```

Совпадение с набором символов: конструкция [. . .]

Допустим, что в процессе редактирования файла требуется найти первое вхождение символов the. В редакторе ed для этого достаточно ввести следующую команду:

```
/the/
```

В итоге прямой поиск в буфере редактора ed будет осуществляется до тех пор, пока не обнаружится строка, содержащая указанную последовательность символов. И первая же найденная строка отобразится в редакторе ed, как показано ниже.

```
$ ed intro
248
/the/                          Найти строку, содержащую слово the
in the late 1960s. One of the primary goals in
```

Обратите внимание на то, что первая строка файла содержит также слово The с теми же самыми символами, но оно начинается с прописной буквы T. Регулярное выражение для поиск слова the или The может быть создано с помощью набора символов, где знаки **[** и **]** служат для ограничения сопоставляемого набора символов. Например, со следующим регулярным выражением:

```
[tT]he
```

совпадут строчная или прописная буква t и следующие за ней символы he:

```
$ ed intro
248
/[tT]he/                          Найти слово the или The
The Unix operating system was pioneered by Ken
/                                 Продолжить поиск
```

```
in the late 1960s. One of the primary goals in
/                                  Повторить поиск еще раз
the design of the Unix system was to create an
1,$s/[aeiouAEIOU]//g               Удалить все гласные буквы
1,$p
Th nx prtng systm ws pnrd by Kn
Thmpsn nd Dnns Rtch t Bll Lbrtrs
n th lt 1960s. n f th prmry gls n
th dsgn f th nx systm ws t crt n
nvrnmnt tht prmtd ffcnt prgrm
dvlpmnt.
```

Обратите внимание на то, что в приведенном выше примере с шаблоном `[aeiouAEIOU]` совпадает единственная гласная буква английского алфавита (строчная или прописная). Но это довольно неуклюжее обозначение можно заменить диапазоном символов, указанных в квадратных скобках через дефис (**-**), разделяющий начальный и конечный символы в диапазоне. Так, для совпадения с символом любой цифры в пределах от **0** до **9** вместо регулярного выражения

```
[0123456789]
```

можно составить более лаконичное регулярное выражение:

```
[0-9]
```

Для совпадения с прописной буквой английского алфавита подойдет такое регулярное выражение:

```
[A-Z]
```

А для совпадения с прописной или строчной буквой английского алфавита — приведенное ниже регулярное выражение.

```
[A-Za-z]
```

Ниже приведены некоторые примеры применения регулярных выражений для поиска на совпадение с набором символов в редакторе ed.

```
$ ed intro
248
/[0-9]/                Найти строку, содержащую цифру
in the late 1960s. One of the primary goals in
/^[A-Z]/               Найти строку, начинающуюся с прописной буквы
The Unix operating system was pioneered by Ken
/                      Повторить поиск
Thompson and Dennis Ritchie at Bell Laboratories
1,$s/[A-Z]/*/g         Заменить все прописные буквы на знаки *
1,$p
*he *nix operating system was pioneered by *en
*hompson and *ennis *itchie at *ell *aboratories
in the late 1960s. *ne of the primary goals in
```

```
the design of the *nix system was to create an
environment that promoted efficient program
development.
```

Как поясняется ниже, знак звездочки имеет специальное назначение в регулярных выражениях. Но его не нужно экранировать знаком обратной косой черты в строке замены, указываемой в команде подстановки, поскольку эта строка замены составляется на другом языке выражений (как упоминалось ранее, с регулярными выражениями не все так просто).

В редакторе ed такие последовательности регулярных выражений, как *****, **.**, **[...]**, **$** и **^**, имеют смысл только в строке поиска, но не имеют никакого специального назначения в строке замены. Так, если знак вставки (**^**) оказывается первым символом после левой (открывающей) квадратной скобки, то его смысл меняется на *обратный*. (А в оболочке для этой же цели служит восклицательный знак **!**, указываемый вместе с наборами символов.) Например, со следующим регулярным выражением:

```
[^A-Z]
```

совпадет любой символ, *кроме* прописной буквы. Аналогично с регулярным выражением:

```
[^A-Za-z]
```

совпадет любой небуквенный символ. В качестве примера ниже демонстрируется удаление всех небуквенных символов из строк в рассматриваемом здесь тестовом файле.

```
$ ed intro
248
1,$s/[^a-zA-Z]//g              Удалить все небуквенные символы
1,$p
TheUnixoperatingsystemwaspioneeredbyKen
ThompsonandDennisRitchieatBellLaboratories
InthelatesOneoftheprimarygoalsin
ThedesignoftheUnixsystemwastocreatean
Environmentthatpromotedefficientprogram
development
```

Совпадение с нулевым или большим количеством символов: знак звездочки (*)

Знак звездочки (*****) применяется в оболочке при подстановке имен файлов для совпадения с нулевым или большим количеством символов. И при составлении регулярных выражений знак звездочки служит для совпадения с нулевым или большим числом вхождений *предшествующего* ему элемента регулярного выражения, который сам может быть регулярным выражением.

Например, со следующим регулярным выражением:

```
X*
```

совпадает нуль, одна, две, три заглавные букв X и больше, тогда как с регулярным выражением

```
XX*
```

совпадает одна заглавная буква X или больше, поскольку в данном выражении указан один символ X, после которого следует нуль или больше символов X. Того же самого результата можно добиться и с помощью знака "плюс" (+), который обозначает совпадение с *одним* предшествующим выражением *или больше*. Так, выражения XX* и X+ выполняют одну и ту же функцию.

Аналогичный шаблон нередко применяется для сопоставления с одним пустым пробелом или больше в строке, как демонстрируется в следующем примере:

```
$ ed lotsaspaces
85
1,$p
This         is    an example     of a
file    that   contains          a   lot
of    blank spaces                Заменить несколько пустых
                                  пробелов одиночными пробелами
1,$s/  */ /g
1,$p
This is an example of a
file that contains a lot
of blank spaces
```

В данном примере команда

```
1,$s/  */ /g
```

предписывает редактору ed заменить все вхождения пробела с последующим нулевым или большим количеством пробелов на единственный пробел — иными словами, сократить весь пробельный материал в тексте до одиночных пробелов. Если совпадение произойдет с одиночным пробелом, то никакой замены не последует. Но если произойдет совпадение, например, с тремя пробелами, то все они будут заменены одиночным пробелом.

Следующее регулярное выражение:

```
.*
```

нередко служит для обозначения нулевого или большего количества вхождений *любых* символов. Следует, однако, иметь в виду, что данное регулярное выражение сопоставляется с *самой длинной* символьной строкой, совпадающей с указанным в нем шаблоном. Следовательно, само это выражение всегда сопоставляется со *всей* строкой текста.

Еще одним примером сочетания знаков . и * служит следующее регулярное выражение:

```
e.*e
```

с которым совпадают все символы от первой до последней буквы e в строке.

Однако данное выражение совсем *не* обязательно совпадает *только* со строками, начинающимися и оканчивающимися на букву e. Ведь оно *не укоренено* ни слева, ни справа, т.е. *не* содержит знак ^ или $ в своем шаблоне. Ниже приведен характерный тому пример.

```
$ ed intro
248
1,$s/e.*e/+++/
1,$p
Th+++n
Thompson and D+++S
in th+++ primary goals in
th+++ an
+++nt program
d+++nt.
```

Рассмотрим еще одно любопытное регулярное выражение. Как вы думаете, с чем оно совпадет?

```
[A-Za-z][A-Za-z]*
```

Это регулярное выражение совпадет с любым буквенным символом, после которого следует нулевое или большее количество буквенных символов. Оно очень похоже на регулярное выражение, совпадающее со словами. Так, в приведенном ниже примере оно применяется для замены всех слов буквой X с сохранением всех пробелов и знаков препинания.

```
$ ed intro
248
1,$s/[A-Za-z][A-Za-z]*/X/g
1,$p
X X X X X X X X
X X X X X X X
X X X 1960X.   X X X X X X
X X X X X X X X X X
X X X X X
X.
```

С регулярным выражением в данном примере не совпала только числовая последовательность символов **1960**. Чтобы учесть и эту последовательность цифр как слово, регулярное выражение можно изменить следующим образом:

```
$ ed intro
248
1,$s/[A-Za-z0-9][A-Za-z0-9]*/X/g
1,$p
X X X X X X X X
X X X X X X X
X X X X.   X X X X X X
X X X X X X X X X X
X X X X X
X.
```

Данное регулярное выражение можно было бы расширить таким образом, чтобы учесть слова, которые пишутся через дефис или сокращенно (например, слово don't), но оставим это вам в качестве упражнения. Следует лишь заметить, что если требуется сопоставление с символом дефиса в наборе символов, заключаемых в квадратные скобки, то этот символ следует указать первым после левой (открывающей) квадратной скобки, но после знака обращения ^, если он присутствует в шаблоне, или же перед правой (закрывающей) скобкой, чтобы правильно его интерпретировать. Так, с любым из приведенных ниже регулярных выражений совпадает одиночный знак дефиса или цифры.

```
[-0-9]
[0-9-]
```

Аналогичным образом, если требуется сопоставление со знаком правой квадратной скобки, его следует указать первым после левой (открывающей) квадратной скобки, но после знака обращения ^, если он присутствует в шаблоне. Так, с регулярным выражением

```
[]a-z]
```

совпадает правая квадратная скобка или строчная буква английского алфавита.

Совпадение с точным количеством подшаблонов: конструкция \{ . . . \}

В предыдущих примерах было показано, как пользоваться знаком звездочки для обозначения *одного* или нескольких совпадающих вхождений предшествующего регулярного выражения. Например, следующее регулярное выражение:

```
XX*
```

означает совпадение с буквой X, после которой следует нулевое или большее количество вхождений той же самой буквы X. Аналогично регулярное выражение

```
XXX*
```

означает совпадение по меньшей мере с *двумя* подряд буквами X.

Но, откровенно говоря, это не очень удобно, и поэтому имеется более общий способ указать точное количество совпадающих символов, используя следующую конструкцию:

```
\{min, max\}
```

где ***min*** обозначает минимальное количество совпадающих вхождений предшествующего регулярного выражения, а ***max*** — максимальное. Следует, однако, иметь в виду, что фигурные скобки необходимо *экранировать* знаком обратной косой черты.

Так, со следующим регулярным выражением:

```
X\{1,10\}
```

совпадает от одной до десяти следующих подряд букв X. Всякий раз, когда появляется выбор, совпадение происходит с наибольшим шаблоном. Так, если входной текст содержит восемь букв X подряд, то именно с таким количеством букв и произойдет совпадение в приведенном выше регулярном выражении.

В качестве еще одного примера рассмотрим следующее регулярное выражение:

```
[A-Za-z]\{4,7\}
```

с которым совпадает последовательность букв длиной от четырех до семи символов.

А теперь попробуем осуществить подстановку в следующем примере, используя рассмотренную выше конструкцию:

```
$ ed intro
248
1,$s/[A-Za-z]\{4,7\}/X/g
1,$p
The X Xng X was Xed by Ken
Xn and X X at X XX
in the X 1960s.  One of the X X in
the X of the X X was to X an
XX X Xd Xnt X
XX.
```

Данный пример является особым случаем глобального поиска и замены в редакторе ed, а следовательно, и в редакторе vi, по шаблону *s/прежнее/новое/*. В данном случае этот шаблон предваряется указанным диапазоном 1,$ и завершается флагом g, чтобы многочисленные замены происходили в каждой строке там, где это уместно.

Особого внимания заслуживает ряд специальных случаев применения рассматриваемой здесь конструкции. Так, если в фигурных скобках указано только одно число, как в следующем выражении:

```
\{10\}
```

то оно обозначает, что предшествующее регулярное выражение должно совпадать именно *столько* раз. Так, со следующим регулярным выражением:

```
[a-zA-Z]\{7\}
```

точно совпадают семь буквенных символов, а с регулярным выражением

```
.\{10\}
```

десять символов, какими бы они ни были, как показано в приведенном ниже примере.

```
$ ed intro
248
1,$s/^.\{10\}//        Удалить первые 10 символов из каждой строки
1,$p
perating system was pioneered by Ken
nd Dennis Ritchie at Bell Laboratories
e 1960s. One of the primary goals in
of the Unix system was to create an
t that promoted efficient program
t.
1,$s/.\{5\}$//         Удалить последние 5 символов из каждой строки
1,$p
perating system was pioneered b
nd Dennis Ritchie at Bell Laborat
e 1960s. One of the primary goa
 of the Unix system was to crea
t that promoted efficient pr
t.
```

Обратите внимание на то, что в последней строке тестового файла не осталось пяти символов при выполнении последней команды подстановки. Поэтому совпадения с этой строкой не произошло, а следовательно, она была оставлена без изменения, поскольку в данной команде *точно* указано удалить пять символов.

Если в фигурных скобках указано единственное число, после которого следует запятая, то совпасть должно, по крайней мере, столько же вхождений предшествующего регулярного выражения, хотя верхний предел не установлен. Так, со следующим регулярным выражением:

```
+\{5,\}
```

совпадет по меньшей мере пять следующих подряд знаков "плюс". Если же во входных данных встречается больше пяти следующих подряд знаков "плюс", то совпадение происходит с наибольшим их количеством. Ниже приведен характерный пример применения подобной конструкции.

```
$ ed intro
248
1,$s/[a-zA-Z]\{6,\}/X/g                    Заменить на букву X слова,
                                           состоящие хотя бы из 6 букв
1,$p
The Unix X X was X by Ken
X and X X at Bell X
in the late 1960s. One of the X goals in
the X of the Unix X was to X an
X that X X X
X.
```

Сохранение совпавших символов: конструкция \(. . . \)

Заключив символы в круглые скобки, экранированные знаками обратной косой черты, можно ссылаться на них при сопоставлении с регулярным выражением. Эти зафиксированные символы сохраняются в предопределенных переменных, называемых *регистрами* синтаксического анализатора регулярных выражений и нумеруемых от **1** до **9**.

Такая конструкция кажется не совсем понятной и требует подробных разъяснений. В качестве первого примера рассмотрим следующее регулярное выражение:

```
^\(.\)
```

С этим выражением совпадает первый символ в строке, каким бы он ни был. При этом совпавший символ сохраняется в регистре 1.

Для извлечения символов, хранящихся к отдельном регистре, служит конструкция *n*, где *n* — цифра от **1** до **9**. Так, со следующим регулярным выражением:

```
^\(.\)\1
```

первоначально совпадает первый символ в строке, который сохраняется в регистре 1, а затем сопоставление происходит с тем, что хранится в регистре 1, как указано в конструкции \1. Конечным результатом данного регулярного выражения является совпадение первых двух символов в строке, при условии, что они *одинаковы*. Хитро придумано, не так ли?

А со следующим регулярным выражением:

```
^\(.\).*\1$
```

совпадают все строки, где первый символ (**^.**) такой же, как и последний (\1$). А шаблон **.*** означает совпадение со всеми промежуточными символами.

Проанализируем данное регулярное выражение по частям. Напомним, что знак **^** обозначает совпадение в начале строки, а знак **$** — в конце. В упрощенном виде шаблон **..*** обозначает первый символ в строке (первый знак **.**), после

которого следует остальная часть строки (.*). Чтобы сохранить первый символ в регистре 1, вводится обозначение \(\), а для ссылки на него — обозначение \1. Теперь вам должно быть ясно, каким образом действует данное регулярное выражение.

Если конструкция \(...\) следует несколько раз подряд, то совпавшие символы по очереди сохраняются в соответствующих регистрах. Так, если для сопоставления с каким-нибудь текстом применяется следующее регулярное выражение:

```
^\(...\)\(...\)
```

то первые три символа в строке сохраняются в регистре 1, а следующие три символа — в регистре 2. Если присоединить к этому выражению шаблон \2\1, то в итоге получится совпадение с 12-символьной строкой, где символы 1–3 совпадают с символами 10–12, а символы 4–6 — с символами 7–9.

Когда регистр применяется в команде подстановки редактора ed, к нему можно также обращаться из строки замены, что может быть очень эффективно, как показано в следующем примере:

```
$ ed phonebook
114
1,$p
Alice Chebba      973-555-2015
Barbara Swingle   201-555-9257
Liz Stachiw       212-555-2298
Susan Goldberg    201-555-7776
Tony Iannino      973-555-1295
1,$s/\(.*\)    \(.*\)/\2 \1/          Поменять оба поля местами
1,$p
973-555-2015 Alice Chebba
201-555-9257 Barbara Swingle
212-555-2298 Liz Stachiw
201-555-7776 Susan Goldberg
973-555-1295 Tony Iannino
```

Имена и номера телефонов отделяются друг от друга в файле phonebook одним знаком табуляции. Поэтому регулярное выражение

```
\(.*\)     \(.*\)
```

означает совпадение со всеми символами вплоть до первого знака табуляции, т.е. последовательность символов .* между знаками \(и \), которая присваивается регистру 1, а все совпавшие символы после знака табуляции присваиваются регистру 2. В следующей строке замены:

```
\2 \1
```

указывается содержимое регистра 2, после которого следует пробел и содержимое регистра 1.

Когда редактор ed применяет команду подстановки к первой строке файла:

```
Alice Chebba    973-555-2015
```

все, что совпадает вплоть до знака табуляции (Alice Chebba), сохраняется в регистре 1, а все, что совпадает после знака табуляции (973-555-2015), — в регистре 2. Сам же знак табуляции при этом теряется, поскольку он не заключен в круглые скобки в рассматриваемом здесь регулярном выражении. После этого редактор ed заменяет совпавшие символы (т.е. всю строку) содержимым регистра 2 (973-555-2015), пробелом и содержимым регистра 1 (Alice Chebba). И в конечном итоге получается следующая строка:

```
973-555-2015 Alice Chebba
```

Как видите, регулярные выражения являются весьма эффективными инструментальными средствами, допускающими сопоставление и манипулирование довольно сложными шаблонами. Хотя порой они выглядят довольно запутанно!

В табл. 3.1 сведены специальные символы, распознаваемые в регулярных выражениях, чтобы помочь вам лучше понимать чужие регулярные выражения и научиться составлять собственные.

Таблица 3.1. Специальные символы, распознаваемые в регулярных выражениях

Обозначение	Назначение	Пример	С чем совпадает
.	*Любой* символ	**a..**	Символ **a** и два любых последующих символа
^	Начало строки	**^wood**	Слово **wood**, если оно появляется в начале строки
$	Конец строки	**x$**	Буква **x**, если это последний символ в строке
		^INSERT$	Строка, содержащая только символы **INSERT**
		^$	Строка, не содержащая никаких символов
*	Нулевое и большое количество вхождений предшествующего регулярного выражения	**x***	Нулевое или большее количество следующих подряд букв **x**
		xx*	Одна или больше следующих подряд букв **x**
		.*	Нулевое или большее количество *любых* символов
		w.*s	Буква **w**, нулевое или большее количество *любых* символов и буква **s**

Окончание табл. 3.1

Обозначение	Назначение	Пример	С чем совпадает
`+`	Одно или больше вхождений предшествующего регулярного выражения	`x+`	Одна или больше следующих подряд букв `x`
		`xx+`	Две или больше следующих подряд буквы `x`
		`.+`	Один символ или больше
		`w.+s`	Буква `w`, один или больше символов и буква `s`
`[`*символы*`]`	Любые указанные *символы*	`[tT]`	Строчная или прописная буква `t`
		`[a-z]`	Любая строчная буква английского алфавита
		`[a-zA-Z]`	Любая строчная или прописная буква английского алфавита
`[^`*символы*`]`	Что угодно, только не указанные *символы*	`[^0-9]`	Любой нечисловой символ
		`[^a-zA-Z]`	Любой небуквенный символ
`\{`*min*`, `*max*`\}`	Количество вхождений предшествующего регулярного выражения в переделах от *min* до *max*	`x\{1,5\}`	От 1 до 5 букв `x`
		`[0-9]\{3,9\}`	От 3 до 9 следующих подряд цифр
		`[0-9]\{3\}`	Точно 3 цифры
		`[0-9]\{3,\}`	Минимум 3 цифры
`\(...\)`	Сохранение в регистрах 1–9 совпавших символов, указанных в круглых скобках	`^\(.\)`	Первый символ в строке, сохраняемый в регистре 1
		`^\(.\)\1`	Первый и второй символы в строке, сохраняемые в регистре 1, если они одинаковы
		`^\(.\)\(.\)`	Первый и второй символы в строке, сохраняемые в регистрах 1 и 2 соответственно

Команда cut

В этом разделе рассматривается очень полезная команда cut, которая очень удобна для извлечения (буквально "вырезания") различных полей данных из файла или результата, выводимого другой командой. Общая форма команды cut выглядит следующим образом:

```
cut -cсимволы файл
```

где *символы* — количество символов (по местоположению), которые требуется извлечь из каждой строки, которую содержит указанный *файл*. Этот набор символов может состоять из одного числа, например, -c5 для извлечения пятого символа из каждой строки входных данных; разделенного запятыми списка чисел, например, -c1, 13, 50 для извлечения символов 1, 13 и 50; или же диапазона чисел, указываемых через дефис, например, -c20-50 для извлечения символов в пределах от 20 до 50 включительно. Чтобы извлечь символы в конце строки, второе число в указываемом диапазоне можно опустить. Так, по команде

```
cut -c5- data
```

извлекаются пять символов, отсчитываемых от конца каждой строки в файле data, а полученный результат направляется в стандартный вывод.

Если же *файл* не указан, команда cut читает свои исходные данные из стандартного ввода. Это означает, что ее можно использовать в качестве фильтра в конвейере.

Рассмотрим еще раз результат, выводимый командой who:

```
$ who
root     console   Feb 24  08:54
steve    tty02     Feb 24  12:55
george   tty08     Feb 24  09:15
dawn     tty10     Feb 24  15:55
$
```

Как видите, в системе зарегистрированы четыре пользователя. Допустим, требуется узнать только имена зарегистрированных пользователей, не обращая внимания на их терминалы или время входа в систему. Чтобы извлечь только имена зарегистрированных пользователей из результата выполнения команды who, можно воспользоваться командой cut следующим образом:

```
$ who | cut -c1-8              Извлечь первые 8 символов
root
steve
george
dawn
$
```

Параметр **-c1-8** команды cut указывает на то, что из каждой строки входных данных следует извлечь от 1 до 8 символов и направить их в стандартный вывод.

В следующем примере демонстрируется порядок присоединения команды sort в конце конвейера из предыдущего примера с целью получить отсортированный список зарегистрированных пользователей:

```
$ who | cut -c1-8 | sort
dawn
george
root
steve
$
```

Следует заметить, что это первый трехкомандный конвейер в рассмотренных до сих пор примерах. Ясно понимая принцип подключения выхода предыдущей команды на вход последующей, нетрудно образовать логический конвейер из трех, четырех и большего числа команд.

Если же требуется просмотреть применяемые в настоящий момент физические, псевдотерминалы или виртуальные терминалы, из результата, выводимого командой who, можно извлечь только поле tty, как показано ниже.

```
$ who | cut -c10-16
console
tty02
tty08
tty10
$
```

А откуда известно, что команда who отображает сведения о терминалах на позициях символов 10–16? Выяснить это совсем не трудно! Достаточно выполнить команду who на своем терминале и подсчитать позиции соответствующих символов.

Используя команду cut, можно извлечь столько различных символов, сколько потребуется. В следующем примере команда cut используется для отображения только имен всех зарегистрированных пользователей и времени их входа в систему:

```
$ who | cut -c1-8,18-
root Feb 24 08:54
steve Feb 24 12:55
george Feb 24 09:15
dawn Feb 24 15:55
$
```

Параметр **-c1-8,18-** определяет точное количество символов от 1 до 8 (т.е. имя пользователя), а также символы 18 и до конца строки (т.е. время входа в систему).

Параметры -d и -f

Команда cut с параметром **-c** удобна для извлечения данных из файла или результатов выполнения другой команды, но при условии, что формат файла или вывода из другой команды фиксирован.

В частности, командой cut можно пользоваться вместе с командой who потому, что заранее известно, что имена зарегистрированных пользователей отображаются на позициях 1–8, терминалы — на позициях 10–16, а время входа в систему — на позициях 18–29. Но, к сожалению, не все данные настолько хорошо организованы!

Рассмотрим в качестве примера содержимое следующего файла /etc/passwd:

```
$ cat /etc/passwd
root:*:0:0:The Super User:/:/usr/bin/ksh
cron:*:1:1:Cron Daemon for periodic tasks:/:
bin:*:3:3:The owner of system files:/:
uucp:*:5:5::/usr/spool/uucp:/usr/lib/uucp/uucico
asg:*:6:6:The Owner of Assignable Devices:/:
steve:*.:203:100::/users/steve:/usr/bin/ksh
other:*:4:4:Needed by secure program:/:
$
```

Файл /etc/passwd является главным местом, где содержатся имена всех пользователей в системе. Он содержит и другие сведения, в том числе идентификаторы пользователей, пути к их начальным каталогам, а также имена программ, запускаемых на выполнение при входе конкретных пользователей в систему.

Очевидно, что данные в этом файле организованы не так хорошо, как в результате выполнения команды who. Следовательно, извлечь список всех пользователей системы из этого файла не удастся по команде cut с параметром **-c**.

Но если проанализировать содержимое этого файла более внимательно, то можно заметить, что поля в нем разделяются двоеточием. И хотя эти поля отличаются по длине от одной строки к другой, тем не менее, можно определить положение знаков двоеточия, чтобы извлечь одно и то же поле из каждой строки.

Если данные разделяются конкретным символом, то для их извлечения по команде cut можно воспользоваться параметрами **-d** и **-f**. В частности, параметр **-d** определяет разделитель полей, а параметр **-f** — одно или несколько извлекаемых полей. И в таком случае для вызова команды cut служит следующая форма:

```
cut -ddchar -fполя файл
```

где *dchar* — символ, разделяющий поля данных, а *поля* — извлекаемые поля, которые содержит указанный *файл*. Номера полей начинаются с **1**, и для обозначения номеров полей может быть использован такой же тип формата, как и прежде для позиций символов (например, -f1, 2, 8, -f1-3, -f4-).

Чтобы извлечь имена всех пользователей из файла /etc/passwd, достаточно ввести следующую команду:

```
$ cut -d: -f1 /etc/passwd                    Извлечь поле 1
root
cron
bin
uucp
asg
steve
other
$
```

Если начальный каталог каждого пользователя находится в поле 6, то имена пользователей системы можно вывести вместе с их начальными каталогами следующим образом:

```
$ cut -d: -f1,6 /etc/passwd                  Извлечь поля 1-6
root:/
cron:/
bin:/
uucp:/usr/spool/uucp
asg:/
steve:/users/steve
other:/
$
```

Если команда cut предназначается для извлечения полей из файла, а параметр **-d** не указан, то по умолчанию в качестве разделителя полей в ней используется символ табуляции. В следующем примере демонстрируется типичное препятствие, скрывающееся на пути применения команды cut. Допустим, имеется файл phonebook со следующим содержимым:

```
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Jeff Goldberg       201-555-3378
Liz Stachiw         212-555-2298
Susan Goldberg      201-555-7776
Tony Iannino        973-555-1295
$
```

Если требуется получить только имена абонентов из телефонного справочника, то первым побуждением станет воспользоваться командой cut следующим образом:

```
$ cut -c1-15 phonebook
Alice Chebba     97
Barbara Swingle
Jeff Goldberg    2
```

```
Liz Stachiw      212
Susan Goldberg
Tony Iannino     97
$
```

Но ведь это совсем не то, что нужно! А произошло это потому, что имена абонентов отделяются от номеров телефонов символом табуляции, а не рядом пробелов. Что же касается команды `cut`, то символы табуляции считаются в ней как один символ, если она применяется с параметром **`-c`**. Именно поэтому команда `cut` извлекает в рассматриваемом здесь примере первые 15 символов из каждой строки, выводя представленный выше результат.

В тех случаях, когда поля данных разделяются символами табуляции, команду `cut` следует применять с параметром **`-f`**, как показано ниже. Напомним, что указывать разделитель полей вместе с параметром **`-d`** необязательно, поскольку по умолчанию в команде `cut` для разделения полей употребляется символ табуляции.

```
$ cut -f1 phonebook
Alice Chebba
Barbara Swingle
Jeff Goldberg
Liz Stachiw
Susan Goldberg
Tony Iannino
$
```

А как узнать заранее, что поля данных разделяются пробельными символами или символами табуляции? С одной стороны, это можно сделать методом проб и ошибок, как было показано ранее. А с другой стороны, можно ввести с терминала следующую команду:

```
sed -n l файл
```

Если поля данных разделяются символами табуляции, то вместо табуляции отобразится комбинация символов `\t`:

```
$ sed -n l phonebook
Alice Chebba\t973-555-2015
Barbara Swingle\t201-555-9257
Jeff Goldberg\t201-555-3378
Liz Stachiw\t212-555-2298
Susan Goldber\t201-555-7776
Tony Iannino\t973-555-1295
$
```

Как следует из приведенного выше результата, имя каждого абонента отделяется от каждого номера телефона символом табуляции. Более подробно потоковый редактор `sed` рассматривается далее в этой главе.

Команда paste

Команда paste является полной противоположностью команды cut. Вместо разделения строк на части она собирает их вместе. Общая форма команды paste выглядит следующим образом:

```
paste файлы
```

где соответствующие строки из указанных *файлов* "склеиваются" или объединяются вместе, образуя единые строки, которые затем направляются в стандартный вывод. Для обозначения исходных данных из стандартного ввода в заданной последовательности *файлов* может быть указан знак дефиса (-).

Допустим, в файле names содержится список имен абонентов:

```
$ cat names
Tony
Emanuel
Lucy
Ralph
Fred
$
```

Допустим также, что в другом файле numbers содержатся соответствующие номера телефонов для каждого абонента из файла names:

```
$ cat numbers
(307) 555-5356
(212) 555-3456
(212) 555-9959
(212) 555-7741
(212) 555-0040
$
```

Чтобы вывести имена абонентов вместе с номерами их телефонов, можно воспользоваться командой paste. Каждая строка из файла names отображается вместе со строкой из файла numbers, как показано ниже.

```
$ paste names numbers                    Объединить файлы вместе
Tony        (307)  555-5356
Emanuel     (212)  555-3456
Lucy        (212)  555-9959
Ralph       (212)  555-7741
Fred        (212)  555-0040
$
```

В следующем примере демонстрируется объединение строк из трех файлов по команде paste:

```
$ cat addresses
55-23 Vine Street, Miami
39 University Place, New York
17 E. 25th Street, New York
38 Chauncey St., Bensonhurst
17 E. 25th Street, New York
$ paste names addresses numbers
Tony      55-23 Vine Street, Miami         (307)  555-5356
Emanuel   39 University Place, New York    (212)  555-3456
Lucy      17 E. 25th Street, New York      (212)  555-9959
Ralph     38 Chauncey St., Bensonhurst     (212)  555-7741
Fred      17 E. 25th Street, New York      (212)  555-0040
$
```

Параметр -d

Если поля в выводимом результате не требуется разделять символами табуляции, то с помощью параметра **-d** можно указать другой разделитель в следующем формате:

```
-dchars
```

где *chars* — один или несколько символов, употребляемых для разделения объединяемых строк. Это означает, что первый из перечисленных в списке *chars* символов будет использоваться для отделения строк из первого файла, которые объединяются вместе со строками из второго файла; второй символ — для отделения строк из второго файла от строк из третьего файла и т.д.

Если задано больше файлов, чем символов, перечисленных в списке *chars*, команда `paste` автоматически переходит в начало этого списка, выбирая оттуда символы для разделения объединяемых вместе строк. В простейшей форме параметр **-d** указывает единственный символ для разделения *всех* объединяемых полей в выводимом результате, как показано в следующем примере:

```
$ paste -d'+' names addresses numbers
Tony+55-23 Vine Street, Miami+(307) 555-5356
Emanuel+39 University Place, New York+(212) 555-3456
Lucy+17 E. 25th Street, New York+(212) 555-9959
Ralph+38 Chauncey St., Bensonhurst+(212) 555-7741
Fred+17 E. 25th Street, New York+(212) 555-0040
```

Следует, однако, иметь в виду, что символы разделения надежнее заключать в одиночные кавычки, как показано выше. Вескость основания для этого поясняется далее.

Параметр -s

Параметр **-s** предписывает команде `paste` объединить вместе строки из одного и того же файла, а не из разных файлов. Если же указан лишь один файл, в результате будут объединены все строки из заданного файла, разделенные

символами табуляции или же символом разделения, указанным в параметре **-d**. В следующих примерах наглядно демонстрируется употребление параметров **-s** и **-d** в команде paste:

```
$ paste -s names                    Объединить все строки из файла names
Tony    Emanuel    Lucy    Ralph    Fred
$ ls | paste -d' ' -s -             Объединить строки из результата,
                                    выводимого командой ls, используя
                                    пробел в качестве разделителя
addresses intro lotsaspaces names numbers phonebook
$
```

В первом примере результат, выводимый командой ls, передается по каналу команде paste, объединяющей вместе строки (параметр **-s**) из стандартного ввода (знак **-**), разделяя каждое поле пробелом (параметр **-d' '**). Как упоминалось в главе 1, по команде

```
echo *
```

перечисляются также все файлы в текущем каталоге. Возможно, это несколько более простой вариант, чем ls | paste.

Команда sed

Команда sed представляет собой программу, применяемую для редактирования данных в канале или последовательности команд. Название команды sed сокращенно обозначает *потоковый редактор* (stream editor). В отличие от строкового редактора ed, потоковый редактор sed нельзя употреблять в диалоговом режиме, хотя их команды одинаковы. Общая форма команды sed выглядит следующим образом:

```
sed команда файла
```

где *команда* — это команда в стиле редактора ed, применяемая к *каждой* строке, указанной в заданном *файле*. Если же *файл* не указан, то предполагается стандартный ввод.

Применяя одну или несколько указанных команд к каждой строке исходных данных, редактор sed направляет результаты в стандартный вывод. Рассмотрим применение этого редактора на конкретном примере. Обратимся снова к тестовому файлу intro:

```
$ cat intro
The Unix operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the Unix system was to create an
environment that promoted efficient program
```

```
development.
$
```

Допустим, в этом файле требуется заменить все вхождения слова "Unix" словом "UNIX". Это нетрудно сделать в редакторе sed следующим образом:

```
$ sed 's/Unix/UNIX/' intro              Заменить слово Unix на UNIX
The UNIX operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the UNIX system was to create an
environment that promoted efficient program
development.
$
```

Выработайте в себе привычку заключать команду редактора sed в одиночные кавычки. В дальнейшем будет разъяснено, когда лучше употребить кавычки, а когда вместо них — двойные кавычки.

Команда s/Unix/UNIX/ редактора sed применяется к каждой строке из файла intro. Независимо от того, видоизменяется ли строка или нет, она направляется в стандартный вывод. А поскольку они выводятся в потоке данных, то редактор sed не вносит никаких изменений в первоначальный входной файл.

Чтобы сделать изменения постоянными, результат, выводимый из редактора sed, следует перенаправить во временный файл, а затем заменить первоначальный файл на вновь созданный следующим образом:

```
$ sed 's/Unix/UNIX/' intro > temp       Внести изменения
$ mv temp intro                         И сделать их постоянными
$
```

Прежде чем перезаписывать первоначальный файл, следует убедиться в правильности внесенных изменений. С этой целью полезно просмотреть временный файл temp по команде cat и только после этого выполнять команду mv для перезаписи первоначального файла.

Если слово "Unix" встречается в строке текста неоднократно, то в приведенном выше примере команда редактора sed заменит только первое вхождение этого слова в строке. Если же требуется заменить все вхождения слова "Unix", то в конце команды подстановки s следует дополнительно указать *глобальный* параметр **g**. В данном случае команда редактора sed примет следующий вид:

```
$ sed 's/Unix/UNIX/g' intro > temp
```

А теперь допустим, что из результата выполнения команды who требуется извлечь только имена файлов. Как вам должно быть уже известно, это можно, с одной стороны, сделать по команде cut следующим образом:

```
$ who | cut -c1-8
root
ruth
steve
pat
$
```

А с другой стороны, для удаления всех символов после первого пробела, обозначающего конец имени пользователя, и до конца строки можно применить регулярное выражение в редакторе sed:

```
$ who | sed 's/ .*$//'
root
ruth
steve
pat
$
```

Приведенная выше команда редактора sed заменяет пробел и следующие за ним любые символы до конца строки (**.*$**) *ничем* (**//**), т.е. удаляет их до конца каждой строки исходных данных.

Параметр -n

По умолчанию редактор sed направляет каждую строку в стандартный вывод, изменяется ли она или нет. Но иногда редактором sed требуется воспользоваться для извлечения лишь конкретных строк из файла. Именно для этой цели служит параметр **-n**, который сообщает редактору sed, что по умолчанию не нужно выводить любые строки. А в сочетании с командой p можно вывести те строки, которые совпадают с указанным диапазоном или шаблоном. Например, вывести только первые две строки из файла:

```
$ sed -n '1,2p' intro              Вывести только первые 2 строки
The UNIX operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
$
```

Если вместо номеров строк команде p предшествует последовательность символов, заключаемых в знаки косой черты, то редактор sed выведет лишь те строки из стандартного ввода, которые совпадают с указанным шаблоном. В следующем примере демонстрируется применение редактора sed для отображения только тех строк, которые содержат конкретную символьную строку:

```
$ sed -n '/UNIX/p' intro           Вывести только строки,
                                   содержащие слово UNIX
The UNIX operating system was pioneered by Ken
the design of the UNIX system was to create an
$
```

Удаление строк

Чтобы удалить строки, можно воспользоваться командой d редактора sed. Указав в этой команде один или ряд номеров строк, можно удалить конкретные строки из исходных данных. В следующем примере редактор sed применяется для удаления двух первых строк текста из тестового файла intro:

```
$ sed '1,2d' intro        Удалить строки 1 и 2
in the late 1960s. One of the primary goals in
the design of the UNIX system was to create an
environment that promoted efficient program
development.
$
```

Напомним, что по умолчанию редактор sed направляет все строки в стандартный вывод. Поэтому в данном примере весь оставшийся текст, начиная с третьей строки и до конца файла, будет направлен в стандартный вывод.

Предварив шаблоном команду d редактора sed, можно удалить из файла все строки, содержащие текст, совпадающий с этим шаблоном. В следующем примере редактор sed применяется для удаления из тестового файла всех строк, содержащих слово UNIX:

```
$ sed '/UNIX/d' intro                    Удалить все строки,
                                         содержащие слово UNIX
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
environment that promoted efficient program
development.
$
```

Истинный потенциал и удобство редактора sed выходит далеко за рамки приведенных выше примеров. Доступные в редакторе sed средства позволяют организовать цикл, вставить текст в буфер и употребить сочетания многих команд в одном сценарии редактирования. Некоторые дополнительные примеры применения редактора sed представлены в табл. 3.2.

Таблица 3.2. Примеры применения редактора sed

Команда редактора sed	Описание
`sed '5d'`	Удалить пятую строку
`sed '/[Tt]est/d'`	Удалить все строки, содержащие слово **Test** или **test**
`sed -n '20,25p' text`	Вывести из текста только строки 20-25
`sed '1,10s/unix/UNIX/g' intro`	Заменить слово **unix** на **UNIX** везде, где оно появляется в первых десяти строках из файла **intro**

Окончание табл. 3.2

Команда редактора sed	Описание
`sed '/jan/s/-1/-5/'`	Заменить первое значение `-1` на `-5` во всех строках, содержащих слово `jan`
`sed 's/...//' data`	Удалить первые три символа из каждой строки в файле `data`
`sed 's/...$//' data`	Удалить последние три символа из каждой строки в файле `data`
`sed -n 'l' text`	Вывести все строки из текста, отобразив непечатаемые знаки как `\nn`, где *nn* — восьмеричное значение символа, а символы табуляции — как `\t`

Команда `tr`

Команда `tr` служит в качестве фильтра для преобразования символов из стандартного ввода. Общая форма команды `tr` выглядит следующим образом:

```
tr исходные_символы целевые_символы
```

где *исходные_символы* и *целевые_символы* — один символ, несколько или целый набор соответствующих символов. Любой символ, который содержат *исходные_символы* во входных данных, будет преобразован в соответствующий символ, включаемый в *целевые_символы*. А результат этого преобразования направляется в стандартный вывод.

В своей простейшей форме команда `tr` может быть использована для преобразования одного символа в другой. Напомним еще раз содержимое тестового файла `intro`, применяемого в примерах из этой главы:

```
$ cat intro
The UNIX operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the UNIX system was to create an
environment that promoted efficient program
development.
$
```

В следующем примере демонстрируется применение команды `tr` для преобразования всех букв `e` в буквы `x`:

```
$ tr e x < intro
Thx UNIX opxrating systxm was pionxxrxd by Kxn
Thompson and Dxnnis Ritchix at Bxll Laboratorixs
in thx latx 1960s. Onx of thx primary goals in
thx dxsign of thx UNIX systxm was to crxatx an
```

```
xnvironmxnt that promotxd xfficixnt program
dxvxlopmxnt.
$
```

Входные данные команды `tr` должны быть переадресованы из файла `intro`, поскольку команда `tr` всегда предполагает получить свои входные данные из стандартного ввода. Результаты преобразования направляются в стандартный вывод, не затрагивая первоначальный файл. Приводя следующий, более практический пример, напомним о конвейере, применявшемся для извлечения имен файлов и начальных каталогов всех зарегистрированных в системе пользователей.

```
$ cut -d: -f1,6 /etc/passwd
root:/
cron:/
bin:/
uucp:/usr/spool/uucp
asg:/
steve:/users/steve
other:/
$
```

Знаки двоеточия из приведенного выше результата можно преобразовать в символы табуляции, чтобы получить более удобочитаемый результат, просто присоединив команду `tr` в конце конвейера следующим образом:

```
$ cut -d: -f1,6 /etc/passwd | tr : '    '
root        /
cron        /
bin         /
uucp        /usr/spool/uucp
asg         /
steve      /users/steve
other     /
$
```

В приведенном выше примере символ табуляции заключен в одиночные кавычки. И хотя он не отображается, можете нам поверить, что он присутствует. Этот символ следует заключить в одиночные кавычки, чтобы исключить его синтаксический анализ и удаление как лишнего пробела в оболочке.

Для обращения с непечатаемыми символами можно указать их восьмеричное представление в команде `tr`, используя следующую форму:

```
\nnn
```

где *nnn* — восьмеричное представление символа. Такой способ применяется нечасто, но его все же следует взять на вооружение, поскольку он удобен для представления непечатаемых символов.

Например, восьмеричное значение символа табуляции равно **11**. Следовательно, преобразовать знаки двоеточия в символы табуляции с помощью команды `tr` можно и таким способом:

```
tr : '\11'
```

В табл. 3.3 перечислены символы, которые нередко требуется представить в восьмеричной форме.

Таблица 3.3. Восьмеричные значения некоторых символов в коде ASCII

Символ	Восьмеричное значение
Звонковая сигнализация	7
Возврат на одну позицию	10
Табуляция	11
Новая строка	12
Перевод строки	12
Перевод страницы	14
Возврат каретки	15
Переход	33

В приведенном ниже примере команда `tr` принимает результат, выводимый командой `date`, и преобразует все пробелы в знаки новой строки. Таким образом, каждое поле из конечного результата появляется в отдельной строке.

```
$ date | tr ' ' '\12'                    Преобразовать пробелы
                                         в знаки новой строки
Sun
Jul
28
19:13:46
EDT
2002
$
```

С помощью команды `tr` можно также преобразовывать диапазоны символов. Так, в следующем примере демонстрируется преобразование всех строчных букв в прописные буквы текста из файла `intro`:

```
$ tr '[a-z]' '[A-Z]' < intro
THE UNIX OPERATING SYSTEM WAS PIONEERED BY KEN
THOMPSON AND DENNIS RITCHIE AT BELL LABORATORIES
IN THE LATE 1960S.  ONE OF THE PRIMARY GOALS IN
THE DESIGN OF THE UNIX SYSTEM WAS TO CREATE AN
ENVIRONMENT THAT PROMOTED EFFICIENT PROGRAM
DEVELOPMENT.
$
```

В данном примере диапазоны символов [a-z] и [A-Z] заключаются в кавычки, чтобы оболочка не интерпретировала их как шаблон. Попробуйте выполнить приведенную выше команду tr без кавычек, и вы сразу же получите совсем не тот результат, какой ожидали.

Поменяв местами оба аргумента в приведенной выше команде tr, можно преобразовать все прописные буквы в строчные:

```
$ tr '[A-Z]' '[a-z]' < intro
the unix operating system was pioneered by ken
thompson and dennis ritchie at bell laboratories
in the late 1960s.  one of the primary goals in
the design of the unix system was to create an
environment that promoted efficient program
development.
$
```

В качестве более занимательного примера догадайтесь, к какому результату приведет выполнение следующей команды:

```
tr '[a-zA-Z]' '[A-Za-z]'
```

Догадались? В данном случае прописные буквы преобразуются в строчные, а строчные — в прописные.

Параметр -s

Указав параметр -s, можно "уплотнить" многие вхождения символов в *целевые_символы*. Иными словами, если заданные *исходные_символы* содержат несколько экземпляров отдельного символа, то после преобразования они заменяются единственным экземпляром данного символа.

Так, в следующем примере команда tr преобразует все знаки двоеточия в символы табуляции, заменяя несколько символов табуляции одним:

```
tr -s ':'  '\11'
```

Таким образом, один или несколько знаков двоеточия, следующих подряд во входных данных, заменяются *единственным* символом табуляции в выводимом результате. Следует заметить, что вместо восьмеричного значения '\11' символ табуляции можно обозначить как '\t'. Можете опробовать такой способ, чтобы добиться большей удобочитаемости приведенной выше команды tr!

Допустим, имеется файл lotsaspaces со следующим содержимым:

```
$ cat lotsaspaces
This         is     an example    of a
file     that contains     a   lot
of    blank  spaces.
$
```

Используя команду tr с параметром **-s** и указав одиночный пробел в качестве первого и второго аргументов, можно уплотнить многочисленные пробелы в приведенном выше файле следующим образом:

```
$ tr -s ' ' ' ' < lotsaspaces
This is an example of a
file that contains a lot
of blank spaces.
$
```

Данная команда tr, по существу, означает следующее: преобразовать последовательности одних пробелов в другой пробел, заменив многие пробелы в выводимом результате единственным пробелом.

Параметр -d

С помощью команды tr можно также удалить отдельные символы из потока ввода. Для этой цели служит следующая форма команды tr:

```
tr -d исходные_символы
```

где любой символ, включаемый в *исходные_символы*, будет удален из стандартного ввода. В следующем примере по команде tr с параметром -d из файла intro удаляются все пробелы:

```
$ tr -d ' ' < intro
TheUNIXoperatingSystemwaspioneeredbyKen
ThompsonandDennisRitchieatBellLaboratories
inthelate1960s.Oneoftheprimarygoalsin
thedesignoftheUNIXSystemwastocreatean
environmentthatpromotedefficientprogram
development.
$
```

Вы, вероятно, уже сообразили, что той же самой цели можно добиться и с помощью редактора sed, как показано ниже.

```
$ sed 's/ //g' intro
TheUNIXoperatingsystemwaspioneeredbyKen
ThompsonandDennisRitchieatBellLaboratories
inthelate1960s.Oneoftheprimarygoalsin
thedesignoftheUNIXsystemwastocreatean
environmentthatpromotedefficientprogram
development.
$
```

Для системы Unix характерно наличие нескольких подходов к решению одной и той же задачи. В рассмотренном здесь примере любой из приведенных выше подходов (т.е. применение команды tr или sed) вполне пригоден. Тем не менее в данном случае лучше выбрать команду tr, поскольку она выглядит более лаконично и выполняется быстрее.

В табл. 3.4 приведены характерные примеры применения команды tr для преобразования и удаления символов. Но следует иметь в виду, что команда tr манипулирует только *одиночными* символами. Так, если требуется преобразовать что-нибудь длиннее одного символа (например, все экземпляры слова unix в слово UNIX), для этой цели придется воспользоваться другой командой, например sed.

Таблица 3.4. Характерные примеры применения команды tr

Команда tr	Описание
`tr 'X' 'x'`	Преобразовать все прописные буквы **X** в строчные буквы **x**
`tr '()' '{}'`	Преобразовать все открывающие и закрывающие круглые скобки в фигурные
`tr '[a-z]' '[A-Z]'`	Преобразовать все строчные буквы в прописные
`tr '[A-Z]' '[N-ZA-M]'`	Преобразовать прописные буквы **A–M** в **N–Z**, а буквы **N–Z** — в буквы **A–M** соответственно
`tr '  ' '  '`	Преобразовать в пробелы все знаки табуляции, указанные в первой паре кавычек
`tr -s '  ' '  '`	Преобразовать многие пробелы в один пробел
`tr -d '\14'`	Удалить все знаки перевода страницы, обозначенные значением **14**, в восьмеричном представлении
`tr -d '[0-9]'`	Удалить все цифры

Команда grep

Команда grep дает возможность находить в одном или нескольких файлах содержимое, совпадающее с указанным шаблоном. Общая форма команды grep выглядит следующим образом:

```
grep шаблон файлы
```

Всякая строка, содержащая указанный *шаблон*, выводится на терминал. Если в команде указано несколько *файлов*, каждая найденная в них строка предваряется именем соответствующего файла, чтобы выяснить файл, в котором было обнаружено совпадение с указанным шаблоном.

Допустим, в файле ed.cmd требуется найти все вхождения слова shell. Как следует из результата выполнения приведенной ниже команды grep, в файле ed.cmd обнаружены две строки, содержащие слово shell.

```
$ grep shell ed.cmd
files, and is independent of the shell.
to the shell, just type in a q.
$
```

Если шаблон не существует в указанном файле или нескольких файлах, команда grep ничего и выведет на терминал:

```
$ grep cracker ed.cmd
$
```

Как было показано ранее при рассмотрении редактора sed, для вывода из файла intro всех строк, содержащих слово UNIX, достаточно было выполнить следующую команду:

```
sed -n '/UNIX/p' intro
```

Но того же самого результата можно добиться и с помощью приведенной ниже команды grep.

```
grep UNIX intro
```

Вернемся снова к упоминавшемуся ранее файлу phonebook, который содержит следующее:

```
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Jeff Goldberg       201-555-3378
Liz Stachiw         212-555-2298
Susan Goldberg      201-555-7776
Tony Iannino        973-555-1295
$
```

Если в данном файле требуется найти конкретный номер телефона, для этой цели как нельзя лучше подойдет следующая команда grep:

```
$ grep Susan phonebook
Susan Goldberg      201-555-7776
$
```

Команда grep особенно удобна в том случае, если имеется много файлов и требуется выяснить, в каком из них содержатся определенные слова или фразы. В приведенном ниже примере демонстрируется применение команды grep для поиска слова shell во *всех* файлах текущего каталога. Как отмечалось выше, если в команде grep указано несколько файлов, каждая выводимая строка предваряется именем файла, в котором содержится эта строка.

```
$ grep shell *
cmdfiles:shell that enables sophisticated
ed.cmd:files, and is independent of the shell.
ed.cmd:to the shell, just type in a q.
grep.cmd:occurrence of the word shell:
grep.cmd:$ grep shell *
grep.cmd:every use of the word shell.
$
```

Подобно выражениям, применяемым в редакторе sed, а также шаблонам, указываемым в команде tr, шаблоны, задаваемые в команде grep, рекомендуется заключать в *одиночные* кавычки, чтобы "защитить" их от интерпретации в оболочке. Рассмотрим в качестве примера, что произойдет, если этого не сделать. Допустим, требуется найти все строки, содержащие знак звездочки в файле stars. Если ввести команду

```
grep * stars
```

то достичь желаемого результата не удастся, поскольку, обнаружив знак звездочки, оболочка автоматически подставит вместо него имена всех файлов в текущем каталоге:

```
$ ls
circles
polka.dots
squares
stars
stripes
$ grep * stars
$
```

В данном случае оболочка сначала подставила вместо знака звездочки список файлов из текущего каталога. А затем запустила на выполнение команду grep, которая приняла в качестве первого аргумента файл circles и безуспешно попыталась найти его имя в файлах, указанных в качестве оставшихся аргументов (рис. 3.1).

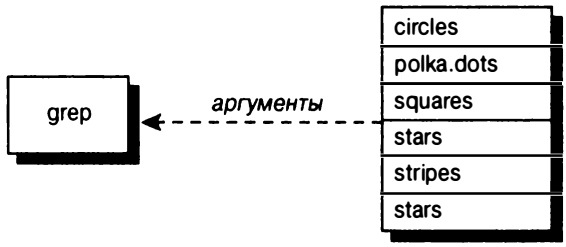


Рис. 3.1. Процесс выполнения команды **grep * stars**

Но если заключить знак звездочки в кавычки, он тем самым будет защищен от синтаксического анализа и интерпретации в оболочке. Ниже показано, как это сделать в рассматриваемом здесь примере.

```
$ grep '*' stars
The asterisk (*) is a special character that
**********
5 * 4 = 20
$
```

Кавычки сообщают оболочке оставить без внимания заключенные в них символы. В таком случае оболочка запустит команду `grep` на выполнение, передав ей два аргумента: знак * (*без* кавычек, которые попутно удаляются оболочкой), а также имя файла `stars` (рис. 3.2).

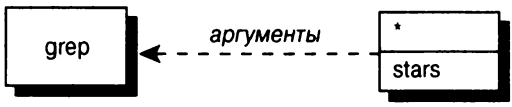


Рис. 3.2. Процесс выполнения команды **`grep '*' stars`**

Помимо знака *, существуют и другие символы, имеющие специальное назначение в оболочке, а следовательно, они должны быть заключены в кавычки, когда они употребляются в шаблоне. Следует, однако, признать, что вопросы обработки кавычек в оболочке непросты, и поэтому им посвящена отдельная глава 5.

Команда `grep` принимает свои исходные данные из стандартного ввода, если в ней не указано имя файла. Это означает, что команду `grep` можно применять в конвейере для просмотра строк, совпадающих с указанным шаблоном в результате, выводимом предыдущей командой. Допустим, требуется выяснить, вошел ли пользователь `jim` в систему. С этой целью можно просмотреть по команде `grep` результат выполнения команды `who`, организовав конвейер следующим образом:

```
$ who | grep jim
jim tty16 Feb 20 10:25
$
```

Однако, если файл для поиска не указан, команда `grep` автоматически просмотрит стандартный ввод. Естественно, что если пользователь `jim` не вошел в систему, то вместо приведенного выше результата в командной строке будет просто выдано приглашение:

```
$ who | grep jim
$
```

Регулярные выражения и команда `grep`

Вернем снова к тестовому файлу `intro`, отобразив его содержимое:

```
$ cat intro
The UNIX operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s.  One of the primary goals in
the design of the UNIX system was to create an
environment that promoted efficient program
development.
$
```

Команда grep дает возможность указать шаблон, используя регулярные выражения таким же образом, как и в строковом редакторе ed. Это означает, что в команде grep можно указать следующий шаблон:

```
[tT]he
```

чтобы найти строчную или прописную букву t или T, а вслед за ней — буквы he.

Чтобы перечислить все строки из файла intro, содержащие комбинацию символов the или The, можно ввести следующую команду grep:

```
$ grep '[tT]he' intro
The UNIX operating system was pioneered by Ken
in the late 1960s.  One of the primary goals in
the design of the UNIX system was to create an
$
```

Более изящный способ состоит в том, чтобы употребить в команде grep параметр **-i**, позволяющий составлять шаблоны без учета регистра. Так, по команде

```
grep -i 'the' intro
```

поиск строк в файле intro на совпадение с указанным шаблоном осуществляется по команде grep без учета регистра. Следовательно, на терминал будут выведены строки, содержащие комбинацию символов the или The, а также строки, содержащие комбинации символов THE, THe, tHE и т.д.

В табл. 3.5 приведены другие примеры регулярных выражений, которые могут быть указаны в команде grep, а также типы шаблонов, с которыми они совпадают.

Таблица 3.5. Некоторые примеры применения команды grep

Команда grep	Описание
`grep '[A-Z]' list`	Строки из файла **list**, содержащие прописную букву английского алфавита
`grep '[0-9]' data`	Строки из файла **data**, содержащие цифру
`grep '[A-Z]...[0-9]' list`	Строки из файла **list**, содержащие шаблоны из пяти символов, начинающиеся с прописной буквы и оканчивающиеся цифрой
`grep '\.pic$' filelist`	Строки из файла **filelist**, оканчивающиеся комбинацией символов **.pic**

Параметр -v

Иногда требуется найти строки, которые *не* содержат указанный шаблон. Именно для этой цели и служит параметр **-v** команды grep. В частности, он меняет на *обратную* логику совпадения с шаблоном. В следующем примере команда

grep применяется с параметром **-v** для поиска в файле intro всех строк, которые *не* содержат слово UNIX:

```
$ grep -v 'UNIX' intro                    Вывести все строки,
                                          не содержащие слово UNIX
Thompson and Dennis Ritchie at Bell Laboratories
in the late 19605.  One of the primary goals in
environment that promoted efficient program
development.
$
```

Параметр -l

Порой требуется выявить *не* строки, совпадающие с шаблоном, а только имена файлов, содержащих этот шаблон. Допустим, в текущем каталоге находится ряд программ на языке C (имена исходных файлов программ на C принято дополнять расширением **.c**) и требуется выяснить, в каком из них применяется переменная Move_history. Решить эту задачу можно следующим образом:

```
$ grep 'Move_history' *.c                 Искать переменную Move_history
                                          во всех исходных файлах на C
exec.c:MOVE Move_history[200] = {0};
exec.c:     cpymove(&Move_history[Number_half_moves -1],
exec.c: undo_move(&Move_history[Number_half_moves-1],;
exec.c: cpymove(&last_move,&Move_history[Number_half_moves-1]);
exec.c:     convert_move(&Move_history[Number_half_moves-1]),
exec.c: convert_move(&Move_history[i-1]),
exec.c: convert_move(&Move_history[Number_half_moves-1]),
makemove.c:IMPORT MOVE Move_history[];
makemove.c:     if ( Move_history[j].from != BOOK (i,j,from) OR
makemove.c:          Move_history[j] .to != BOOK (i,j,to) )
testch.c:GLOBAL MOVE Move_history[100] = {0};
testch.c:     Move_history[Number_half_moves-1].from = move.from;
testch.c:     Move_history[Number_half_moves-1].to = move.to;
$
```

Просеяв приведенный выше результат, можно обнаружить три исходных файла, exec.c, makemove.c и testch.c, где применяется искомая переменная. Если же дополнить команду grep параметром **-l**, то будет получен список файлов, содержащих указанный шаблон, а не те строки, которые с ним совпадают:

```
$ grep -l 'Move_history' *.c              Перечислить файлы, содержащие
                                          переменную Move_history
exec.c
makemove.c
testch.c
$
```

Найденные файлы удобно перечисляются по команде grep в отдельных строках, и поэтому результат выполнения команды grep -l можно передать по каналу

следующей далее команде wc, чтобы подсчитать *количество* файлов, содержащих указанный шаблон, как демонстрируется в следующем примере:

```
$ grep -l 'Move_history' *.c | wc -l
      3
$
```

Как следует из приведенного выше результата, ссылки на переменную Move_history содержатся в трех исходных файлах. А теперь проверьте себя: что вам удастся подсчитать, если вы введете команду grep без параметра **-l** и передадите результат ее выполнения по каналу следующей далее команде wc -l?

Параметр -n

Если в команде указан параметр -n, каждая строка из заданного файла, совпадающая с заданным шаблоном, предваряется соответствующим номером. Как следует из предыдущих примеров, исходный файл testch.c оказался одним из тех файлов, где присутствовали ссылки на переменную Move_history. А в следующем примере демонстрируется, как выявить в этом исходном файле конкретные строки кода, содержащие ссылку на переменную Move_history:

```
$ grep -n 'Move_history' testch.c          Предварить строки кода
                                           их номерами
13:   GLOBAL MOVE Move_history[100] = {0};
197:  Move_history[Number_half_moves-1].from = move.from;
198:  Move_history[Number_half_moves-1].to = move.to;
$
```

Как видите, переменная Move_history применяется в строках кода **13**, **197** и **198** из исходного файла testch.c. Команда grep является одной из наиболее употребительных опытными пользователями системы Unix благодаря своей гибкой и развитой логике сопоставления с шаблоном. Поэтому она вполне заслуживает более углубленного изучения.

Команда sort

В простейшем виде команду sort очень легко понять. Достаточно предоставить ей строки исходных данных, и она отсортирует их в алфавитном порядке, выведя результат на терминал, как показано ниже. По умолчанию команда sort принимает каждую строку из указанного входного файла и сортирует ее по возрастанию.

```
$ sort names
Charlie
Emanuel
Fred
Lucy
```

```
Ralph
Tony
Tony
$
```

Специальные символы сортируются в соответствии с внутренней кодировкой символов. Например, символ пробела внутренне представлен числовым значением **32**, а знак двойной кавычки — числовым значением **34**. Это означает, что в результате сортировки пробел будет расположен прежде двойной кавычки. Порядок сортировки для других языковых стандартов может быть иным, и поэтому порядок следования букв, знаков препинания и других специальных символов на иностранных языках не всегда оказывается таким, как можно было бы ожидать. Но, как правило, можно быть уверенным, что команда `sort` обработает буквенно-цифровые исходные данные должным образом.

У команды `sort` имеется немало параметров, предоставляющих дополнительные удобства для сортировки. Ниже будут описаны некоторые из них.

Параметр `-u`

Параметр **`-u`** предписывает команде `sort` исключить дубликаты строк из выводимого результата, как показано в следующем примере:

```
$ sort -u names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
$
```

В данном примере дубликат строки с именем `Tony` исключен из результата, выводимого командой `sort`. По традиции многие пользователи системы Unix выполняли ту же самую операцию, используя отдельную программу `uniq`, и если просмотреть системные сценарии оболочки, то в них нередко можно обнаружить команды вроде `sort | uniq`. Теперь их можно заменить командой `sort -u`!

Параметр `-r`

Изменить порядок сортировки на обратный можно, указав параметр **`-r`** следующим образом:

```
$ sort -r names                    Изменить порядок сортировки
Tony
Tony
Ralph
Lucy
Fred
```

```
Emanuel
Charlie
$
```

Параметр -o

По умолчанию команда `sort` направляет отсортированные данные в стандартный вывод. Чтобы направить их в файл, можно, с одной стороны, указать переадресацию вывода следующим образом:

```
$ sort names > sorted_names
$
```

А с другой стороны, можно воспользоваться параметром **-o**, указав сразу же после него выходный файл, как показано ниже. В итоге отсортированное содержимое файла `names` будет выведено в файл `sorted_names`.

```
$ sort names -o sorted_names
$
```

В чем же ценность параметра **-o**? Зачастую требуется отсортировать строки в файле, заменив в нем первоначальные данные отсортированными. Но если ввести следующую команду:

```
$ sort names > names
$
```

то в конечном итоге содержимое файла `names` будет очищено! А с помощью параметра **-o** вполне допустимо указывать одно и то же имя как входного, как и выходного файла в команде `sort`, как демонстрируется в следующем примере:

```
$ sort names -o names
$ cat names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
Tony
$
```

Совет

Будьте внимательны, заменяя первоначальный входной файл при фильтрации, сортировке или иной обработке его содержимого. Непременно убедитесь в правильности выполнения конкретной операции, прежде чем перезаписывать исходные данные. Система Unix предоставляет немало удобных средств, но в ней отсутствует команда *отмены удаления*, чтобы восстановить потерянные данные или файлы.

Параметр -n

Допустим, имеется файл, содержащий пары координат точек данных (*x*, *y*):

```
$ cat data
5 27
2 12
3 33
23 2
-5 11
15 6
14 -9
$
```

Допустим также, что эти данные требуется предоставить программе построения графиков под названием `plotdata`. Но этой программе требуется, чтобы исходные пары данных были отсортированы по возрастанию значения координаты *x* (т.е. первого значения в каждой строке).

Параметр **-n** команды `sort` обозначает, что первое поле в строке следует считать *числом*, а данные — отсортировать в алфавитном порядке. Сравните результат выполнения команды `sort` без параметра **-n** и вместе с ним в следующем примере:

```
$ sort data
-5    11
14    -9
15    6
2     12
23    2
3     33
5     27
$ sort -n data          Отсортировать в арифметическом порядке
-5    11
2     12
3     33
5     27
14    -9
15    6
23    2
$
```

Пропуск полей

Если попытаться отсортировать файл `data` по значению координаты *y*, т.е. по второму числу в каждой строке, то команде `sort` можно было бы предписать, чтобы она начинала сортировку со второго поля, указав следующий параметр:

```
-k2n
```

вместо параметра **-n**. Параметр **-k2** предписывает пропустить первое поле и начать анализ сортировки со второго поля в каждом поле. Аналогично параметр

-k5n означает следующее: начать сортировку с пятого поля в каждой строке, а затем отсортировать данные в числовом порядке, как показано ниже.

```
$ sort -k2n data          Начать сортировку со второго поля
14    -9
23    2
15    6
-5    11
2     12
5     27
3     33
$
```

По умолчанию поля разделяются символами пробела или табуляции. Если же требуется указать другой разделитель, это можно сделать с помощью параметра **-t**.

Параметр -t

Как упоминалось выше, если пропустить некоторые поля, команда `sort` предположит, что сортируемые поля разделяются символами пробела или табуляции. Параметр `-t` позволяет указать иное. В этом случае символ, указываемый после параметра `-t`, принимается в качестве разделителя.

Рассмотрим снова в качестве примера файл пароля со следующим содержимым:

```
$ cat /etc/passwd
root:*:0:0:The super User:/:/usr/bin/ksh
steve:*:203:100::/users/steve:/usr/bin/ksh
bin:*:3:3:The owner of system files:/:
cron:*:1:1:Cron Daemon for periodic tasks:/:
george:*:75:75::/users/george:/usr/lib/rsh
pat:*:300:300::/users/pat:/usr/bin/ksh
uucp:nc823ciSiLiZM:5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
asg:*:6:6:The Owner of Assignable Devices:/:
sysinfo:*:10:10:Access to System Information:/:/usr/bin/sh
mail:*:301:301::/usr/mail:
$
```

Если этот файл требуется отсортировать по имени пользователя (т.е. первому полю в каждой строке), с этой целью можно выдать следующую команду:

```
sort /etc/passwd
```

Чтобы отсортировать этот файл по третьему полю, разделяемому двоеточием и содержащему так называемый *идентификатор пользователя*, следует указать арифметический порядок сортировки (параметр **-k3**) и знак двоеточия в качестве разделителя полей (параметр **-t:**). В следующем примере третье поле специально выделено полужирным в каждой строке, чтобы легко проверить, что файл правильно отсортирован по идентификатору пользователя:

```
$ sort -k3n -t: /etc/passwd          Отсортировать по
                                     идентификатору пользователя
root:*:0:0:The Super User:/:/usr/bin/ksh
cron:*:1:1:Cron Daemon for periodic tasks:/:
bin:*:3:3:The owner of system files:/:
uucp:*:5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
asg:*:6:6:The Owner of Assignable Devices:/:
sysinfo:*:10:10:Access to System Information:/:/usr/bin/sh
george:*:75:75::/users/george:/usr/lib/rsh
steve:*:203:100::/users/steve:/usr/bin/ksh
pat:*:300:300::/users/pat:/usr/bin/ksh
mail:*:301:301::/usr/mail:
$
```

Другие параметры

Имеются и другие параметры команды `sort`, позволяющие пропускать символы в поле, указывать поле, на котором должна *завершаться* сортировка, объединять отсортированные входные файлы и выполнять сортировку в словарном (или лексикографическом) порядке, где для сравнения применяются буквы, числа и пробелы. Подробнее об этих параметрах можно узнать из документации на команду `sort` в *Руководстве пользователя системы Unix*.

Команда `uniq`

Командой `uniq` удобно пользоваться в тех случаях, когда в файле требуется найти или удалить дубликаты строк. Ниже приведена общая форма команды `uniq`.

```
uniq входной_файл выходной_файл
```

В соответствии с этим форматом команда `uniq` копирует *входной_файл* в *выходной_файл*, удаляя в ходе этого процесса любые дубликаты строк. Согласно определению команды `uniq`, дубликатами строк считаются *следующие подряд* и *точно совпадающие* строки.

Если же *выходной_файл* не указан, результаты выполнения команды `uniq` будут направлены в стандартный вывод. А если не указан и *входной_файл*, то команда `uniq` действует в качестве фильтра, читая свои исходные данные из стандартного ввода.

Рассмотрим некоторые примеры применения команды `uniq`. Допустим, имеет файл `names` со следующим содержимым:

```
$ cat names
Charlie
Tony
Emanuel
Lucy
Ralph
```

```
Fred
Tony
$
```

Как видите, имя Tony появляется в файле names дважды. С помощью команды uniq такие дубликаты записей в файле могут быть удалены следующим образом:

```
$ uniq names                    Вывести однозначные строки
Charlie
Tony
Emanuel
Lucy
Ralph
Fred
Tony
$
```

Но позвольте, имя Tony так и осталось в приведенном выше результате, поскольку несколько повторяющихся экземпляров записей следуют в файле *не подряд*. Следовательно, определение дубликатов строк в команде uniq оказывается неудовлетворительным. В качестве выхода из этого положения зачастую применяется команда sort, чтобы получить расположенные рядом дубликаты строк, как упоминалось ранее в этой главе. После этого результат сортировки передается на обработку команде uniq, как показано ниже.

```
$ sort names | uniq
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
$
```

В данном примере команда sort перемещает строки с именем Tony, располагая их рядом, а команда uniq отсеивает их дубликаты. Напомним, однако, что то же самое можно сделать и по команде sort с параметром **-u**.

Параметр -d

Зачастую требуется найти только дубликаты записей в файле. С этой целью можно указать параметр **-d**, предписывающий команде uniq вывести *только* дублирующиеся строки в указанный *выходной_файл* (или направить их в стандартный вывод). Такие строки выводятся только один раз независимо от того, сколько раз они повторяются, как демонстрируется в следующем примере:

```
$ sort names | uniq -d          Перечислить дублирующиеся строки
Tony
$
```

Для рассмотрения более практического примера вернемся снова к упоминавшемуся ранее файлу /etc/passwd. Этот файл содержит сведения о каждом пользователе, зарегистрированном в системе. Вполне вероятно, что в процессе ввода и удаления сведений о пользователях из этого файла одно и то же имя пользователя могло быть неумышленно введено не один раз. Чтобы найти подобные дублирующиеся записи, можно сначала отсортировать содержимое файла /etc/passwd, а затем передать полученные результаты по каналу следующей далее команде uniq -d, как это демонстрировалось в предыдущем примере и показано ниже.

```
$ sort /etc/passwd | uniq -d          Найти дубликаты записей
                                      в файле /etc/passwd
$
```

Но ведь полностью дублирующиеся строки в файле /etc/passwd отсутствуют. А для того чтобы найти дублирующиеся записи в полях с именами пользователей, нужно просмотреть только первое поле в каждой строке (напомним, что поле с именем пользователя составляют начальные символы вплоть до первого знака двоеточия в каждой строке из файла /etc/passwd). Одной команды uniq с параметром **-d** для этой цели недостаточно, но ее можно все же достичь, если сначала извлечь по команде cut имя пользователя из каждой строки в файле /etc/passwd, а затем направить его по каналу команде uniq, как показано в следующем примере:

```
$ sort /etc/passwd | cut -f1 -d: | uniq -d        Найти дубликаты
cem
harry
$
```

Как следует из приведенного выше результата, в файле /etc/passwd имеется несколько записей с именами пользователей cem и harry. Если же требуется получить дополнительные сведения о конкретных записях, их можно извлечь по команде grep из файла /etc/passwd следующим образом:

```
$ grep -n 'cem' /etc/passwd
20:cem:*:91:91::/users/cem:
166:cem:*:91:91::/users/cem:
$ grep -n 'harry' /etc/passwd
29:harry:*:103:103:Harry Johnson:/users/harry:
79:harry:*:90:90:Harry Johnson:/users/harry:
$
```

Параметр **-n** использован в данном примере с целью выявить дублирующиеся записи. Так, в строках **20** и **166** обнаружены две записи с именем пользователя cem и еще две записи с именем пользователя harry в строках **29** и **79**.

Другие параметры

Указав в команде uniq параметр **-c**, можно подсчитать количество вхождений данных (строк или записей), что может быть очень удобно в сценариях. В следующем примере демонстрируется применение параметра **-c** в команде uniq:

```
$ sort names | uniq -c          Подсчитать вхождения строк
    1 Charlie
    1 Emanuel
    1 Fred
    1 Lucy
    1 Ralph
    2 Tony
$
```

Команда uniq -c нередко применяется с целью выявить слова, чаще всего встречающиеся в файле данных. Это нетрудно сделать, введя следующую команду:

```
tr '[A-Z]' '[a-z]' datafile | sort | uniq -c | head
```

У команды uniq имеются еще два параметра, но для их описания здесь недостаточно места. Упомянем лишь, что с их помощью можно удалить начальные символы или поля в строке. Более подробно ознакомиться с этими параметрами можно, обратившись за справкой к соответствующей оперативной странице руководства по системе Unix, введя команду man uniq.

Было бы опрометчиво не упомянуть здесь команды awk и perl, которыми удобно пользоваться для написания программ в оболочке. Но обе эти команды сами по себе являются довольно крупными и сложными средами программирования. Поэтому для изучения команды awk рекомендуется руководство *Awk—A Pattern Scanning and Processing Language*, Aho et al., а ее описание можно найти во втором томе *Руководства по программированию в системе Unix* (Unix Programmer's Manual, Volume II). Неплохой справочный и учебный материал по языку Perl можно найти в книгах *Learning Perl* и *Programming Perl* (издательство O'Reilly and Associates; соответствующие издания обеих книг вышли в русском переводе).

4

Итак, приступим!

Изучив материал главы 2, вы должны теперь понимать, что, вводя всякий раз команду вроде следующей:

```
who | wc -l
```

вы фактически программируете в оболочке. В данном случае оболочка интерпретирует командную строку, распознавая знак канала, соединяя выход первой команды со входом второй и начиная выполнение обеих указанных команд. Из этой главы вы узнаете, как писать собственные команды и пользоваться переменными оболочки.

Командные файлы

Программа, выполняемая в оболочке, может быть введена непосредственно, как в следующем примере:

```
$ who | wc -l
```

или же набрана сначала в файле, а затем выполнена из этого файла в оболочке. Допустим, требуется выяснить количество пользователей, неоднократно входивших в систему в течение дня. С этой целью было бы неблагоразумно вводить приведенную выше конвейерную операцию всякий раз, когда требуется подобная информация. Но ради примера наберем ее в отдельном файле nu, имя которого сокращенно означает "количество пользователей".

Если обратиться к файлу nu, то окажется, что он содержит упомянутую выше конвейерную операцию:

```
$ cat nu
who | wc -l
$
```

А для того чтобы выполнить команды из файла nu, достаточно ввести его имя в качестве имени команды в оболочке следующим образом:

```
$ nu
sh: nu: cannot execute
(sh: nu: нельзя выполнить)
$
```

Введенный выше файл не удалось выполнить, потому что мы забыли упомянуть о следующем: прежде чем выполнять сценарий из командной строки, необходимо изменить права доступа к его файлу таким образом, чтобы сделать его *исполняемым*. И для этой цели служит команда chmod. В частности, чтобы добавить право на исполнение файла nu, достаточно ввести следующую команду:

```
chmod +x файл(ы)
```

где **+x** обозначает, что указанные *файл(ы)* требуется сделать исполняемыми. Оболочка требует, чтобы указанные *файл(ы)* были доступны для чтения и исполнения, прежде чем вызывать их непосредственно из командной строки, как показано ниже.

```
$ ls -l nu
-rw-rw-r-- 1 steve steve 12 Jul 10 11:42 nu
$ chmod +x nu                Сделать файл исполняемым
$ ls -l nu
-rwxrwxr-x 1 steve steve 12 Jul 10 11:42 nu
$
```

Итак, сделав файл nu исполняемым, попробуем выполнить его снова:

```
$ nu
      8
$
```

На этот раз сработало. Конвейерная операция из файла nu выполнена успешно.

Предупреждение

Если после разрешения вопросов, связанных с правами доступа, вы все равно получите сообщение об ошибке **"Command not found"** (Команда не найдена), попробуйте ввести знаки **./** перед именем файла (например, **./nu**), чтобы оболочка искала этот командный файл как в текущем каталоге команд, так и в обычных местах системы. А для того чтобы исправить положение в долгосрочной перспективе, можете ввести знак . в конце переменной окружения **PATH**, которая обычно находится в вашем профильном файле с расширением **.profile**.

В файле можно разместить любые команды, сделать его исполняемым, а затем выполнить его содержимое, введя имя этого файла в командной строке оболочки. Этот очень простой и в то же время эффективный способ работы в режиме командной строки пригоден и для написания сценариев оболочки.

Стандартные механизмы оболочки вроде переадресации ввода-вывода и каналов можно применять и в собственных программах, как демонстрируется в следующем примере:

```
$ nu > tally
$ cat tally
      8
$
```

Допустим, вы работаете над предложением в файле sys.caps и для вывода этого предложения всякий раз требуется следующая последовательность команд:

```
tbl sys.caps | nroff -mm -Tlp | lp
```

Чтобы сэкономить время и труд на наборе этой последовательности команд, ее можно разместить в командном файле (например, под именем run), сделав его исполняемым, а затем вводить его имя run в командной строке всякий раз, когда потребуется вывести новую копию предложения, как показано в приведенном ниже примере, где сообщение об идентификаторе запроса выводится по команде lp.

```
$ cat run
tbl sys.caps | nroff -mm -Tlp | lp
$ chmod +x run
$ run
request id is laser1-15 (standard input)
$
```

В качестве еще одного примера допустим, что требуется написать программу оболочки под названием stats, выводящую дату и время, количество пользователей, зарегистрированных в системе, а также содержимое текущего рабочего каталога. Для получения этих сведений потребуется следующая последовательность из трех команд date, who | wc -l и pwd:

```
$ cat stats
date
who | wc -l
pwd
$ chmod +x stats
$ stats                    Опробовать командный файл
Wed Jul 10 11:55:50 EDT 2002
    13
/users/steve/documents/proposals
$
```

В командный файл stats можно также ввести ряд команд echo, чтобы сделать выводимый результат более информативным, как показано ниже.

```
$ cat stats
echo The current date and time is:
date
echo
echo The number of users on the system is:
who | wc -l
echo
echo Your current working directory is:
pwd
$ stats          Выполнить командный файл
```

```
The current date and time is:
Wed Jul 10 12:00:27 EDT 2002
The number of users on the system is:

    13

Your current working directory is:
/users/steve/documents/proposals
$
```

Напомним, что команда `echo` без аргументов выдает пустую строку. Далее будет показано, как выводить сообщение и результат выполнения команды в одной строке, как в следующем примере:

```
The current date and time is: Wed Jul 10 12:00:27 EDT 2002
```

Комментарии

Программирование на языке оболочки было бы неполноценным без оператора *комментариев*. Комментарий — это способ ввести в исходном тексте программы примечания или заметки, не оказывающие никакого влияния на ее выполнения, а только поясняющие ее назначение.

Всякий раз, когда оболочка обнаруживает специальный символ **#**, она игнорирует все, что следует после этого символа до конца строки. Если строка начинается с символа **#**, она полностью интерпретируется как комментарий. Ниже приведены примеры достоверных комментариев.

```
#  Этот комментарий занимает всю строку
who | wc -l       # подсчитать количество пользователей
#
#  Проверить, правильно ли были предоставлены аргументы
#
```

Комментарии удобны для документирования отдельных команд или их последовательностей, назначение которых может оказаться не вполне очевидным или настолько сложным, что можно легко забыть, для чего они служат и что вообще делают. Благоразумное употребление комментариев упрощает также отладку и сопровождение программ оболочки как их авторам, так и тем, кому придется поддерживать эти программы впоследствии.

Вернемся к рассмотренному выше командному файлу `stats` и введем в него ряд комментариев и пустых строк для повышения удобочитаемости содержащейся в нем программы:

```
$ cat stats
#
# Программа stats -- выводит: дату, количество пользователей,
#                    зарегистрированных в системе, а также
#                    текущий рабочий каталог
```

```
echo The current date and time is:
date

echo
echo The number of users on the system is:
who | wc -l

echo
echo Your current working directory is:
pwd
$
```

Лишние пустые строки не оказывают никакого влияния на выполнение программы, но повышают ее удобочитаемость. Оболочка их просто игнорирует.

Переменные

Подобно буквально всем языкам программирования, оболочка допускает хранение значений в *переменных*. Имя переменной оболочки должно начинаться с буквы или знака подчеркивания (_), после чего следует нулевое или большее количество буквенно-цифровых символов или знаков подчеркивания.

Примечание

Таким образом, регулярное выражение для обнаружения переменной оболочки должно выглядеть следующим образом:

```
[A-Za-z_] [a-zA-Z0-9_]*
```

Чтобы сохранить значение в переменной оболочки, достаточно указать имя этой переменной, знак равенства (=) и присваиваемое ей значение, как демонстрирует следующая общая форма:

переменная=значение

Например, чтобы присвоить значение **1** переменной оболочки `count`, достаточно написать следующую строку кода:

```
count=1
```

а для того чтобы присвоить значение `/users/steve/bin` переменной оболочки `my_bin`, такую строку кода:

```
my_bin=/users/steve/bin
```

Прежде всего обратите внимание на то, что пробелы по обе стороны от знака равенства *не* допускаются. Это обстоятельство необходимо иметь в виду особенно тем, кто привык указывать операции присваивания с пробелами в других языках программирования. А в языке оболочки ничего подобно *не* разрешается.

Кроме того, в языке оболочки, в отличие от других языков программирования, отсутствует понятие *типов данных*. Всякий раз, когда значение присваивается переменной оболочки, каким бы оно ни было, оболочка просто интерпретирует это значение как символьную строку. Так, если присвоить значение **1** переменной `count`, оболочка просто сохранит *символ* **1** в переменной `count`, допустив, что сохраненное значение является целочисленным.

Если у вас имеется опыт программирования на таких языках, как Perl, Swift или Ruby, где переменные должны непременно *объявляться*, имейте в виду, что в оболочке понятие типов данных отсутствует, и поэтому переменные в ней *не* объявляются перед их употреблением. Чтобы воспользоваться переменной оболочки, достаточно присвоить ей требующееся значение.

В оболочке поддерживаются целочисленные операции над ее переменными, содержащими символьные строки, которые преобразуются в достоверные числа с помощью встроенных специальных операций. Но даже тогда происходит постоянное вычисление числового значения переменной с целью гарантировать его достоверность.

Язык оболочки является интерпретирующим, и поэтому значения переменным можно присваивать непосредственно в командной строке, как показано ниже.

```
$ count=1                        Присвоить символ 1 переменной count
$ my_bin=/users/steve/bin        Присвоить символы /users/steve/bin
                                 переменной my_bin
$
```

Итак, показав, каким образом значения присваиваются переменным, выясним, какую из этого можно извлечь пользу. А ведь она действительно существует, как поясняется далее.

Отображение значений переменных

Команда `echo`, применявшаяся в предыдущих примерах для вывода значений, в том числе и символьных строк, получаемых из стандартного ввода, может также служить для отображения значений, хранящихся в переменных оболочки. Для этого достаточно ввести команду `echo` в следующем формате:

```
echo $переменная
```

Знак **$** принимает в оболочке специальное назначение, если после него следует один или несколько буквенно-цифровых символов. Так, если после знака **$** следует имя переменной, оболочка воспринимает это как указание на то, что в переменной хранится значение, которое должно быть подставлено вместо ее имени. Так, если ввести команду `echo $count`, оболочка заменит указанное имя переменной `$count` хранящимся в ней значением и затем выполнит команду `echo`, как показано ниже.

```
$ echo $count
1
$
```

Напомним, что оболочка выполняет подстановку значений переменных *перед* выполнением команд, как показано на рис. 4.1.

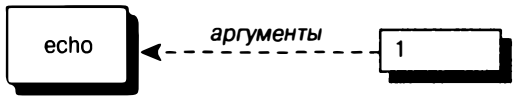


Рис. 4.1. Процесс выполнения команды **echo $count**

Одновременно можно подставить несколько значений переменных, как демонстрируется в следующем примере:

```
$ echo $my_bin
/users/steve/bin
$ echo $my_bin $count
/users/steve/bin 1
$
```

В данном примере оболочка подставляет значения переменных `my_bin` и `count`, а затем выполняет команду `echo`, как показано на рис. 4.2.

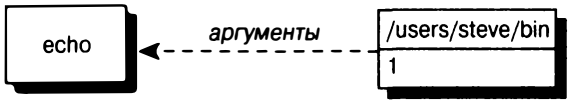


Рис. 4.2. Процесс выполнения команды **echo $my_bin $count**

Переменные можно указывать в любом месте командной строки. Они заменяются своими значениями в оболочке перед выполнением конкретной команды, как демонстрируется в следующих примерах:

```
$ ls $my_bin
mon
nu
testx
$ pwd                                    Отобразить текущий каталог
/users/steve/documents/memos
$ cd $my_bin                             Перейти в каталог bin
$ pwd
/users/steve/bin
$ number=99
$ echo There are $number bottles of beer on the wall
There are 99 bottles of beer on the wall
$
```

Ниже приведен ряд других примеров применения переменных оболочки.

```
$ command=sort
$ $command names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
Tony
$ command=wc
$ option=-l
$ file=names
$ $command $option $file
    7 names
$
```

Как видите, в переменной можно хранить даже имя команды. Оболочка подставит его, прежде чем определит имя исполняемой команды и ее аргументы, и с этой целью она произведет синтаксический анализ следующей командной строки:

```
$command $option $file
```

а затем она сделает все запрашиваемые подстановки, сформировав в итоге приведенную ниже команду. Таким образом, оболочка вызовет команду wc, передав ей аргументы **-l** и **names**.

```
wc -l names
```

Имеется даже возможность присваивать значения одних переменных другим переменным, как показано в приведенном ниже примере. Напомним, что перед именем переменной следует указывать знак денежной единицы (**$**) всякий раз, когда требуется воспользоваться хранящимся в ней значением.

```
$ value1=10
$ value2=value1
$ echo $value2
value1                          Так нельзя!
$ value2=$value1
$ echo $value2
10                              Так-то лучше!
$
```

Неопределенные переменные имеют пустое значение

Как вы думаете, что произойдет, если попытаться отобразить содержимое переменной, которой еще не присвоено значение? Попробуйте ввести приведенную ниже команду, чтобы посмотреть, к чему это приведет.

```
$ echo $nosuch          Этой переменной вообще не присвоено значение
$
```

В итоге вы не получите никакого сообщения об ошибке. А отобразила ли команда `echo` хотя бы что-нибудь? Если попытаться точнее определить переменную `nosuch`, заключив ее в двоеточия, то окажется, что оболочка не подставит вместо нее *никакого* значения:

```
$ echo :$nosuch:          Заключить значение переменной в двоеточия
::
$
```

Переменная, не содержащая никакого значения, считается неопределенной и фактически содержит *пустое* значение. Такое состояние принимают переменные, которым вообще не присвоено никакого значения. Когда же оболочка выполняет подстановку значений переменных, любые значения, которые оказываются пустыми, по существу, удаляются из командной строки, что вполне благоразумно, как демонстрируется в следующем примере:

```
$ wc $nosuch -l $nosuch $nosuch names
      7 names
$
```

В данном примере оболочка просматривает командную строку, подставляя пустое значение вместо переменной `nosuch`. По окончании просмотра командная строка примет приведенный ниже вид, наглядно поясняющий, каким образом пустые значения переменных обрабатываются оболочкой.

```
wc -l names
```

Впрочем, иногда возникает потребность инициализировать переменную пустым значением. Для этого достаточно не присваивать переменной никакого значения:

```
dataflag=
```

Но на практике лучше указать пару кавычек сразу после знака равенства следующим образом:

```
dataflag=""
```

или таким образом:

```
dataflag=''
```

Так или иначе, переменной `dataflag` будет присвоено пустое значение. Это более наглядный способ намеренного присваивания пустого значения переменной, чем предыдущий способ, который может быть вполне воспринят как ошибка или опечатка.

Следует, однако, иметь в виду, что присваивание

```
dataflag=" "
```

не равнозначно трем предыдущим, поскольку в данном случае переменной dataflag присваивается один символ пробела. А ведь это совсем не одно и то же, что и *не* присвоить вообще никакого символа.

Подстановка имен файлов и переменных

Любопытно, если ввести следующее:

```
x=*
```

сохранит ли оболочка в переменной x знак * или же имена всех файлов из текущего каталога? Попробуйте выполнить следующий пример и убедитесь сами:

```
$ ls                          Показать имеющиеся файлы
addresses
intro
lotsaspaces
names
nu
numbers
phonebook
stat
$ x=*
$ echo $x
addresses intro lotsaSpaces names nu numbers phonebook stat
$
```

Из этого простого примера можно извлечь немало поучительного. В частности, был ли сохранен список файлов в переменной x при следующем присваивании:

```
x=*
```

или же оболочка выполнила подстановку при выполнении приведенной ниже команды?

```
echo $x
```

Оказывается, что оболочка не выполняет подстановку имен файлов, когда переменным присваиваются значения. Следовательно, в выражении

```
x=*
```

переменной x присваивается единственный знак *. Это означает, что оболочка должна выполнить подстановку имен файлов при выполнении команды echo, как следует из приведенного выше результата. В действительности, когда происходит присваивание

```
echo $x
```

оболочка выполняет следующую последовательность действий.

1. Просматривает командную строку, подставляя знак * вместо переменной x как ее значение.
2. Еще раз просматривает командную строку и, обнаружив знак *, подставляет вместо него имена всех файлов из текущего каталога.
3. Начинает выполнение команды `echo`, передав ей список файлов в качестве аргументов (рис. 4.3).

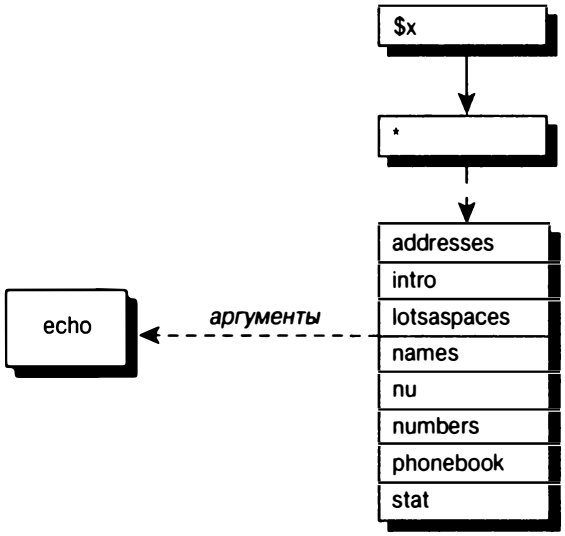


Рис. 4.3. Процесс выполнения команды **echo $x**

Такой порядок вычислений имеет особое значение. Напомним, что сначала оболочка подставляет значения переменных, затем имена файлов и далее выполняет синтаксический анализ командной строки, определяя аргументы команды.

Конструкция ${*переменная*}

Допустим, в переменной `filename` хранится имя файла. Если требуется переименовать этот файл, добавив к его старому имени букву X, первым побуждением станет ввести следующую команду:

```
mv $filename $filenameX
```

Но, просмотрев командную строку, оболочка подставит не только значение переменной `filename`, но и значение переменной `filenameX`. Ведь она интерпретирует `filenameX` как имя переменной, поскольку оно полностью состоит из достоверных символов.

Во избежание подобных недоразумений все имя переменной, кроме начального знака денежной единицы, следует заключить в фигурные скобки:

```
${filename}X
```

Благодаря этому устраняется любая неоднозначность, а команда mv действует должным образом:

```
mv $filename ${filename}X
```

Напомним, что фигурные скобки необходимо указывать лишь в том случае, если за последним символом в имени переменной следует буквенно-цифровой символ или знак подчеркивания. Над переменными в фигурных скобках можно выполнять разные операции, включая извлечение подмножеств, присваивание значений, если в настоящий момент переменной не присвоено никакого значения, и делать многое другое, как поясняется далее.

Встроенные целочисленные арифметические операции

Оболочка, совместимая со стандартом POSIX и входящая в состав всех современных версий Unix и Linux, включая и командный процессор в Mac OS X, предоставляет механизм для выполнения над переменными оболочки целочисленных арифметических операций, называемых *арифметическим расширением*. Однако в ряде прежних оболочек такая возможность не поддерживается.

Общий формат арифметического расширения выглядит следующим образом:

```
$ ( ( выражение ) )
```

где *выражение* — это арифметическое выражение, в котором применяются переменные оболочки и операции. Достоверными считаются такие переменные оболочки, которые содержат числовые значения, где допускаются начальные и конечные пробелы. А достоверные операции взяты из языка программирования C и перечислены в приложении A.

Операции, применяемые в арифметическом расширении

Как ни странно, в арифметическом расширении применяется довольно обширный перечень операций, включая шесть основных: **+**, **-**, *****, **/**, **%** и ******; более сложные, составные операции, в том числе **+=**, **-=**, ***=**, **/=**; а также операции простого инкрементирования и декрементирования в форме ***переменная*++** и ***переменная*--** и прочие.

Эти операции позволяют работать с разными системами счисления и даже преобразовывать числовые значения из одной системы в другую. В следующем примере определяются десятичные эквиваленты восьмеричного (по основанию 8) числа **100** и двоичного (по основанию 2) числа **101010101010101010**:

```
$ echo $(( 8#100 ))
64
$ echo $(( 2#101010101010101010 ))
174762
```

Результат вычисления заданного *выражения* подставляется в командной строке. Например, в команде

```
echo $(i+1)
```

выполняется приращение на **1** значения переменной i, а полученный результат выводится на терминал. Обратите внимание на то, что переменная i не предваряется знаком **$**, поскольку оболочке известно, что единственными элементами арифметических расширений могут быть только операции, числа и переменные. Если же переменная не определена или содержит нулевую (NULL) строку, ее значение считается нулевым. Так, если в следующем примере переменной a еще не присвоено значение, ее все равно можно использовать в целочисленном выражении:

```
$ echo $a                          Переменная a не установлена
$ echo $(( a = a + 1 ))            Равнозначно выражению a = 0 + 1
$ 1
$ echo $a
1                                  Теперь переменная a содержит значение 1
$
```

Следует иметь в виду, что присваивание считается достоверной операцией, и поэтому ее результирующее значение поставляется во второй команде echo из приведенного выше примера. Для группирования элементов в выражениях можно свободно пользоваться круглыми скобками, как показано ниже.

```
echo $((i = (i + 10) * j))
```

Если требуется выполнить присваивание без команды echo или какой-нибудь другой, эту операцию можно указать *до* арифметического расширения. Например, чтобы умножить значение переменной i на **5** и присвоить полученный результат обратно переменной i, достаточно написать следующую строку:

```
i=$(( i * 5 ))
```

Обратите внимание на пробелы в двойных скобках, хотя они совсем не обязательны. Но если присваивание выполняется за пределами двойных скобок, то пробелы в этой операции *не* допускаются.

Ниже приведено более лаконичное и чаще употребляемое обозначение составной операции умножения на **5** и присваивания полученного результата обратно переменной i. Такое обозначение составной операции можно употребить и в другом операторе.

```
$(( i *= 5 ))
```

Если требуется лишь прибавить **1** к значению переменной, это можно сделать более лаконично:

```
$(( i++ ))
```

И наконец, чтобы проверить, находится ли значение переменной `i` в пределах от **0** до **100** включительно, достаточно написать следующую строку:

```
result=$(( i >= 0 && i <= 100 ))
```

где переменной `result` присваивается значение **1** (**true**), если выражение истинно, или значение **0** (**false**), если выражение ложно, как демонстрируется в следующем примере:

```
$ i=$(( 100 * 200 / 10 ))
$ j=$(( i < 1000 ))              Если i < 1000, установить j = 0;
                                 а иначе — j = 1
$ echo $i $j
2000 0                           i = 2000, поэтому j = 0
$
```

На этом введение в написание команд и применение переменных завершается. В следующей главе будут подробно рассмотрены различные механизмы, применяемые в оболочке для заключения в кавычки.

Заключение в кавычки

В этой главе рассматривается характерная особенность языка программирования оболочки: порядок интерпретации знаков кавычек. В оболочке распознаются четыре типа знаков кавычек.

- Знак одиночной кавычки (').
- Знак двойной кавычки (").
- Знак обратной косой черты (\).
- Знак обратной кавычки (`).

Два первых и последний из приведенных выше знаков должны указываться парами, тогда как знак обратной косой черты может употребляться в любом количестве по мере надобности, возникающей в команде. Каждый из этих знаков имеет особое назначение в оболочке. Поэтому они рассматриваются в отдельных разделах этой главы.

Одиночная кавычка

Для употребления кавычек в оболочке имеются самые разные причины. Чаще всего их используют для составления в единый элемент последовательности символов, включая пробелы. В качестве примера рассмотрим снова файл phonebook, содержащий имена абонентов и номера их телефонов:

```
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Liz Stachiw         212-555-2298
Susan Goldberg      201-555-7776
Susan Topple        212-555-4932
Tony Iannino        973-555-1295
$
```

Чтобы найти сведения о конкретном абоненте в файле phonebook, достаточно ввести следующую команду grep:

```
$ grep Alice phonebook
Alice Chebba    973-555-2015
$
```

А вот что произойдет, если попытаться найти сведения об абоненте Susan:

```
$ grep Susan phonebook
Susan Goldberg    201-555-7776
Susan Topple      212-555-4932
$
```

В данном файле обнаруживаются сведения о двух абонентах по имени Susan, и поэтому в выводимом результате появляются две строки. Но допустим, что требуется получить сведения об абоненте Susan Goldberg. С этой целью можно, например, попробовать уточнить имя абонента, дополнив его фамилией:

```
$ grep Susan Goldberg phonebook
grep: can't open Goldberg
Susan Goldberg 201-555-7776
Susan Topple 212-555-4932
$
```

Как видите, такой прием не действует. Дело в том, что пробелы служат в оболочке для разделения аргументов команды. А приведенная выше командная строка интерпретируется как команда grep с тремя аргументами Susan, Goldberg и phonebook (рис. 5.1).

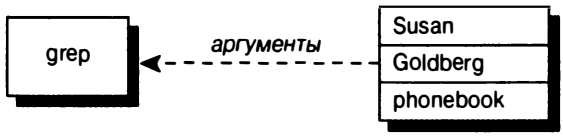


Рис. 5.1. Процесс выполнения команды **grep Susan Goldberg phonebook**

Когда команда grep выполняется, она интерпретирует первый аргумент как шаблон для поиска, а остальные аргументы — как имена файлов для поиска. В данном случае команда grep посчитает, что ей нужно найти абонента по имени Susan в файлах Goldberg и phonebook. С этой целью она попытается открыть файл Goldberg и, не найдя его, выдаст следующее сообщение об ошибке:

```
grep: can't open Goldberg
(grep: нельзя открыть файл Goldberg)
```

Затем она перейдет к следующему файлу, phonebook, откроет его, найдет в нем сведения об абоненте Susan и выведет две совпавшие строки. Как видите, все вполне логично.

В связи с изложенным выше возникает вопрос: как передавать командам и программам аргументы, включая пробелы? Ответ состоит в том, чтобы заключить аргумент в одиночные кавычки, как показано ниже.

```
grep 'Susan Goldberg' phonebook
```

Когда оболочка обнаруживает первую открывающую одиночную кавычку, она *игнорирует* любые последующие специальные символы до тех пор, пока не обнаружит вторую закрывающую одиночную кавычку, как показано в следующем примере:

```
$ grep 'Susan Goldberg' phonebook
Susan Goldberg    201-555-7776
$
```

Как только оболочка обнаружит первую открывающую одиночную кавычку (`'`), она прекратит интерпретировать любые специальные символы до тех пор, пока не обнаружит парную ей закрывающую одиночную кавычку (`'`). Поэтому пробел между именем `Susan` и фамилией `Goldberg`, который обычно разделяет два отдельных аргумента, игнорируется оболочкой. И тогда оболочка разделяет командную строку на *два* аргумента: первый — `Susan Goldberg`, включая пробел, и второй — `phonebook`. После этого оболочка вызывает команду `grep`, передавая ей оба эти аргумента (рис. 5.2).

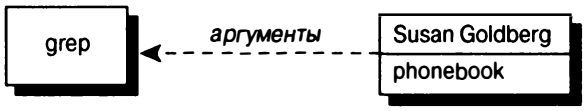


Рис. 5.2. Процесс выполнения команды **`grep 'Susan Goldberg' phonebook`**

Команда `grep`, в свою очередь, интерпретирует первый аргумент `Susan Goldberg` как шаблон, который содержит пробел, и осуществит по нему поиск в файле, указанном в качестве второго аргумента `phonebook`. Следует, однако, иметь в виду, что оболочка *удаляет* кавычки, не передавая их вызываемой программе или команде.

Сколько бы пробелов не заключить в кавычки, все они благополучно сохранятся оболочкой, как показано в следующем примере:

```
$ echo  one       two     three     four
one two three four
$ echo 'one       two     three     four'
one       two     three     four
$
```

В первом случае оболочка удаляет лишние пробелы из строки (без кавычек) и передает команде `echo` четыре аргумента: `one`, `two`, `three` и `four` (рис. 5.3).

Рис. 5.3. Процесс выполнения команды **echo one two three four**

А во втором случае лишние пробелы сохраняются, и при выполнении команды echo оболочка интерпретирует всю строку символов, заключенных в одиночные кавычки, как единый аргумент (рис. 5.4).

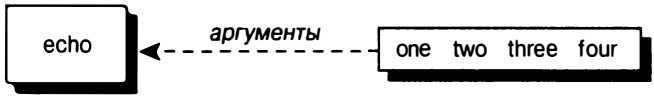


Рис. 5.4. Процесс выполнения команды **echo 'one two three four'**

Следует еще раз подчеркнуть, что *все* специальные символы игнорируются оболочкой, если они заключены в одиночные кавычки. Именно этим объясняются результаты, получаемые в следующем примере:

```
$ file=/users/steve/bin/progl
$ echo $file
/users/steve/bin/progl
$ echo '$file'                    Знак $ не интерпретируется
$file
$ echo *
addresses intro lotsaspaces names nu numbers phonebook stat
$ echo '*'
*
$ echo '< > | ; ( ) { } >> " &'
< > | ; ( ) { } >> " &
$
```

Даже нажатие клавиши <Enter> сохраняется как составная часть аргумента команды, если оно заключено в одиночные кавычки:

```
$ echo 'How are you today,
> John'
How are you today,
John
$
```

После синтаксического анализа первой строки в данном примере оболочка не обнаружит парную закрывающую кавычку и поэтому выдаст пользователю приглашение (>) на ввод закрывающей кавычки. Приглашение > называется *вспомогательным* и отображается оболочкой всякий раз, когда ожидается завершение ввода многострочной команды.

Кавычки требуются и в том случае, если переменным оболочки присваиваются значения, содержащие пробелы или специальные символы, хотя здесь имеются свои особенности, как демонстрируется в следующем примере:

```
$ message='I must say, this sure is fun'
$ echo $message
I must say, this sure is fun
$ text='* means all files in the directory'
$ echo $text
names nu numbers phonebook stat means all files in the directory
$
```

Одиночные кавычки требуются в первом случае потому, что значение, присваиваемое переменной, содержит пробелы. А во втором случае наглядно показано, что при подстановке значения вместо переменной `text` оболочка интерпретирует знак * в ее значении как указание на подстановку имен файлов из текущего каталога после этого значения, прежде чем выполнить команду `echo`, что весьма досадно! Как же разрешить затруднения подобного рода? Для этого следует воспользоваться двойными кавычками, о которых речь пойдет далее.

Двойные кавычки

Двойные кавычки действуют таким же образом, как и одиночные, за исключением того, что они в меньшей степени защищают свое содержимое. Если одиночные кавычки предписывают оболочке игнорировать *все* заключенные в них символы, то двойные кавычки — игнорировать б*о*льшую их часть. В частности, следующие три символа *не* игнорируются в двойных кавычках.

- Знаки денежных единиц.
- Знаки обратных кавычек.
- Знаки обратной косой черты.

Тот факт, что знаки денежных единиц не игнорируются в двойных кавычках, означает, что вместо имени переменной оболочка подставляет в двойных кавычках значение этой переменной, как демонстрируется в следующем примере:

```
$ filelist=*
$ echo $filelist
addresses intro lotsaspaces names nu numbers phonebook stat
$ echo '$filelist'
$filelist
$ echo "$filelist"
*
$
```

В данном примере демонстрируется главное отличие, проявляющееся в отсутствии кавычек и наличии одиночных и двойных кавычек. В первом случае оболочка обнаруживает знак звездочки и подставляет имена всех файлов из текущего каталога. Во втором случае оболочка оставляет без внимания символы, заключенные в одиночные кавычки, в результате чего выводится имя переменной `$filelist`. И в последнем случае двойные кавычки предписывают оболочке подставить вместо имени заключенной в них переменной ее значение (знак *). Но поскольку подстановка имен файлов *не* выполняется в двойных кавычках, то знак * благополучно передается команде `echo` как отображаемое значение.

Примечание

Обсуждая отличия одиночных кавычек от двойных, следует также иметь в виду, что оболочка не распознает парные кавычки “ и ”, формируемые такими текстовыми редакторами, как Microsoft Word, с целью придать набранному тексту более привлекательный вид при печати. Но дело в том, что употребление парных кавычек нарушает нормальную работу сценариев оболочки, и поэтому их следует всячески избегать!

Если требуется подставить значение вместо имени переменной, но исключить синтаксический анализ подставляемых символов в оболочке, такую переменную следует заключить в двойные кавычки. Ниже приведен пример, наглядно показывающий, чем отсутствие кавычек отличается от наличия двойных кавычек.

```
$ address="39 East 12th Street
> New York, N. Y. 10003"
$ echo $address
39 East 12th Street New York, N. Y. 10003
$ echo "$address"
39 East 12th Street
New York, N. Y. 10003
$
```

В данном конкретном примере не имеет особого значения, присваивается ли значение переменной `address` в одиночных или двойных кавычках. Оболочка в любом случае отображает вспомогательное приглашение на ввод команд, чтобы указать на то, что она ожидает ввода соответствующей закрывающей кавычки.

После присваивания двухстрочного адреса переменной `address` ее значение отображается по команде `echo`. В отсутствие кавычек вокруг имени этой переменной адрес отображается в одной строке по той же самой причине, что и следующий результат:

```
one two three four
```

выполнения команды:

```
echo one       two    three   four
```

Оболочка удаляет символы пробела, табуляции и новой строки (т.е. все пробельные символы) из командной строки, а затем разделяет ее на аргументы, прежде чем передать их запрашиваемой команде. Поэтому в результате вызова следующей команды:

```
echo $address
```

оболочка удалит встроенный в значение переменной address символ новой строки, интерпретируя его как символ пробела или табуляции, служащий в качестве разделителя аргументов. А затем оболочка передаст *девять* аргументов команде echo для последующего отображения. Таким образом, символ новой строки не дойдет до самой команды echo, поскольку он предварительно обрабатывается оболочкой (рис. 5.5).

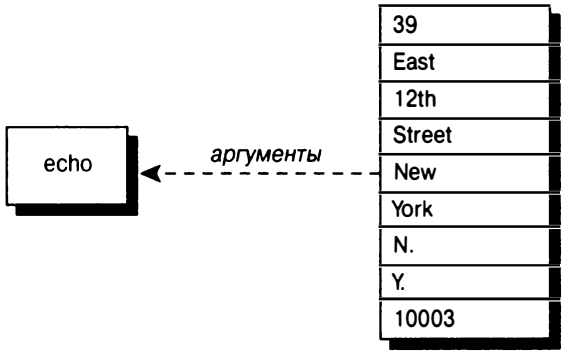


Рис. 5.5. Процесс выполнения команды **echo $address**

Когда же выполняется следующая команда:

```
echo "$address"
```

оболочка подставляет значение переменной address, как и прежде, но в данном случае двойные кавычки предписывают ей оставить без внимания любые заключенные в них пробелы. Таким образом, оболочка передаст команде echo единственный аргумент, содержащий символ новой строки. И команда echo отобразит свой единственный аргумент. На рис. 5.6 этот факт отображает комбинация символов \n, обозначающая новую строку.

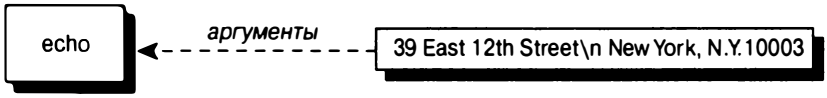


Рис. 5.6. Процесс выполнения команды **echo "$address"**

Дело несколько усложняется в том случае, если двойные кавычки употребляются для сокрытия от оболочки одиночных кавычек, и наоборот, как демонстрируется в следующем примере:

```
$ x="' Hello,' he said"
$ echo $x
'Hello,' he said
$ article=' "Keeping the Logins from Lagging," Bell Labs Record'
$ echo $article
"Keeping the Logins from Lagging," Bell Labs Record
$
```

Обратная косая черта

Функционально знак обратной косой черты, употребляемый в качестве префикса, равнозначен заключению одного символа в одиночные кавычки, хотя и за рядом незначительных исключений. Знак обратной косой черты *экранирует* следующий сразу же после него символ. Общая форма употребления знака обратной косой черты в качестве кавычек выглядит следующим образом:

```
\c
```

где *c* — символ, который требуется заключить в кавычки. В этом случае отменяется любое специальное назначение данного символа, как демонстрируется в следующем примере:

```
$ echo >
syntax error: 'newline or ;' unexpected
(синтаксическая ошибка: 'неожиданное начало строки или знак ;')
$ echo \>
>
$
```

В первом случае, обнаружив символ **>**, оболочка посчитает, что результат выполнения команды `echo` следует переадресовать в файл, и поэтому она ожидает обнаружить далее имя файла. А поскольку имя файла отсутствует, оболочка выдает сообщение об ошибке. А во втором случае знак обратной косой черты экранирует символ **>**, отменяя его специальное назначение, и поэтому он передается команде `echo` для последующего отображения.

В приведенном ниже примере оболочка проигнорирует знак **$**, следующий после знака обратной косой черты, и в конечном итоге подстановки значения вместо имени переменной не произойдет.

```
$ x=*
$ echo \$x
$x
$
```

В связи с тем что знак обратной косой черты отменяет специальное назначение следующего после него символа, нетрудно догадаться, что произойдет, если этим символом окажется еще один знак обратной косой черты: он отменит специальное назначение знака обратной косой черты, как показано ниже.

```
$ echo \\
\
$
```

Того же самого результата можно было бы добиться и с помощью одиночных кавычек:

```
$ echo '\'
\
$
```

Продолжение строк с помощью знака обратной косой черты

Как упоминалось ранее, обозначение *\c*, по существу, равнозначно обозначению *'c'*. Единственное исключение из этого правила возникает в том случае, если знак обратной косой черты употребляется в качестве последнего символа в строке, как демонстрируется в следующем примере:

```
$ lines=one'
> 'two                    Одиночные кавычки предписывают оболочке
                          проигнорировать знак новой строки
$ echo "$lines"
one
two
$ lines=one\              Попробовать ввести то же самое со знаком \
> two
$ echo "$lines"
onetwo
$
```

Когда знак обратной косой черты указывается последним во вводимой строке, оболочка интерпретирует его как знак продолжения строки. В этом случае она *удаляет* следующий далее знак новой строки, не интерпретируя его как разделитель аргументов, будто бы он вообще не был набран. Подобная конструкция нередко применяется для ввода длинных команд в нескольких строках.

Например, следующая операция присваивания вполне допустима:

```
Longinput="The shell treats a backslash that's the \
last character of a line of input as a line \
continuation. It removes the newline too."
```

Употребление обратной косой черты в двойных кавычках

Как отмечалось ранее, знак обратной косой черты относится к числу трех символов, интерпретируемых оболочкой в двойных кавычках. Это означает, что знак обратной косой черты можно применять в двойных кавычках для отмены специального назначения отдельных символов (знаков денежных единиц, обратных кавычек и еще одних двойных кавычек или обратных косых черт), которые в противном случае *будут* интерпретированы оболочкой. Если же знак обратной косой черты предшествует любому другому символу в двойных кавычках, он игнорируется оболочкой и передается программе или команде, как показано в следующем примере:

```
$ echo "\$x"
$x
$ echo "\ is the backslash character"
\ is the backslash character
$ x=5
$ echo "The value of x is \"$x\""
The value of x is "5"
$
```

В первом случае знак обратной косой черты предшествует знаку денежной единицы, и поэтому оболочка игнорирует знак денежной единицы, удаляет знак обратной косой черты и передает полученный результат команде `echo`. Во втором случае знак обратной косой черты предшествует пробелу, который *не* интерпретируется оболочкой в двойных кавычках. Поэтому оболочка игнорирует знак обратной косой черты, передавая его команде `echo` вместе с остальной строкой. И в последнем случае знаки обратной косой черты выполняют функцию двойных кавычек в строке, заключаемой в двойные кавычки.

В качестве еще одного примера применения кавычек допустим, что требуется вывести на терминал следующую строку:

```
<<< echo $x >>> displays the value of x, which is $x
```

В данном примере преследуется цель подставить значение переменной `x` во втором, но не в первом экземпляре `$x`. Присвоим сначала значение переменной `x`:

```
$ x=1
$
```

Если попытаться теперь отобразить эту строку без кавычек, оболочка выдаст сообщение о синтаксической ошибке, как показано ниже. Ведь знак **<** указывает оболочке на переадресацию, но после него не следует имя файла, и поэтому оболочка выдает сообщение об ошибке.

```
$ echo <<< echo $x >>> displays the value of x, which is $x
syntax error: '<' unexpected
$
```

Если заключить всю упомянутую выше строку в одиночные кавычки, значение переменной x не будет подставлено в конце данной строки. А если заключить всю эту строку в двойные кавычки, то значение переменной x будет подставлено вместо обоих экземпляров **$x**. Коварная ситуация!

Ниже приведены два способа правильно заключить строку в кавычки, чтобы добиться желаемого результата.

```
$ echo "<<< echo \$x >>> displays the value of x, which is $x"
<<< echo $x >>> displays the value of x, which is 1
$ echo '<<< echo $x >>> displays the value of x, which is' $x
<<< echo $x >>> displays the value of x, which is 1
$
```

В первом случае вся строка заключается в двойные кавычки, а знак обратной косой черты служит для того, чтобы предотвратить подстановку значения переменной x вместо первого экземпляра `$x`. А во втором случае строка заключается в одиночные кавычки вплоть до второго экземпляра `$x`, добавляемого как переменная без кавычек, вместо имени которой следует подставить ее значение.

Но последний способ заключения в кавычки таит в себе следующую едва заметную опасность: если переменная x содержит подстановку имени файла или символы пробела, они будут интерпретированы. Поэтому рассматриваемую здесь команду `echo` надежнее написать следующим образом:

```
echo '<<< echo $x >>> displays the value of x, which is' "$x"
```

Подстановка команд

Подстановка команд означает для оболочки возможность заменить указанную команду результатом ее выполнения в любом месте командной строки. Выполнить подстановку команд в оболочке можно двумя способами: заключив команду в обратные кавычки или введя ее в конструкцию `$(...)`.

Обратные кавычки

В отличие от всех упоминавшихся ранее типов кавычек, назначение обратных кавычек состоит не в том, чтобы защитить символы от интерпретации в оболочке, но предписать ей заменить заключенную в них команду результатом ее выполнения. Общая форма применения обратных кавычек выглядит следующим образом:

`` `команда` ``

где *команда* — имя команды, результат выполнения которой подставляется в данном месте.

Примечание

Употреблять обратные кавычки для подстановки команд больше *не* рекомендуется. Тем не менее мы рассматриваем такой способ подстановки команд потому, что он применяется в большом количестве прежних сценариев оболочки. Кроме того, о возможностях обратных кавычек следует знать на тот случай, если придется писать программы оболочки, переносимые в прежние версии систем Unix с оболочками, где *не* поддерживается новая и более предпочтительная конструкция **$(...)**.

Рассмотрим следующий пример подстановки команды, заключаемой в обратные кавычки:

```
$ echo The date and time is: `date`
The date and time is: Wed Aug 28 14:28:43 EDT 2002
$
```

Когда оболочка выполняет первоначальный просмотр командной строки, она распознает обратную кавычку, ожидая вслед за ней команду. В данном случае оболочка обнаруживает команду `date` и поэтому выполняет ее, заменяя последовательность символов `` `date` `` в командной строке результатом выполнения команды `date`. После этого оболочка разделяет полученную в итоге командную строку на аргументы, как обычно, передавая их команде `echo`.

В приведенном ниже примере оболочка выполняет команду `pwd`, подставляет результат ее выполнения в командную строку, а затем выполняет команду `echo`. В примерах из следующего раздела обратные кавычки могут быть использованы везде, где применяется рассматриваемая далее конструкция `$(...)`, а последняя — разумеется, в примерах из этого раздела.

```
$ echo Your current working directory is `pwd`
Your current working directory is /users/steve/shell/ch6
$
```

Конструкция $(...)

В современных оболочках Unix, Linux и прочих, совместимых со стандартом POSIX, для подстановки команд поддерживается новая и более предпочтительная конструкция `$(...)`. Общая форма этой конструкции выглядит следующим образом:

```
$(команда)
```

где *команда* — это, как и в обратных кавычках, имя команды, стандартный вывод из которой подставляется в командной строке. Рассмотрим следующий пример:

```
$ echo The date and time is: $(date)
The date and time is: Wed Aug 28 14:28:43 EDT 2002
$
```

Такая конструкция лучше обратных кавычек по двум причинам. Во-первых, в сложных командах прямые и обратные кавычки могут употребляться в различных сочетаниях, что затрудняет их понимание, особенно если выбранное для написания команд начертание шрифта не позволяет визуально отличить одиночные кавычки от обратных. И, во-вторых, конструкция `$(...)` допускает простое вложение, а следовательно, подстановку одной команды в другую. И хотя обратные кавычки также допускают вложение, сделать это труднее. Пример вложенной подстановки команд будет представлен далее в этом разделе.

Следует особо подчеркнуть, что в круглых скобках данной конструкции можно вызывать не одну, а несколько команд, разделяемых точками с запятой. И зачастую для этой цели могут также использоваться каналы.

Ниже приведен вариант программы `nu` для отображения количества пользователей, зарегистрированных в системе. Он был видоизменен с тем, чтобы продемонстрировать применение новой конструкции `$(...)`.

```
$ cat nu
echo There are $(who | wc -l) users logged in
$ nu                                    Выполнить программу
There are 13 users logged in
$
```

Одиночные кавычки защищают все, что в них заключено, и поэтому результат выполнения следующей команды вполне очевиден:

```
$ echo '$(who | wc -l) tells how many users are logged in'
$(who | wc -l) tells how many users are logged in
$
```

Но подстановка команд все же *интерпретируется* в двойных кавычках, как показано ниже

```
$ echo "You have $(ls | wc -l) files in your directory"
You have       7 files in your directory
$
```

Напомним, что оболочка отвечает за выполнение команды, заключаемой в круглые скобки рассматриваемой здесь конструкции. А команда `echo` замечает только подставляемый оболочкой результат выполнения команды, указанной в круглых скобках.

Примечание

Начальные пробелы, получающиеся в результате выполнения команды **wc** в приведенном выше примере, неизменно вызывают досаду у программистов. Попробуйте найти способ избавиться от них с помощью команды **sed**.

Допустим, в создаваемой программе оболочки требуется присвоить текущие дату и время переменной now. Для этого можно воспользоваться подстановкой команд следующим образом:

```
$ now=$(date)              Выполнить команду date и сохранить
                           результат в переменной now
$ echo $now                Посмотреть, что присвоено переменной now
Wed Aug 28 14:47:26 EDT 2002
$
```

Если ввести следующую строку:

```
now=$(date)
```

интерпретировав ее, оболочка присвоит переменной now весь результат выполнения команды date. Следовательно, заключать конструкцию $(date) в двойные кавычки не нужно, хотя это обычно делается на практике.

В переменной могут быть сохранены даже те команды, которые выдают результат в нескольких строках:

```
$ filelist=$(ls)
$ echo $filelist
addresses intro lotsaspaces names nu numbers phonebook stat
$
```

Что же здесь произошло? В конечном итоге в одной строке был выведен перечень всех файлов из текущего каталога, хотя знаки новой строки из команды ls были сохранены в переменной filelist. Эти знаки были поглощены при отображении значения переменной filelist, подставленного оболочкой в процессе обработки командной строки echo. Знаки новой строки можно сохранить, если заключить переменную filelist в двойные кавычки, как в следующем примере:

```
$ echo "$filelist"
addresses
intro
lotsaspaces
names
nu
numbers
phonebook
stat
$
```

Продвигаясь далее в область написания сценариев оболочки, выясним, как применять рассматриваемую здесь конструкцию в сочетании с переадресацией стандартного ввода-вывода в файл. Например, чтобы сохранить содержимое файла в переменной, можно воспользоваться следующей удобной командой cat:

```
$ namelist=$(cat names)
$ echo "$namelist"
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
Tony
$
```

Если же требуется разослать почтой содержимое файла memo всем адресатам, перечисленным в файле names, это можно сделать следующим образом:

```
$ mail $(cat names) < memo
$
```

В данном случае оболочка заменяет сначала команду cat результатом ее выполнения, и поэтому командная строка принимает приведенный ниже вид. А затем оболочка выполняет команду mail, переадресовывая ее стандартный ввод из файла memo и передавая ей семь указанных получателей почты в качестве аргументов.

```
mail Charlie Emanuel Fred Lucy Ralph Tony Tony < memo
```

Обратите внимание на то, что адресат Tony может получить одно и то же почтовое сообщение дважды, поскольку он два раза перечислен в файле names. Чтобы устранить любые дублирующиеся записи из файла, вместо команды cat можно воспользоваться командой sort с параметром **-u**, удаляющим дубликаты строк, как показано ниже. В итоге каждый адресат получит почтовое сообщение только один раз.

```
$ mail $(sort -u names) < memo
$
```

Напомним, что оболочка выполняет подстановку имен файлов *после* подстановки команд. Заключив команды в двойные кавычки, можно исключить подстановку имен файлов при выводе результатов.

Подстановка команд нередко применяется с целью изменить значение, хранящееся в переменной оболочки. Так, если переменная оболочки name содержит чье-то имя, где все буквы требуется сделать прописными, для этого можно воспользоваться командой echo, чтобы передать переменную name на вход команды tr, выполнить преобразование, а затем присвоить результат обратно переменной name, как показано ниже.

```
$ name="Ralph Kramden"
$ name=$(echo $name | tr '[a-z]' '[A-Z]')        Преобразовать
                                                 в прописные буквы
$ echo $name
RALPH KRAMDEN
$
```

Способ употребления команды echo в конвейере для передачи данных на вход последующей команды прост, но эффективен и нередко применяется в программах оболочки. В следующем примере демонстрируется, каким образом с помощью команды cut можно извлечь первый символ из значений, хранящихся в переменной filename:

```
$ filename=/users/steve/memos
$ firstchar=$(echo $filename | cut -c1)
$ echo $firstchar
/
$
```

Отредактировать значение, хранящееся в переменной, зачастую можно и с помощью команды sed. В следующем примере эта команда используется для извлечения последнего символа из каждой строки в файле, указанном в переменной file:

```
$ file=exec.o
$ lastchar=$(echo $file | sed 's/.*\(.\)$/\1/')
$ echo $lastchar
o
$
```

Команда sed заменяет все символы в строке последним ее символом. (Напомним, что шаблон в круглых скобках приводит к сохранению последнего символа в указанном регистре. В данном случае это регистр 1, на который делается ссылка \1.) Подставляемый результат выполнения команды sed сохраняется в переменной lastchar. Команда sed заключена в одиночные кавычки для того, чтобы оболочка не пыталась интерпретировать знаки обратной косой черты, применяемые в этой команде. (Вопрос: подошли бы для этой цели и двойные кавычки?)

И наконец, подстановки команд могут быть вложенными. Допустим, требуется изменить каждое вхождение первого символа в переменной на что-нибудь другое. В приведенном ранее примере команда firstchar=$(echo $filename | cut -c1) извлекает первый символ из переменной filename, но как воспользоваться этим символом, чтобы изменить каждое его вхождение в переменной filename? С одной стороны, эту операцию можно выполнить в два этапа, как показано ниже.

```
$ filename=/users/steve/memos
$ firstchar=$(echo $filename | cut -c1)
$ filename=$(echo $filename | tr "$firstchar" "^")     Преобразовать
                                                       знак / в знак ^
$ echo $filename
^users^steve^memos
$
```

А с другой стороны, ту же самую операцию можно выполнить в течение одной вложенной подстановки команд следующим образом:

```
$ filename=/users/steve/memos
$ filename=$(echo $filename | tr "$(echo $filename | cut -c1)" "^")
$ echo $filename
^users^steve^memos
$
```

Если вам трудно понять приведенный выше пример, сравните его с предыдущим примером, обратив внимание на то, что переменная `firstchar` заменена в нем вложенной подстановкой команд. А в остальном оба примера одинаковы.

Команда `expr`

Несмотря на то что в стандартной оболочке поддерживаются встроенные целочисленные арифметические операции, в прежних оболочках такая поддержка отсутствует. В таком случае может пригодиться команда `expr` для решения математических уравнений:

```
$ expr 1 + 2
3
$
```

Команда `expr` проста в обращении, но она не вполне пригодна для синтаксического анализа уравнений. Поэтому каждую операцию и операнд, указываемые в команде `expr`, необходимо отделять пробелами для правильной их интерпретации. Это обстоятельство поясняется в следующем примере:

```
$ expr 1+2
1+2
$
```

Команда `expr` распознает обычные арифметические операции: сложения (**+**), вычитания (**-**), деления (**/**), умножения (*****) и взятия по модулю (**%**; получения остатка от деления). Ниже приведен характерный тому пример.

```
$ expr 10 + 20 / 2
20
$
```

Операции умножения, деления и взятия по модулю обладают более высоким приоритетом, чем операции сложения и вычитания, как и в обычной арифметике. Следовательно, в предыдущем примере деление выполняется прежде сложения. А что же неверно в приведенном ниже примере?

```
$ expr 17 * 6
expr: syntax error
$
```

В данном примере оболочка, обнаружив знак *, подставила имена всех файлов в каталоге, которые команда expr просто не в состоянии правильно интерпретировать! Если требуется выполнить операцию умножения, выражение, указываемое в команде expr, должно быть заключено в кавычки, чтобы защитить его от интерпретации в оболочке. Но оно не должно быть единственным аргументом, как в следующем примере:

```
$ expr "17 * 6"
17 * 6
$
```

Напомним, что команда expr должна различать каждую операцию и операнд как отдельный аргумент. А в приведенном выше примере все выражение указано в качестве единственного аргумента данной команды, поэтому она и не дает желаемого результата.

И здесь на помощь приходят знаки обратной косой черты, как показано в следующем примере:

```
$ expr 17 \* 6
102
$
```

Естественно, что аргументами команды expr могут служить значения, хранящиеся в переменных оболочки, которая берет на себя обязанность заблаговременно подставить их:

```
$ i=1
$ expr $i + 1
2
$
```

Этот устаревший способ выполнения арифметических операций над переменными оболочки менее эффективен, чем новая конструкция $(...). Если требуется инкрементировать или иным образом видоизменить значение переменной, для этого можно воспользоваться механизмом подстановки команд, присвоив результат выполнения команды expr обратно переменной, как демонстрируется в следующем примере:

```
$ i=1
$ i=$(expr $i + 1)          Увеличить значение переменной i на 1
$ echo $i
2
$
```

В устаревших программах оболочки вероятнее всего можно обнаружить использование обратных кавычек в команде expr:

```
$ i=`expr $i + 1`          Увеличить значение переменной i на 1
$ echo $i
3
$
```

Подобно встроенным арифметическим операциям, команда `expr` вычисляет только целочисленные арифметические выражения. А для математических расчетов с плавающей точкой можно воспользоваться командами `awk` и `bc`. Отличие заключается в том, что число **17** является целым, тогда как число **13.747** — с плавающей (десятичной) точкой.

Следует также иметь в виду, что в команде `expr` допускаются и другие операции. Чаще всего применяется операция для сопоставления символов первого операнда с регулярным выражением, указываемым в качестве второго операнда. По умолчанию такая операция возвращает количество совпавших символов.

Например, следующая команда `expr`:

```
expr "$file" : ".*"
```

возвращает количество символов, хранящихся в переменной `file`, поскольку регулярное выражение **.*** обеспечивает совпадение со всеми символами в строке. Подробнее о команде `expr` и эффективной конструкции **:** можно узнать из *Руководства пользователя системы Unix* или на соответствующей оперативной странице руководства `man` по команде `expr` в конкретной системе. А способы обработки кавычек в оболочке сведены в табл. А.5 приложения А.

6

Передача аргументов

Пользы от программ оболочки станет намного больше, если научиться обрабатывать передаваемые им аргументы. Из материала этой главы вы узнаете, как писать программы, принимающие аргументы в командной строке. Напомним, что в главе 4 был рассмотрен пример однострочной программы `run`, предназначенной для обработки файла `sys.caps` по командам `tbl`, `nroff` и `lp`:

```
$ cat run
tbl sys.caps | nroff -mm -Tlp | lp
$
```

Допустим, с помощью той же самой последовательности команд требуется обработать не только файл `sys.caps`, но и другие файлы. Для обработки каждого такого файла можно было бы создать отдельную версию программы `run` или видоизменить ее таким образом, чтобы иметь возможность указывать имя обрабатываемого файла в командной строке. Такое видоизменение программы `run` означало бы возможность набрать ее в командной строке следующим образом:

```
run new.hire
```

чтобы указать имя файла `new.hire` для обработки с помощью данной последовательности команд, или же таким образом:

```
run sys.caps
```

чтобы указать имя обрабатываемого файла `sys.caps`.

Всякий раз, когда выполняется программа оболочки, первый ее аргумент сохраняется в специальной переменной оболочки `1`, второй аргумент — в переменной оболочки `2` и т.д. (Для большего удобства здесь и далее эти переменные будут обозначаться как `$1`, `$2` и т.д., несмотря на то, что знак **$** фактически является частью обозначения ссылки на переменную, а не ее имени.) Эти специальные переменные формально называются *позиционными параметрами*, поскольку они опираются на позиции значений, указываемых в командной строке и присваиваемых после обычной процедуры обработки содержимого командной строки в оболочке, т.е. переадресации ввода-вывода, подстановки переменных и имен файлов и т.д.

Чтобы программа `run` принимала имя файла в качестве аргумента, достаточно заменить в ней ссылку на файл `sys.caps` ссылкой на первый аргумент, набранный в командной строке:

```
$ cat run
tbl $1 | nroff -mm -Tlp | lp
$
$ run new.hire                  Выполнить с именем файла new.hire
                                в качестве аргумента
request id is laserl-24 (standard input)
$
```

Всякий раз, когда выполняется программа run, все, что набрано после ее имени в командной строке, сначала сохраняется оболочкой в первом позиционном параметре, а затем передается самой программе. Так, в приведенном выше примере имя файла new.hire сохраняется в этом параметре.

Подстановка позиционных параметров осуществляется таким же образом, как и подстановка других типов переменных. Следовательно, когда оболочка обнаруживает команду

```
tbl $1
```

она заменяет ссылку $1 первым аргументом, которым снабжается программа run (в данном случае это имя файла new.hire).

В качестве еще одного примера ниже приведена программа ison, которая позволяет выяснить, зарегистрирован ли указанный пользователь в системе.

```
$ cat ison
who | grep $1
$ who              Показать, кто зарегистрирован в системе
root      console  Jul 7  08:37
barney    tty03    Jul 8  12:28
fred      tty04    Jul 8  13:40
joanne    tty07    Jul 8  09:35
tony      tty19    Jul 8  08:30
lulu      tty23    Jul 8  09:55
$ ison tony
tony      tty19    Jul 8  08:30
$ ison pat
$                  Этот пользователь не зарегистрирован в системе
```

Переменная $#

Помимо позиционных параметров, в оболочке имеется специальная переменная $#, принимающая ряд аргументов, набранных в командной строке. Как будет показано в следующей главе, содержимое этой переменной нередко проверяется в программах на правильность количества аргументов, указанных пользователем.

В следующем примере демонстрируется программа args, написанная с единственной с целью — показать, каким образом аргументы передаются программе оболочки. Внимательно проанализируйте каждый результат ее выполнения, чтобы хорошо понимать, как она действует.

```
$ cat args                    Показать программу
echo $# arguments passed
echo arg 1 = :$1: arg 2 = :$2: arg 3 = :$3:
$ args a b c                  Выполнить программу
3 arguments passed
arg 1 = :a: arg 2 = :b: arg 3 = :c:
$ args a b                  Опробовать программу с другими аргументами
2 arguments passed
arg 1 = :a: arg 2 = :b: arg 3 = ::          Неприсвоенные аргументы
                                            оказываются пустыми
$ args                        Попробовать выполнить программу
                              без аргументов
0 arguments passed
arg 1 =:: arg 2 =:: arg 3 = ::
$ args "a b c"                Попробовать указать аргументы в кавычках
1 arguments passed
arg 1 = :a b c: arg 2 = :: arg 3 = ::
$ ls x*                       Выяснить имена файлов, начинающихся
                              на букву x
xact
xtra
$ args x*                     Попробовать подставить имя файла
2 arguments passed
arg 1 = :xact: arg 2 = :xtra: arg 3 = ::
$ my_bin=/users/steve/bin
$ args $my_bin                И подставить переменную
1 arguments passed
arg 1 = :/users/steve/bin: arg 2 = :: arg 3 = ::
$ args $(cat names)           Передать содержимое файла names
7 arguments passed
arg 1 = :Charlie: arg 2 = :Emanuel: arg3 = :Fred:
$
```

Как видите, оболочка обрабатывает содержимое командной строки обычным образом, когда она выполняет написанные для нее программы. Это означает, что, указывая аргументы в своих программах оболочки, можно выгодно воспользоваться такими привычными удобствами, как подстановка имен файлов и переменных.

Переменная $*

Специальная переменная $* служит для ссылки на *все* аргументы, передаваемые программе. Это нередко оказывается удобным в тех программах, которые принимают неопределенное или *переменное* количество аргументов. Далее будут представлены более практические примеры, а в следующем примере программы демонстрируется применение переменной $*:

```
$ cat args2
echo $# arguments passed
echo they are :$*:
$ args2 a b c
```

```
3 arguments passed
they are :a b c:
$ args2 one                two
2 arguments passed
they are :one two:
$ args2
0 arguments passed
they are ::
$ args2 *
8 arguments passed
they are :args args2 names nu phonebook stat xact xtra:
$
```

Программа для поиска абонентов в телефонном справочнике

Ниже приведено содержимое файла phonebook из предыдущих примеров.

```
$ cat phonebook
Alice Chebba       973-555-2015
Barbara Swingle    201-555-9257
Liz Stachiw        212-555-2298
Susan Goldberg     201-555-7776
Susan Topple       212-555-4932
Tony Iannino       973-555-1295
$
```

Как известно, найти абонента в этом файле нетрудно по команде grep:

```
$ grep Cheb phonebook
Alice Chebba       973-555-2015
$
```

Известно также, что если требуется найти абонента по имени и фамилии, их следует указать в кавычках как единый аргумент:

```
$ grep "Susan T" phonebook
Susan Topple       212-555-4932
$
```

Но было бы неплохо написать отдельную программу оболочки для поиска абонентов в телефонном справочнике. Такая программа может называться lu и должна принимать в качестве своего аргумента имя искомого абонента:

```
$ cat lu
#
# Найти абонента в телефонном справочнике
#

grep $1 phonebook
$
```

Ниже приведены конкретные примеры применения программы `lu`.

```
$ lu Alice
Alice Chebba      973-555-2015
$ lu Susan
Susan Goldberg    201-555-7776
Susan Topple      212-555-4932
$ lu "Susan T"
grep: can't open T
phonebook:Susan Goldberg    201-555-7776
phonebook:Susan Topple      212-555-4932
$
```

В последнем из приведенных выше примеров имя и фамилия `Susan T` были аккуратно заключены в кавычки. В чем же тогда дело? Чтобы выяснить недоразумение, рассмотрим еще раз следующий вызов команды `grep` из программы `lu`:

```
grep $1 phonebook
```

Как видите, заключение в кавычки имени и фамилии `Susan T` приводит к тому, что они передаются программе `lu` как единый аргумент `$1`. Но когда оболочка подставляет его значение из командной строки в самой программе, то команде `grep` оно фактически передается как *два* аргумента. Этот недостаток можно устранить, заключив в двойные кавычки аргумент `$1` непосредственно в программе `lu`:

```
$ cat lu
#
# Найти абонента в телефонном справочнике - версия 2
#

grep "$1" phonebook
$
```

Одиночные кавычки в данном случае не подойдут. А теперь попробуем снова выполнить программу `lu`, как показано ниже.

```
$ lu Tony
Tony Iannino 973-555-1295          Действует по-прежнему
$ lu "Susan T"                     А теперь попробуем ввести
                                   тот же самый аргумент снова
Susan Topple    212-555-4932
$
```

Программа для ввода абонентов в телефонный справочник

Продолжим разработку программ для обработки файла phonebook. В какой-то момент может возникнуть потребность ввести нового абонента в этот файл телефонного справочника, особенно если он невелик. С этой целью напишем приведенную ниже программу add, принимающую два аргумента: имя абонента, вводимого в телефонный справочник, и номер его телефона.

```
$ cat add
#
# Ввести нового абонента в телефонный справочник
#

echo "$1          $2" >> phonebook
$
```

И хотя этого не видно, на самом деле аргументы $1 и $2 разделяются в приведенной выше команде echo знаком табуляции. Этот знак должен быть указан в кавычках, чтобы воспроизводиться по команде echo, а не "поглощаться" оболочкой. Опробуем новую программу следующим образом:

```
$ add 'Stromboli Pizza' 973-555-9478
$ lu Pizza              Проверить, удастся ли найти новую запись
Stromboli Pizza 973-555-9478              Пока что все нормально
$ cat phonebook         Вывести содержимое телефонного справочника
Alice Chebba      973-555-2015
Barbara Swingle   201-555-9257
Liz Stachiw       212-555-2298
Susan Goldberg    201-555-7776
Susan Topple      212-555-4932
Tony Iannino      973-555-1295
Stromboli Pizza   973-555-9478
$
```

Имя и фамилия Stromboli Pizza были заключены в одиночные кавычки, чтобы оболочка передала их программе add как единый аргумент. (А что бы произошло, если бы имя и фамилия не были заключены в одиночные кавычки?) После программы add в приведенном выше примере была выполнена программа lu, чтобы выяснить, удастся ли найти новую запись в телефонном справочника, что и было благополучно сделано. А затем была выполнена команда cat, чтобы посмотреть, каким образом теперь выглядит видоизмененный телефонный справочник. Как и предполагалось, в самом его конце появилась новая запись.

К сожалению, новый файл стал неотсортированным. И хотя сортировка содержимого этого файла не оказывает никакого влияния на выполнение программы lu, тем не менее, она является полезным свойством. С этой целью в программу add можно ввести команду сортировки sort, как показано ниже.

```
$ cat add
#
# Ввести нового абонента в телефонный справочник - версия 2
#

echo "$1          $2" >> phonebook
sort -o phonebook phonebook
$
```

Напомним, что параметр **-o** команды `sort` определяет место для вывода отсортированного результата. В данном случае это тот же самый входной файл, как показано ниже. Всякий раз, когда в файл `phonebook` вводится новая запись, его содержимое сортируется снова, чтобы многострочные совпадения с критерием поиска всегда располагались в алфавитном порядке.

```
$ add 'Billy Bach' 201-555-7618
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Billy Bach          201-555-7618
Liz Stachiw         212-555-2298
Stromboli Pizza     973-555-9478
Susan Goldberg      201-555-7776
Susan Topple        212-555-4932
Tony Iannino        973-555-1295
$
```

Программа для удаления абонентов из телефонного справочника

Ни один комплект программ, позволяющих искать или вводить абонентов в телефонный справочник, не будет полным без программы удаления абонентов из телефонного справочника. Поэтому напишем программу `rem`, позволяющую удалить из телефонного справочника абонента по имени, указанному в качестве аргумента в командной строке.

Какой должна быть стратегия разработки такой программы? По существу, из файла требуется удалить строку, содержащую указанное имя абонента, что противоположно совпадению с шаблоном. В данном случае можно воспользоваться параметром **-v** команды `grep`, поскольку он позволяет сделать именно то, что требуется: вывести из файла все строки, которые *не* совпадают с шаблоном:

```
$ cat rem
#
# Удалить абонента из телефонного справочника
#

grep -v "$1" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
$
```

По команде grep с параметром **-v** все строки из файла phonebook, не совпадающие с заданным шаблоном, выводятся в файл /tmp/phonebook. (*Примечание:* каталог /tmp служит в системах Unix для хранения временных файлов и обычно удаляется при перезапуске системы.) А после выполнения команды grep прежний файл phonebook заменяется новым файлом из каталога /tmp. Ниже приведены некоторые примеры применения вновь созданной программы rm.

```
$ rem 'Stromboli Pizza'          Удалить эту запись
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Billy Bach          201-555-7618
Liz Stachiw         212-555-2298
Susan Goldberg      201-555-7776
Susan Topple        212-555-4932
Tony Iannino        973-555-1295
$ rem Susan
$ cat phonebook
Alice Chebba        973-555-2015
Barbara Swingle     201-555-9257
Billy Bach          201-555-7618
Liz Stachiw         212-555-2298
Tony Iannino        973-555-1295
$
```

В первом случае из файла phonebook была благополучно удалена запись Stromboli Pizza. Но во втором случае из него были удалены обе записи с именем Susan, поскольку обе они совпали с заданным шаблоном, а это не совсем верно! Чтобы ввести эти записи обратно в файл phonebook, можно воспользоваться программой add следующим образом:

```
$ add 'Susan Goldberg' 201-555-7776
$ add 'Susan Topple' 212-555-4932
$
```

В главе 7 будет показано, как проверять действие прежде, чем выполнять его, чтобы в рассматриваемом здесь примере программы можно было определить факт обнаружения не одной, а нескольких совпадающих записей. Но в программе может, например, возникнуть потребность известить пользователя об обнаружении нескольких совпавших записей вместо слепого их удаления. (Это было бы очень удобно, поскольку в большинстве реализаций команды grep произойдет совпадение со всем, что угодно, если в качестве шаблона передать ей пустую символьную строку. По существу, это привело бы к удалению всего содержимого файла phonebook, что было бы неверно!)

Между прочим, совпавшую запись можно было бы удалить и по команде sed. В таком случае команду grep можно было бы заменить следующей командой:

```
sed "/$1/d" phonebook > /tmp/phonebook
```

чтобы добиться аналогичного результата. Аргумент команды sed необходимо заключить в двойные кавычки, чтобы обеспечить подстановку значения аргумента $1 и в то же время гарантировать, что оболочка не отреагирует на команду, аналогичную приведенной ниже, где команде sed передаются три, а не два аргумента.

```
sed /Stromboli Pizza/d phonebook > /tmp/phonebook
```

Конструкция ${n}

Если предоставить программе больше девяти аргументов, то десятый и последующие аргументы ($10, $11 и т.д.) окажутся недоступными. И если попытаться получить доступ к десятому аргументу по ссылке $10, то оболочка фактически подставит значение аргумента $1, а затем значение **0**. Вместо этого следует воспользоваться конструкцией ${*n*}. Так, чтобы получить непосредственный доступ к десятому аргументу, следует указать ссылку ${10} в своей программе и так далее для одиннадцатого, двенадцатого и остальных аргументов.

Команда shift

Эта команда позволяет, по существу, *смещать влево* позиционные параметры. Если выполнить команду shift, предыдущее значение позиционного параметра $2 будет присвоено позиционному параметру $1, а предыдущее значение позиционного параметра $3 — позиционному параметру $2 и т.д. В то же время прежнее значение позиционного параметра $1 будет безвозвратно утрачено.

При выполнении этой команды значение переменной $#, содержащей количество аргументов, также автоматически уменьшается на единицу, как показано ниже.

```
$ cat tshift                    Программа для проверки смещения
                                позиционных параметров
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
$ tshift a b c d e
5 a b c d e
4 b c d e
3 c d e
2 d e
1 e
0
$
```

Если попытаться выполнить команду `shift` в отсутствие переменных для смещения (т.е. когда значение переменной `$#` уже равно нулю), то оболочка выдаст следующее сообщение об ошибке, хотя его содержимое зависит от конкретной версии оболочки:

```
prog: shift: bad number
```

где *prog* — имя программы, в которой была выполнена команда `shift`, а `bad number` — неверное количество аргументов.

Смещение может быть произведено сразу на несколько позиций, если указать их количество при вызове команды `shift`. Так, выполнение следующей команды:

```
shift 3
```

дает такой же результат, как и выполнение подряд трех команд `shift` без аргументов:

```
shift
shift
shift
```

Команда `shift` удобна для обработки переменного количества аргументов. Ее практическое применение будет продемонстрировано в главе 8, когда речь пойдет о циклах. А до тех пор достаточно запомнить, что позиционные параметры можно продвигать по цепочке, используя команду `shift`.

7

Выбор по условию

В этой главе рассматривается условный оператор `if` — конструкция, присутствующая практически в каждом языке программирования. Этот оператор позволяет сначала проверить заданное условие, а затем изменить ход выполнения программы в зависимости от результата проверки. Ниже приведена общая форма команды, реализующей условный оператор `if`.

```
if команда1
then
  команда
  команда
...
fi
```

где *команда*$_1$ — выполняемая команда и проверяемый *код ее завершения*. Так, если код завершения равен нулю, выполняются команды, указанные в промежутке между операторами `then` и `fi`, а иначе эти команды пропускаются.

Код завершения

Чтобы понять принцип действия условного оператора, важно знать, каким образом в системе Unix обрабатывается так называемый *код завершения*. Всякий раз, когда программа завершает свое выполнение, она возвращает оболочке соответствующий код завершения. Условно нулевой код завершения обозначает удачный исход выполнения программы, а ненулевой код завершения — неудачный исход, причем разные значения кода завершения обозначают разные виды неудачного исхода.

Неудачный исход может быть обусловлен тем, что программе переданы недостоверные аргументы или в ней обнаружено ошибочное условие. Например, команда `cp` возвращает ненулевой (сбойный) код завершения, если копия оказывается неудачной (например, если нельзя найти исходный файл или создать целевой файл) или же если аргументы указаны неверно (например, неверное количество аргументов или больше двух аргументов, последний из которых не является каталогом).

Что же означает ненулевой код завершения? Он означает любое целочисленное значение, не равное нулю. На большинстве оперативных страниц руководства по командам системы Unix перечисляются и возможные коды завершения. Так, возможными ошибочными условиями завершения команды копирования

файлов могли бы быть следующие: **1** (файл не найден), **2** (файл недоступен для чтения), **3** (целевая папка не найдена), **4** (целевая папка недоступна для записи), **5** (общая ошибка копирования файла) и, конечно, **0** (успешное завершение).

Что же касается команды `grep`, то она возвращает нулевой код завершения, если обнаружит заданный шаблон, по крайней мере, в одном из указанных файлов, или ненулевой код завершения, если найти шаблон не удалось или если возникло другое ошибочное условие вроде сбоя при открытии указанного исходного файла. А в конвейере код завершения отражает состояние последней команды в канале. Таким образом, в следующем конвейере:

```
who | grep fred
```

код завершения команды `grep` используется оболочкой в качестве кода завершения всего конвейера. В этом случае нулевой код (успешного) завершения означает, что пользователь `fred` был обнаружен в результате выполнения команды `who` (т.е. пользователь `fred` был зарегистрирован в тот момент, когда выполнялась эта команда).

Переменная $?

В переменной `$?` оболочка автоматически устанавливает код завершения команды, выполнявшейся последней. Естественно, что для вывода значения этого кода на терминал можно воспользоваться командой `echo`, как показано ниже. Следует, однако, заметить, что числовой результат "сбоя" в некоторой команде может изменяться при переходе от одной версии Unix к следующей, но удачный исход всегда обозначается нулевым кодом завершения.

```
$ cp phonebook phone2
$ echo $?
0                       "Удачный" исход копирования
$ cp nosuch backup
cp: cannot access nosuch
$ echo $?
2                       "Неудачный" исход копирования
$ who                   Выяснить, кто зарегистрирован в системе
root     console   Jul 8  10:06
wilma    tty03     Jul 8  12:36
barney   tty04     Jul 8  14:57
betty    tty15     Jul 8  15:03
$ who | grep barney
barney   tty04     Jul 8  14:57
$ echo $?               Вывести код завершения последней
                        команды (grep)
0                       "Удачный" исход выполнения команды grep
$ who | grep fred
$ echo $?
1                       "Неудачный" исход выполнения команды grep
$ echo $?
0                       Код завершения последней команды echo
$
```

В качестве примера напишем программу оболочки on, которая сообщает, зарегистрирован ли указанный пользователь в системе. Имя проверяемого пользователя будет передано этой программе из командной строки. Если указанный пользователь зарегистрирован в системе, данная программа выведет соответствующее сообщение, а иначе — ничего не сообщит. Ниже показано, каким образом выглядит программа on.

```
$ cat on
#
# Выяснить, зарегистрирован ли указанный пользователь в системе
#

user="$1"

if who | grep "$user"
then
      echo "$user is logged on"
fi
$
```

Сначала первый аргумент, набранный в командной строке, сохраняется в переменной оболочки user. Затем в команде if выполняется следующий конвейер команд:

```
who | grep "$user"
```

и проверяется код завершения, возвращаемый командой grep. Если код завершения оказывается нулевым (т.е. обозначает удачный исход), это означает, что команда grep обнаружила пользователя, имя которого хранится в переменной user, в переданном ей по конвейеру результате выполнения команды who. А если код завершения оказывается ненулевым (т.е. обозначает неудачный исход), то это означает, что указанный пользователь не зарегистрирован в системе, и поэтому команда echo пропускается.

Команда echo указана в рассматриваемой здесь программе с отступом от левого края исключительно из эстетических соображений. В данном случае в промежутке между операторами then и fi заключена лишь одна команда. Если же таких команд оказывается больше, а их вложенность углубляется, то наличие отступов значительно повышает удобочитаемость программы. Это обстоятельство будет наглядно продемонстрировано в последующих примерах. Ниже приведены некоторые примеры применения программы on.

```
$ who
root      console   Jul 8  10:37
barney    tty03     Jul 8  12:38
fred      tty04     Jul 8  13:40
joanne    tty07     Jul 8  09:35
tony      tty19     Jul 8  08:30
lulu      tty23     Jul 8  09:55
```

```
$ on tony                          Известно, что этот пользователь
                                   зарегистрирован в системе
tony      tty19     Jul 8  08:30       Где он был зарегистрирован?
Tony is logged on
$ on steve                         Известно, что этот пользователь не
                                   зарегистрирован в системе
$ on ann                           Попробовать выяснить присутствие
                                   этого пользователя в системе
joanne    tty07     Jul 8  09:35
ann is logged on
$
```

Рассматриваемой здесь программе все же присущи определенные недостатки. Если указанный пользовать зарегистрирован в системе, соответствующая строка из результата, выводимого командой who, любезно отображается при вызове команды grep. И хотя в этом нет ничего предосудительного, тем не менее, от данной программы требуется выводить только сообщение о том, что указанный пользователь зарегистрирован в системе, и ничего больше.

Упомянутая выше строка отображается вследствие проверки по условию. Ведь команда grep не только возвращает код своего завершения в следующий конвейер:

```
who | grep "$user"
```

но и выполняет свою обычную функцию, направляя любые совпавшие строки в стандартный вывод, даже если они нас не интересуют.

Но поскольку нас интересует не результат выполнения команды grep, а только код ее завершения, то мы можем избавиться от этого результата, переадресовав его в системный "мусорный бачок" по пути /dev/null, где находится специальный системный файл, куда любой пользователь может записывать или откуда читать, получая сразу же признак конца файла. При записи в этот файл информация исчезает как в огромной черной дыре! Поэтому недостаток вывода избыточной информации разрешается следующим образом:

```
who | grep "$user" > /dev/null
```

Другой недостаток программы on проявляется, когда она выполняется с аргументом ann. Несмотря на то что пользователь ann не зарегистрирован в системе, команда grep обнаруживает совпадение с символами ann в имени пользователя joanne. Поэтому в данном случае требуется более строгий шаблон, о составлении которого шла речь при рассмотрении регулярных выражений в главе 3. По команде who в первом столбце выводимого результата перечисляется имя каждого пользователя, работающего в системе, и поэтому составляемый шаблон можно ограничить сопоставлением с началом строки, предварив этот шаблон знаком ^, как показано ниже.

```
who | grep "^$user" > /dev/null
```

Но и этого оказывается недостаточно. Ведь, осуществляя поиск по шаблону вроде bob, команда все равно обнаружит строку, аналогичную следующей:

```
bobby      tty07    Jul 8  09:35
```

Поэтому шаблон необходимо ограничить и справа. Если учесть, что имя каждого пользователя выводится по команде who с одним или двумя пробелами, то приведенный ниже видоизмененный шаблон обеспечит совпадение только с теми строками, где имеется указанное имя пользователя.

```
"^$user "
```

Таким образом, оба недостатка рассматриваемой здесь программы устранены! А теперь опробуем новую, видоизмененную версию этой программы:

```
$ cat on
#
# Выяснить, зарегистрирован ли указанный пользователь
# в системе - версия 2
#

user="$1"
if who | grep "^$user " > /dev/null
then
  echo "$user is logged on"
fi
$ who                          Кто теперь зарегистрирован в системе?
Root      console    Jul 8  10:37
barney    tty03      Jul 8  12:38
fred      tty04      Jul 8  13:40
joanne    tty07      Jul 8  09:35
tony      tty19      Jul 8  08:30
lulu      tty23      Jul 8  09:55
$ on lulu
lulu is logged on
$ on ann                       Попробовать выяснить присутствие
                               этого пользователя в системе
$ on                           Выяснить, что произойдет, если
                               вообще не указать никаких аргументов?
$
```

Если аргументы вообще не указаны, переменная user окажется пустой и команда grep будет искать в результате, выводимом командой who, строки, начинающиеся с пробела (подумайте, почему?). Не обнаружив ни одной из таких строк, команда grep просто возвратит приглашение на ввод в командной строке. В следующем разделе будет показано, как проверить, было ли указано правильное количество аргументов программы, и если оно оказалось неверным, то предпринять соответствующее действие.

Команда test

Несмотря на то что для проверки условия в операторе if из предыдущего примера программы применялся конвейер, для проверки одного или нескольких условий намного чаще употребляется встроенная в оболочку команда test. Ниже приведена общая форма этой команды.

```
test выражение
```

где *выражение* обозначает проверяемое условие. Команда test вычисляет выражение, если оно дает *истинный* результат (TRUE), то возвращается нулевой код завершения, а если результат оказывается *ложным* (FALSE), то возвращается ненулевой код завершения.

Строковые операции

В качестве примера ниже приведена команда, возвращающая нулевой код завершения, если переменная оболочки name содержит символьную строку "julio".

```
test "$name" = julio
```

Операция = служит для проверки равенства двух значений. В данном случае проверяется, равно ли *содержимое* переменной оболочки name символьной строке "julio". Если такое равенство подтверждается, команда test возвращает нулевой код завершения, а иначе — ненулевой код завершения.

Следует, однако, иметь в виду, что все операнды (name и julio) и операции (=) должны быть доступны команде test в качестве отдельных аргументов. Это означает, что они должны быть разделены одним или несколькими символами пробела.

Вернемся снова к команде, реализующей условный оператор if. Чтобы вывести по команде echo сообщение "Would you like to play a game?" (Не хотите ли поиграть?), при условии, что в переменной name содержится символьная строка "julio", достаточно написать следующую команду if:

```
if test "$name" = julio
then
      echo "Would you like to play a game?"
fi
```

В условном операторе if выполняется следующая за ним команда и вычисляется код ее завершения. В частности, команде test передаются три аргумента: переменная name, вместо ссылки на которую подставляется ее значение, операция = и символьная строка "julio". И тогда в команде test проверяется, равны ли первый и третий ее аргументы. Если они равны, возвращает нулевой код завершения, а иначе — ненулевой код завершения.

Затем в условном операторе `if` проверяется код завершения, возвращаемый командой `test`. Если этот код нулевой, выполняются команды, находящиеся в промежутке между операторами `then` и `fi`. В данном случае выполняется единственная команда `echo`. А если код завершения оказывается ненулевым, то команда `echo` пропускается.

Как было показано выше, переменные оболочки, принимающие аргументы команды `test`, целесообразно заключать в *двойные* кавычки, чтобы разрешить постановку значений этих переменных. Этим гарантируется, что команда `test` выявит аргумент, даже если он принимает пустое значение. Для иллюстрации этого положения рассмотрим следующий пример:

```
$ name=                    Задать пустое значение переменной $name
$ test $name = julio
sh: test: argument expected
$
```

В данном примере задано пустое значение переменной `name`, и поэтому команде `test` передано только два аргумента: **=** и **julio**. Ведь оболочка подставила значение переменной `name` *прежде* синтаксического анализа аргументов в командной строке. На самом деле после постановки значения переменной `name` получается так, как будто набрана следующая команда:

```
test = julio
```

Когда эта команда выполняется, она обнаруживает лишь два аргумента, как показано на рис. 7.1. И поэтому выдается сообщение об ошибке, где извещается, что ожидаемый аргумент отсутствует.

Рис. 7.1. Процесс выполнения команды **test $name = julio** при пустом значении переменной **$name**

Заключив переменную в двойные кавычки, можно гарантировать, что команда `test` обнаружит аргумент, поскольку двойные кавычки служат "заполнителем" пустого значения аргумента. Несмотря на то что переменной `name` присвоено пустое значение, команде `test` передаются три аргумента, причем первый из них принимает пустое значение, как показано ниже и на рис. 7.2.

```
$ test "$name" = julio
$ echo $?                  Вывести код завершения
1
$
```

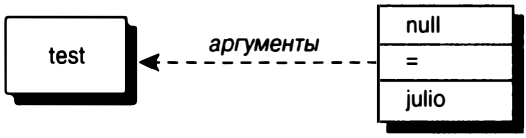


Рис. 7.2. Процесс выполнения команды **`test "$name" = julio`** при пустом значении переменной **`$name`**

Для проверки символьных строк можно применять и другие операции. Все они перечислены в табл. 7.1.

Таблица 7.1. Строковые операции в команде `test`

Операция	Возвращает *истинный* результат (нулевой код завершения), если
***строка*₁ = *строка*₂**	***строка*₁** и ***строка*₂** одинаковы
***строка*₁ != *строка*₂**	***строка*₁** и ***строка*₂** *не* одинаковы
строка	***строка*** не нулевая
-n *строка*	***строка*** не нулевая (и должна быть обнаружена командой **`test`**)
-z *строка*	***строка*** нулевая (и должна быть обнаружена командой **`test`**)

Выше было показано, каким образом применяется операция **=**. Аналогичным образом применяется и операция **!=**, за исключением того, что в ней проверяется неравенство символьных строк. Это означает, что команда `test` возвращает нулевой код завершения, если две символьные строки неравны, а если они равны — ненулевой код завершения. Рассмотрим три сходных примера:

```
$ day="monday"
$ test "$day" = monday
$ echo $?
0                    Истинно
$
```

В данном примере команда `test` возвращает нулевой код завершения, поскольку значение переменной `day` равно символьной строке `"monday"`. А теперь приведем следующий пример:

```
$ day="monday"
$ test "$day" = monday
$ echo $?
1                      Ложно
$
```

В данном примере переменной `day` присвоены символы `monday` и *последующий пробельный символ*. А проверка равенства двух строк в данном случае дает *ложный* результат, поскольку символьная строка `"monday"` не равна символьной строке `"monday"`.

Если требуется обеспечить равенство этих строк, следует опустить двойные кавычки, в которые заключается ссылка на переменную. В этом случае оболочка "поглотит" конечный пробельный символ, и команда `test` вообще не обнаружит его, как показано ниже.

```
$ day="monday"
$ test $day = monday
$ echo $?
0                    Истинно
$
```

И хотя это, по-видимому, нарушает упомянутое выше правило всегда заключать в двойные кавычки переменные, принимающие аргументы команды `test`, тем не менее, опускать двойные кавычки вполне допустимо, если точно известно, что значение переменной не является пустым (и не состоит только из пробельных символов).

Чтобы проверить, не является ли значение переменной оболочки пустым, можно воспользоваться операцией, перечисленной третьей в табл. 7.1:

```
test "$day"
```

Эта команда возвратит *истинный* результат (логическое значение `TRUE`), если значение переменной `day` не является пустым, а иначе — *ложный* результат (логическое значение `FALSE`). Двойные кавычки в данном случае не нужны, поскольку команде все равно, обнаружит ли она вообще аргумент. Тем не менее их лучше указать. Ведь если значение переменной состоит только из пробельных символов, оболочка исключит аргумент, если он не заключен в двойные кавычки.

```
$ blanks="    "
$ test $blanks              Не пустое ли это значение?
$ echo $?
1                           Ложно – это пустое значение
$ test "$blanks"            А теперь?
$ echo $?
0                           Истинно – это непустое значение
$
```

В первом случае команде `test` не было передано *никаких* аргументов, поскольку оболочка "поглотила" четыре пробела в переменной `blanks`. А во втором случае команде `test` был передан один аргумент, значение которого состоит из четырех пробелов и не является пустым.

Если вам покажется, что мы уделяем здесь слишком много внимания пробелам и кавычкам, имейте в виду, что именно они нередко оказываются источником едва уловимых программных ошибок. Рассмотренные здесь принципы стоит твердо усвоить теперь, чтобы застраховаться от многих неприятностей при программировании в дальнейшем.

Чтобы проверить, является ли символьная строка нулевой, можно также воспользоваться двумя операциями, перечисленными последними в табл. 7.1. В частности, операция **-n** возвращает нулевой код завершения, если указанный после нее аргумент оказывается непустым. Эту операцию можно рассматривать как проверку на ненулевую длину аргумента.

А в операции **-z** проверяется, не является ли пустым следующий после нее аргумент, и в этом случае возвращается нулевой код завершения. Эту операцию можно рассматривать как проверку на нулевую длину аргумента.

Таким образом, следующая команда:

```
test -n "$day"
```

возвратит нулевой код завершения, если переменная `$day` содержит хотя бы один символ. А команда

```
test -z "$dataflag"
```

возвратит нулевой код завершения, если переменная `dataflag` *не* содержит ни одного символа. Это означает, что операции **-n** и **-z** противоположны по своему действию. Обе они существуют лишь для того, чтобы упростить написание условных операторов и сделать их удобочитаемыми.

Следует, однако, иметь в виду, что в обеих упомянутых выше операциях предполагается обязательное наличие аргумента. Поэтому возьмите себе за правило заключать их аргументы в двойные кавычки.

```
$ nullvar=
$ nonnullvar=abc
$ test -n "$nullvar"        Имеет ли аргумент nullvar ненулевую длину?
$ echo $?
1                           Нет, не имеет
$ test -n "$nonnullvar"     А что же аргумент nonnullvar?
$ echo $?
0                           Да, имеет
$ test -z "$nullvar"        Имеет ли аргумент nullvar нулевую длину?
$ echo $?
0                           Да, имеет
$ test -z "$nonnullvar"     А что же аргумент nonnullvar?
$ echo $?
1                           Нет, не имеет
$
```

Следует также иметь в виду, что команда `test` может быть весьма требовательной к своим аргументам. Так, если переменная оболочки `symbol` содержит знак равенства, то любопытно выяснить, что же произойдет, если попытаться проверить ее на нулевую длину:

```
$ echo $symbol
=
$ test -z "$symbol"
sh: test: argument expected
$
```

Операция = обладает более высоким приоритетом, чем операция **-z**, и поэтому в команде `test` предпринимается попытка выполнить проверку на равенство, а следовательно, в ней предполагается наличие аргумента после операции =. Во избежание подобного рода недоразумений многие программирующие на языке оболочки предпочитают писать свои команды `test` следующим образом:

```
test X"$symbol" = X
```

Эта команда даст истинный результат, если значение переменной `symbol` окажется пустым, а иначе — ложный результат. Наличие символа X перед переменной `symbol` не позволяет команде `test` интерпретировать символы, хранящиеся в переменной `symbol`, как операцию.

Альтернативная форма команды `test`

Команда `test` нередко применяется программирующими на языке оболочки в альтернативной, более лаконичной форме **[**. Эта форма повышает удобочитаемость условных операторов `if` и прочих проверок по условию в сценариях оболочки.

Напомним, что общая форма команды `test` выглядит следующим образом:

```
test выражение
```

Эту же команду можно выразить и в следующей альтернативной форме:

```
[ выражение ]
```

Знак открывающей квадратной скобки (**[**) фактически обозначает наименование команды. (Как видите, наименования команд могут состоять не только из букв и цифр!) И в этой форме начинается выполнение той же самой команды `test`, но в такой форме предполагается также наличие знака закрывающей квадратной скобки (**]**) в конце выражения. После знака **[** и перед знаком **]** должны быть указаны пробелы.

Команду `test` из предыдущего примера можно переписать в альтернативной форме следующим образом:

```
$ [ -z "$nonnullvar" ]
$ echo $?
1
$
```

В условном операторе `if` альтернативна форма команды `test` выглядит следующим образом:

```
if [ "$name" = julio ]
then
      echo "Would you like to play a game?"
fi
```

Выбор конкретной формы команды `test` отдается на ваше усмотрение. Мы же предпочитаем форму **`[ . . . ]`**, которая делает программирование на языке оболочки в большей степени подобным программированию на других распространенных языках, и поэтому именно она будет использоваться в примерах, приведенных в остальной части данной книги.

Целочисленные операции сравнения

Команда `test` имеет большой арсенал операций для выполнения целочисленного сравнения. Эти операции сведены в табл. 7.2.

Таблица 7.2. Целочисленные операции сравнения в команде `test`

Операция	Возвращает *истинный* результат (нулевой код завершения), если
***int*$_1$ `-eq` *int*$_2$**	***int*$_1$** равно ***int*$_2$**
***int*$_1$ `-ge` *int*$_2$**	***int*$_1$** больше или равно ***int*$_2$**
***int*$_1$ `-gt` *int*$_2$**	***int*$_1$** больше ***int*$_2$**
***int*$_1$ `-le` *int*$_2$**	***int*$_1$** меньше или равно ***int*$_2$**
***int*$_1$ `-lt` *int*$_2$**	***int*$_1$** меньше ***int*$_2$**
***int*$_1$ `-ne` *int*$_2$**	***int*$_1$** не равно ***int*$_2$**

Например, в операции **`-eq`** проверяется, равны ли два целочисленных значения. Так, если имеется переменная оболочки `count` и требуется выяснить, равно ли ее значение нулю, достаточно написать следующую команду:

```
[ "$count" -eq 0 ]
```

Аналогичным образом ведут себя остальные целочисленные операции. Так, в следующей операции проверяется, является ли значение переменной `choice` меньшим **5**:

```
[ "$choice" -lt 5 ]
```

А в приведенной ниже команде проверяется неравенство значений переменных `index` и `max`.

```
 [ "$index" -ne "$max" ]
```

И наконец, в следующей команде проверяется, не равно ли нулю количество передаваемых ей аргументов:

```
[ "$#" -ne 0 ]
```

Следует, однако, иметь в виду, что именно команда `test`, а не сама оболочка интерпретирует значение переменной как целочисленное, когда применяется целочисленная операция. Таким образом, рассматриваемые здесь целочисленные операции сравнения действуют независимо от типа переменной оболочки.

Рассмотрим более подробно отличия строковых и целочисленных операций в команде `test` на ряде следующих примеров:

```
$ x1="005"
$ x2=" 10"
$ [ "$x1" = 5 ]                  Сравнение строк
$ echo $?
1                                Ложный результат
$ [ "$x1" -eq 5 ]                Сравнение целых чисел
$ echo $?
0                                Истинный результат
$ [ "$x2" = 10 ]                 Сравнение строк
$ echo $?
1                                Ложный результат
$ [ "$x2" -eq 10 ]               Сравнение целых чисел
$ echo $?
0                                Истинный результат
$
```

При первой поверке:

```
[ "$x1" = 5 ]
```

применяется *строковая* операция сравнения **=** двух символьных строк на равенство. Они не равны, поскольку первая строка состоит из трех символов **005**, а вторая — из единственного символа **5**.

При второй проверке применяется целочисленная операция сравнения **-eq**. Если интерпретировать оба сравниваемых значения как целочисленные, то **005** равно **5**, что и подтверждает код завершения команды `test`.

Третья и четвертая проверки сходны, но при этом можно заметить, как наличие даже начального пробела в переменной `x2` способно оказать влияние на результат проверки с помощью строковой операции, в отличие от целочисленной операции.

Файловые операции

Буквально в каждой программе оболочки приходится оперировать одним или несколькими файлами. Именно поэтому команда `test` снабжена обширным рядом операций, позволяющих решать различные задачи манипулирования

файлами. Каждая из этих операций носит *унарный* характер, а это означает, что в них предполагается единственный аргумент. Но в любом случае в качестве второго аргумента служит имя файла, включая и имя каталога там, где это уместно. В табл. 7.3 перечислены наиболее употребительные файловые операции.

Таблица 7.3. Наиболее употребительные файловые операции в команде `test`

Операция	Возвращает *истинный* результат (нулевой код завершения), если
`-d` ***файл***	***файл*** — это каталог
`-e` ***файл***	***файл*** — это существующий файл
`-f` ***файл***	***файл*** — это обыкновенный файл
`-r` ***файл***	***файл*** — это доступный для чтения файл
`-s` ***файл***	***файл*** — это файл ненулевой длины
`-w` ***файл***	***файл*** — это доступный для записи файл
`-x` ***файл***	***файл*** — это исполняемый файл
`-L` ***файл***	***файл*** — это символическая ссылка

В команде

```
[ -f /users/steve/phonebook ]
```

проверяется, существует ли файл `/users/steve/phonebook` и является ли он обыкновенным, т.е. не специальным файлом или каталогом.

А в приведенной выше команде проверяется, существует ли указанный файл и является ли он доступным для чтения.

```
[ -r /users/steve/phonebook ]
```

И наконец, по следующей команде:

```
[ -s /users/steve/phonebook ]
```

проверяется, имеет ли указанный файл ненулевое содержимое (т.е. не является пустым). Это удобно, если сначала создается файл регистрации ошибок, а затем требуется выяснить, было ли что-нибудь в него записано.

```
if [ -s $ERRFILE ]
then
      echo "Errors found:"
      cat $ERRFILE
fi
```

В сочетании с описанными ранее операциями ряд других операций, доступных в команде `test`, позволяет составлять более сложные условные выражения.

Операция логического отрицания !

Унарная логическая операция **!** может быть указана перед любым другим выражением в команде `test` для *отрицания* результата вычисления данного выражения. Например, следующая команда:

```
[ ! -r /users/steve/phonebook ]
```

возвращает нулевой код завершения (т.е. истинный результат), если файл `/users/steve/phonebook` оказывается *недоступным* для чтения, а команда

```
[ ! -f "$mailfile" ]
```

возвращает истинный результат, если файл, указанный в переменной `$mailfile`, *не* существует или *не* является обыкновенным файлом. И наконец, следующая команда:

```
[ ! "$x1" = "$x2" ]
```

возвращает истинный результат, если значения операндов `$x1` и `$x2` неодинаковы, что, очевидно, равнозначно приведенному ниже, более понятному условному выражению.

```
[ "$x1" != "$x2" ]
```

Операция -a логическое И

Операция **-a** выполняет логическую операцию И над двумя выражениями и возвращает истинный результат (логическое значение `TRUE`), если оба соединяемые выражения оказываются истинными. Так, следующая команда:

```
[ -f "$mailfile" -a -r "$mailfile" ]
```

возвращает истинный результат, если файл, указанный в переменной `$mailfile`, является обыкновенным и доступным для чтения файлом. (В данном примере операция **-a** специально отделена пробелами для повышения удобочитаемости условного выражения, что, очевидно, не оказывает никакого влияния на выполнение этой операции.)

А команда

```
[ "$count" -ge 0 -a "$count" -lt 10 ]
```

возвратит истинный результат, если переменная `count` содержит целочисленное значение, большее или равное нулю, но меньшее **10**. Операция **-a** имеет более низкий *приоритет*, чем целочисленные операции сравнения (как, впрочем, строковые и файловые операции). Это означает, что приведенное выше выражение, очевидно, вычисляется как следующее выражение:

```
("$count" -ge 0) -a ("$count" -lt 10)
```

В некоторых случаях важно знать, что команда `test` сразу же прекращает вычисление логического выражения И, как только что-нибудь в нем окажется ложным. Так, оператор, подобный следующему:

```
[ ! -f "$file" -a $(who > $file) ]
```

не будет выполнен в подоболочке с вызовом команды `who`, если проверка `! -f` (на несуществование) не пройдет, поскольку команде `test` уже известно, что логическое выражение И (в операции **`-a`**) ложно. Это может поставить программирующих на языке оболочки в затруднительное положение, если они попытаются втиснуть слишком много операций в условные выражения, предполагая, что все они будут выполнены до проверок по условию.

Круглые скобки

В условном выражении команды `test` *можно* воспользоваться круглыми скобками, чтобы изменить порядок вычисления требуемым образом. Нужно лишь экранировать сами круглые скобки, поскольку у них имеется особое назначение в оболочке. Таким образом, предыдущий пример применения команды `test` можно переписать следующим образом:

```
[ \( "$count" -ge 0 \) -a \( "$count" -lt 10 \) ]
```

Как правило, круглые скобки должны быть отделены пробелами, поскольку в команде `test` предполагается обнаружить каждый элемент условного выражения в качестве отдельного аргумента.

Операция `-o` логическое ИЛИ

Операция **`-o`** подобна операции **`-a`**, но только она образует логическое ИЛИ из двух выражений. Это означает, что вычисление всего выражения даст истинный результат, если истинно первое *или* второе *или* же оба исходных выражения.

```
[ -n "$mailopt" -o -r $HOME/mailfile ]
```

Данная команда возвратит истинный результат, если значение переменной `$mailopt` не равно нулю *или* же если файл `$HOME/mailfile` доступен для чтения. Операция **`-o`** имеет более низкий приоритет, чем операция **`-a`**, а это означает, что следующее выражение:

```
"$a" -eq 0 -o "$b" -eq 2 -a "$c" -eq 10
```

будет вычислено в команде `test` как приведенное ниже выражение.

```
"$a" -eq 0 -o ("$b" -eq 2 -a "$c" -eq 10)
```

Естественно, что такой порядок вычисления можно при необходимости изменить с помощью круглых скобок следующим образом:

```
\( "$a" -eq 0 -o "$b" -eq 2 \) -a "$c" -eq 10
```

Приоритет выполнения операций в сложных условных операторах имеет решающее значение, и поэтому многие программирующие на языке оболочки пользуются вложенными уловными операторами `if`, чтобы обойти любые затруднения, а другие — круглыми скобками, чтобы прояснить порядок интерпретации. Очевидно, что допускать порядок интерпретации слева направо весьма опрометчиво!

В примерах, приведенных в данной книге, можно обнаружить немало примеров применения команды `test`, поскольку практически невозможно написать даже самую простую программу оболочки без определенного рода условных выражений. Все операции, доступные в команде `test`, сведены в табл. А.11 приложения А к данной книге.

Конструкция `else`

Конструкция, называемая условным оператором `else`, может быть добавлена к условному оператору `if` в одноименной команде. Ниже приведена ее общая форма.

```
if команда1
then
      команда
      команда

else
      команда
      команда

fi
```

В приведенном выше примере выполняется указанная *команда*$_1$ и вычисляется код ее завершения. Если этот код оказывается нулевым, выполняется блок кода, следующий после оператора `then`, т.е. все команды, находящиеся в промежутке между операторами `then` и `else`, а блок кода, следующий после оператора `else`, т.е. все команды, находящиеся в промежутке между операторами `else` и `fi`, игнорируется. Но в любом случае может быть выполнена лишь одна из этих последовательностей команд: первая из них, если код завершения равен нулю, а вторая — если этот код не равен нулю.

Более лаконично это можно пояснить следующим образом:

```
if условие then команды_если_истинный_результат
else команды_если_ложный_результат fi
```

А теперь рассмотрим видоизмененный вариант программы `on`. Вместо того чтобы вообще не выводить ничего, если запрошенный пользователь не зарегистрирован в системе, программа `on` должна сообщать пользователю об этом факте. Ниже приведена третья версия данной программы.

```
$ cat on
#
# Выяснить, зарегистрирован ли указанный пользователь
# в системе - версия 3
#

user="$1"

if who | grep "^$user " > /dev/null
then
      echo "$user is logged on"
else
      echo "$user is not logged on"
fi
$
```

Если пользователь указан в качестве первого аргумента программы `on` и команда `grep` завершится успешно, на терминал будет выведено сообщение "*`user`* `is logged on`" (*пользователь* зарегистрирован), а иначе — сообщение "*`user`* `is not logged on`" (*пользователь* не зарегистрирован):

```
$ who                              Кто работает в системе?
Root       console    Jul 8  10:37
barney     tty03      Jul 8  12:38
fred       tty04      Jul 8  13:40
joanne     tty07      Jul 8  09:35
tony       tty19      Jul 8  08:30
lulu       tty23      Jul 8  09:55
$ on pat
pat is not logged on
$ on tony
tony is logged on
$
```

Чтобы превратить созданный на скорую руку прототип в программу, которая станет полезной на долгое время, целесообразно проверить, передано ли этой программе правильное количество аргументов. Если же пользователь укажет неверное количество аргументов, должно быть выведено соответствующее сообщение об ошибке наряду с сообщением о правильном применении данной программы:

```
$ cat on
#
# Выяснить, зарегистрирован ли указанный пользователь
# в системе - версия 4
#
#
# Выяснить, было ли предоставлено правильное количество аргументов
#
```

```
if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments"
      echo "Usage: on user"
else
      user="$1"

      if who | grep "^$user " > /dev/null
      then
            echo "$user is logged on"
      else
            echo "$user is not logged on"
      fi
fi
$
```

На первый взгляд, в данную программу внесено немало изменений, но самое главное, что для проверки правильности количества предоставленных аргументов в нее введен дополнительный условный оператор `if` вместо того, чтобы втиснуть эту проверку в пару условных операторов `else-fi`. Если значение переменной `$#` не равно **1**, что соответствует требующемуся количеству аргументов, то данная программа выводит два сообщения об ошибках. В противном случае выполняются команды, следующие после условного оператора `else`. Обратите внимание на то, что в данном случае требуются два условных оператора `fi`, поскольку применяются два условных оператора `if`.

Как видите, отступы заметно повышают удобочитаемость данной программы. Поэтому возьмите себе за правило употреблять отступы в своих программах, и вы еще не раз с благодарностью вспомните эту нашу рекомендацию, когда ваши программы станут намного сложнее.

Как показано ниже, взаимодействие с конечным пользователем в рассматриваемой здесь версии программы `on` заметно улучшилось по сравнению с предыдущими ее версиями.

```
$ on                                  Без аргументов
Incorrect number of arguments
Usage: on user
$ on priscilla                        Один аргумент
priscilla is not logged on
$ on jo anne                          Два аргумента
Incorrect number of arguments
Usage: on user
$
```

Команда exit

Встроенная в оболочку команда exit позволяет немедленно прервать выполнение программы оболочки. Ниже приведена общая форма этой команды.

```
exit n
```

где *n* — код завершения, который требуется возвратить. Если же ничего не указано, то используется код завершения команды, выполнявшейся последней перед командой exit, что, по существу, равнозначно команде exit $?. Следует, однако, иметь в виду, что если команда exit выполняется непосредственно с терминала, то это фактически означает выход из системы и прерывание выполнения исходной оболочки.

Повторное рассмотрение программы rem

Команда exit нередко применяется обычным образом, чтобы прервать выполнение программы оболочки. В качестве примера рассмотрим еще раз программу rem, предназначенную для удаления записи из файла phonebook. В своем исходном виде программа rem может вызвать осложнения, если возникнут неожиданные ситуации, а возможно, нарушить или даже стереть полностью файл phonebook.

```
$ cat rem
#
# Удалить абонента из телефонного справочника
#

grep -v "$1" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
$
```

Допустим, что введена следующая команда:

```
rem Susan Topple
```

В этом случае оболочка передаст оба аргумента команде rem из-за отсутствия кавычек. И тогда программа rem удалит все записи с именем Susan, указанным в переменной $1, даже не обращая внимания на то, что пользователь указал слишком много аргументов.

Таким образом, в любой потенциально деструктивной программе целесообразно принять меры предосторожности, убедившись в том, что предполагаемое пользователем действие должно быть согласовано с действием, которое готова выполнить данная программа.

Одной из первых в программе rem может быть организована проверка правильности количества аргументов, как это уже было сделано ранее в программе on. Но на этот раз будет использована команда exit, чтобы прервать выполнение программы, если ей предоставлено неверное количество аргументов:

```
$ cat rem
#
# Удалить абонента из телефонного справочника - версия 2
#

if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments."
      echo "Usage: rem name"
      exit 1
fi

grep -v "$1" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
$ rem Susan Goldberg                    Опробовать программу
Incorrect number of arguments.
Usage: rem name
$
```

Команда `exit` возвращает код завершения **1**, чтобы известить о сбое на тот случай, если в какой-нибудь другой программе потребуется проверить его в условном выражении. Можно ли было бы написать приведенную выше программу с условными операторами `if-else` вместо применения команды `exit`?

Вам решать, пользоваться ли командной `exit` или условными операторами `if-else`. Иногда команда `exit` предоставляет более удобный и быстрый выход из программы, особенно если это делается на ранней стадии ее выполнения. И она обладает дополнительным преимуществом, исключая потребность прибегать к глубоко вложенным условным выражениям.

Конструкция `elif`

По мере усложнения программ приходится все чаще прибегать к написанию вложенных условных операторов `if` аналогично следующему:

```
if команда1
then
      команда
      команда
      ...
else
      if команда2
      then
            команда
            команда

      else
            if командаn
            then
                  команда
```

```
                команда

        else
                команда
                команда

        fi

    fi
fi
```

Такая последовательность команд может оказаться полезной, когда требуется сделать не двойной, а больший выбор. В таком случае организуется многовариантный выбор, причем последний условный оператор `else` выполняется, если не удовлетворяется ни одно из предыдущих условий.

В качестве относительно простого примера рассмотрим написание программы `greetings`, выводящей дружественное приветствие "`Good morning`" (Доброе утро), "`Good afternoon`"(Добрый день) или "`Good evening`" (Добрый вечер) в зависимости от времени суток. Ради простоты данного примера будем считать утром любой период времени от полуночи до полудня, днем — от полудня до 6 часов после полудня, а вечером — от 6 часов после полудня до полуночи.

Чтобы написать такую программу, придется выяснить текущее время суток. И для этой цели вполне подходит команда `date`. Ниже приведен результат ее выполнения.

```
$ date
Wed Aug 29 10:42:01 EDT 2002
$
```

Формат вывода результата выполнения команды `date` фиксирован, и этим обстоятельством можно выгодно воспользоваться, поскольку время всегда указывается на позициях символов от 12 до 19. Хотя в данном случае достаточно извлечь значение часа, указываемого на позициях 12 и 13, как показано ниже.

```
$ date | cut -c12-13
10
$
```

Таким образом, задача написания программы `greetings` существенно упрощается:

```
$ cat greetings
#
# Программа для вывода приветствия
#

hour=$(date | cut -c12-13)
```

```
if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
      echo "Good morning"
else
      if [ "$hour" -ge 12 -a "$hour" -le 17 ]
      then
            echo "Good afternoon"
      else
            echo "Good evening"
      fi
fi
$
```

Если значение переменной `$hour` больше или равно **0** (полночь) и меньше или равно **11** (до **11:59:59**), то выводится приветствие "Good morning". Если же значение переменной `hour` больше или равно **12** (полдень) и меньше или равно **17** (до **17:59:59**), то выводится приветствие "Good afternoon". А если ни одно из предыдущих условий не удовлетворяется, то выводится приветствие "Good evening".

```
$ greetings
Good morning
$
```

Если внимательно проанализировать программу `greetings`, то станет ясно, насколько неуклюже выглядит употребляемая в ней последовательность вложенных условных операторов `if-then-else`. Для упрощения подобного рода последовательностей в оболочке поддерживается также специальная конструкция `elif`, действующая аналогично конструкции `else if` *условие*, за исключением того, что она не повышает уровень вложенности. Ниже приведена общая форма этой конструкции.

```
if команда1
then
      команда
      команда
      ...
elif команда2
then
     команда
     команда

else
      команда
      команда
      ...
fi
```

Последовательность *команда*$_1$, *команда*$_2$ ... *команда*$_n$ выполняется по очереди, и при этом проверяются коды завершения этих команд. Как только одна из них возвратит нулевой код завершения (TRUE), следующие после нее команды выполняются вплоть до очередного условного оператора `elif`, `else` или `fi`. А если ни одно из условных выражений не оказывается истинным, то выполняются команды, следующие после дополнительного условного оператора `else`.

Используя новую конструкцию `elif`, программу `greetings` можно теперь переписать следующим образом:

```
$ cat greetings
#
# Программа для вывода приветствия - версия 2
#

hour=$(date | cut -c12-13)

if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
      echo "Good morning"
elif [ "$hour" -ge 12 -a "$hour" -le 17 ]
then
      echo "Good afternoon"
else
      echo "Good evening"
fi
$
```

Усовершенствование налицо. Теперь программу легче читать, а ее исходный текст не стремится больше выйти за правый край вследствие постепенно увеличивающихся отступов.

Между прочим, команды `date | cut` редко объединяются в конвейер, поскольку у самой команды `date` имеется богатый и разнообразный ряд форматов вывода результатов, которыми можно воспользоваться для получения требующейся информации или значения. Например, чтобы вывести текущий час в формате **0–23**, в команде `date` достаточно указать **`%H`** с префиксом **`+`**, необходимым для указания на то, что это форматирующая строка:

```
$ date +%H
10
$
```

В качестве упражнения внесите изменения в программу `greetings`, чтобы упростить подобным способом получение текущего часа суток.

Еще одна версия программы `rem`

Вернемся еще раз к программе `rem` для удаления абонентов из телефонного справочника. Как упоминалось ранее, пользоваться ей было небезопасно,

поскольку в отсутствие проверки правильности ее действий она могла слепо удалить больше записей, чем требовалось пользователю.

Чтобы устранить подобный недостаток, можно, в частности, организовать проверку *количества* записей, совпадающих с шаблоном, указанным пользователем, прежде чем удалять записи. Если обнаружено больше одной совпавшей записи, следует вывести соответствующее сообщение и прервать выполнение программы. Но как определить количество совпавших записей?

Для этого проще всего обработать файл phonebook по команде grep и подсчитать количество обнаруженных совпадений с помощью команды wc. Если совпадений окажется больше одного, то выводится соответствующее сообщение. Такую логику можно запрограммировать следующим образом:

```
$ cat rem
#
# Удалить абонента из телефонного справочника - версия 3
#

if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments."
      echo "Usage: rem name"
      exit 1
fi

name=$1

#
# Выявить количество совпавших записей
#

matches=$(grep "$name" phonebook | wc -l)

#
# Если совпавших записей больше одной, вывести сообщение,
# а иначе – удалить запись
#

if [ "$matches" -gt 1 ]
then
      echo "More than one match; please qualify further"
elif [ "$matches" -eq 1 ]
then
      grep -v "$name" phonebook > /tmp/phonebook
      mv /tmp/phonebook phonebook
else
      echo "I couldn't find $name in the phone book"
fi
$
```

Ради повышения удобочитаемости рассматриваемой здесь программы позиционный параметр $1 присваивается переменной name после проверки количества аргументов. Присваивание результата, выводимого последовательностью команд, весьма распространено в программах оболочки, как демонстрируется в следующей строке:

```
matches=$(grep "$name" phonebook | wc -l)
```

Как только количество совпавших записей будет подсчитано и сохранено в переменной $matches, нетрудно перейти к последовательности проверок в условных операторах if...elif...else, где сначала проверяется, является ли количество совпадений большим одного. Если совпадений больше одного, то выводится соответствующее сообщение. В противном случае проверяется, равно ли количество совпадений одному. Если оно равно, то совпавшая запись удаляется из телефонного справочника. А если количество совпадений не равно и не больше одного, то оно должно быть равно нулю, и тогда выводится сообщение, предупреждающее об этом пользователя.

Обратите внимание на то, что команда grep используется в данной программе дважды: сначала для того, чтобы определить количество совпадений, а затем вместе с параметром **-v**, чтобы удалить совпавшую запись, предварительно убедившись, что только она совпадает с заданным шаблоном. Ниже приведен ряд примеров применения команды rem в новой версии.

```
$ rem
Incorrect number of arguments.
Usage: rem name
$ rem Susan
More than one match; please qualify further
$ rem 'Susan Topple'
$ rem 'Susan Topple'
I couldn't find Susan Topple in the phone book      Ушла в небытие
$
```

Теперь программа rem действует довольно надежно. В частности, она проверяет правильное количество аргументов, выводит сообщение о надлежащем ее применении, если ей предоставлено неверное количество аргументов, а также обеспечивает удаление только одной записи из файла phonebook. По традиции, принятой в системах Unix, программа rem ничего не выводит при удачном исходе выполнения запрашиваемого действия.

Команда case

Команда case, реализующая одноименный оператор выбора, позволяет сравнивать единственное значение с рядом других значений или выражений и

выполнять одну или несколько команд при обнаружении совпадения. Ниже приведена общая форма этой команды.

```
case значение in
шаблон1) команда
         команда
         ...
         команда;;
шаблон2) команда
         команда
         ...
         команда;;
...
шаблонn) команда
         команда
         ...
         команда;;
esac
```

Заданное *значение* последовательно сравнивается со значениями $шаблон_1$, $шаблон_2$ и $шаблон_n$ до тех пор, пока не будет обнаружено совпадение. Как только совпадение обнаруживается, выполняются команды, следующие после совпавшего значения и вплоть до двух знаков точки с запятой, которые служат в качестве оператора прерывания, обозначающего завершение последовательности команд, выполняемых по данному конкретному условию. Как только будут достигнуты два знака точки с запятой, выполнение оператора `case` завершается. Если же совпадение не обнаруживается, то не выполняется ни одна из команд, перечисленных в операторе `case`.

В качестве примера применения команды, реализующей оператор `case`, в следующей программе под названием `number` берется единственная цифра, которая преобразуется в ее название на английском языке:

```
$ cat number
#
# Преобразовать цифру в ее название на английском языке
#

if [ "$#" -ne 1 ]
then
      echo "Usage: number digit"
      exit 1
fi

case "$1"
in
       0) echo zero;;
       1) echo one;;
       2) echo two;;
```

```
        3) echo three;;
        4) echo four;;
        5) echo five;;
        6) echo six;;
        7) echo seven;;
        8) echo eight;;
        9) echo nine;;
esac
$
```

А теперь проверим эту программу:

```
$ number 0
zero
$ number 3
three
$ number                    Попробовать выполнить без аргументов
Usage: number digit
$ number 17                 Попробовать выполнить с числом из двух цифр
$
```

В последнем случае наглядно показано, что произойдет, если ввести больше одной цифры. В этом случае значение позиционного параметра `$1` не совпадает ни с одним из значений, перечисленных в операторе `case`, а следовательно, ни одна из указанных в нем команд `echo` не будет выполнена.

Специальные символы совпадения с шаблоном

Оператор `case` довольно эффективен, поскольку он позволяет создавать сложные регулярные выражения в обозначении оболочки вместо того, чтобы просто указывать последовательности букв. Это означает, что одни и те же специальные символы могут быть использованы для обозначения шаблонов в операторе `case` таким же образом, как и при подстановке имен файлов. Например, знак **?** может быть использован для обозначения любого одиночного символа, знак ***** — для обозначения нулевого и большего количества совпадений с любым символом, а знаки **[...]** — для обозначения любого одиночного символа, заключенного в квадратные скобки.

Шаблон ***** обозначает совпадение с *любыми* символами (аналогично совпадению со всеми файлами в текущем каталоге по команде `echo *`), и поэтому он нередко применяется в конце оператора `case` в качестве "универсального" или значения по умолчанию, которое гарантирует совпадение, когда ни одно из предыдущих значений в операторе `case` не совпадает. Принимая во внимание это обстоятельство, ниже приведена вторая версия программы `number`, где применяется такой универсальный оператор `case`.

```
$ cat number
#
# Преобразовать цифру в ее название на английском языке - версия 2
#
```

```
if [ "$#" -ne 1 ]
then
      echo "Usage: number digit"
      exit 1
fi

case "$1"
in
        0) echo zero;;
        1) echo one;;
        2) echo two;;
        3) echo three;;
        4) echo four;;
        5) echo five;;
        6) echo six;;
        7) echo seven;;
        8) echo eight;;
        9) echo nine;;
        *) echo "Bad argument; please specify a single digit";;
esac
$ number 9
nine
$ number 99
Bad argument; please specify a single digit
$
```

Перейдем к примеру другой программы, называемой `ctype`, выявляющей и выводящей класс одиночного символа, задаваемого в качестве ее аргумента. К типам распознаваемых символов относятся цифры, прописные и строчные буквы, а также специальные символы (т.е. любые символы, не относящиеся к двум первым категориям). В качестве дополнительной меры в данной программе проверяется, что ей передан только один символ как аргумент:

```
$ cat ctype
#
# Классифицировать символ, заданный в качестве аргумента
#

if [ $# -ne 1 ]
then
      echo Usage: ctype char
      exit 1
fi

#
# Проверить, был ли набран только один символ
#

char="$1"
numchars=$(echo "$char" | wc -c)
```

```
if [ "$numchars" -ne 1 ]
then
      echo Please type a single character
      exit 1
fi

#
# А теперь классифицировать этот символ
#

case "$char"
in
        [0-9]     ) echo digit;;
        [a-z]     ) echo lowercase letter;;
        [A-Z]     ) echo uppercase letter;;
        *         ) echo special character;;
esac
$
```

Но если попытаться выполнить данную программу с разными аргументами, то обнаружится, что она действует неверно:

```
$ ctype a
Please type a single character
$ ctype 7
Please type a single character
$
```

Параметр -x для отладки программ

Программные ошибки не столь уж редки в процессе разработки программ, состоят ли они из 5 или 500 строк. В рассматриваемом здесь примере программы оказывается, что та ее часть, которая отвечает за подсчет букв, действует неверно. Но как выяснить, что именно действует неверно?

С этой целью целесообразно внедрить в оболочке параметр **-x**. Чтобы отладить любую программу оболочки или хотя бы выяснить, каким образом она действует, следует выполнить трассировку последовательности ее выполнения, введя команду `sh -x` и вслед за ней имя отлаживаемой программы и ее аргументы. Таким образом, указанная программа начинает выполняться в новой оболочке с активизированным параметром **-x**.

В этом режиме команды выводятся на терминал по мере их выполнения, предваряемые знаком "плюс". Итак, попробуем отладить программу `ctype`!

```
$ sh -x ctype              Выполнение трассировки
+ [ 1 -ne 1 ]              $# равно 1
+ char=a                   Присваивание позиционного
                           параметра $1 переменной char
+ echo a
+ wc -c
```

```
+ numchars= 2              команда wc возвращает значение 2!
+ [ 2 -ne 1 ]              Именно поэтому данная проверка проходит
+ echo please type a single character
please type a single character
+ exit 1
$
```

Как показывает трассировка, команда wc возвращает значение **2** в результате своего выполнения:

```
echo "$char" | wc -c
```

Но почему это значение **2**, если команде wc была передана буква **a**? Оказывается, что данной команде фактически были переданы два символа: одна буква **a** и "невидимый" знак начала строки, который команда echo автоматически выводит в конце каждой строки. Таким образом, вместо проверки наличия единственного символа путем его сравнения со значением **1** в условных выражениях следует сравнивать введенные пользователем данные со значением **2**, которое обозначает набранный символ плюс знак новой строки, добавленный командой echo.

Возвращаясь к программе ctype, обновим в ней условный оператор if:

```
if [ "$numchars" -ne 1 ]
then
      echo Please type a single character
      exit 1
fi
```

следующим образом:

```
if [ "$numchars" -ne 2 ]
then
      echo Please type a single character
      exit 1
fi
```

а затем попробуем выполнить ее снова.

```
$ ctype a
lowercase letter
$ ctype abc
Please type a single character
$ ctype 9
digit
$ ctype K
uppercase letter
$ ctype :
special character
$ ctype
Usage: ctype char
$
```

Похоже, что теперь рассматриваемая здесь программа действует правильно. Как поясняется в главе 11, режим трассировки можно включать и выключать по желанию и в *самой* программе. А до тех пор рекомендуем пользоваться командой sh -x в различных созданных до сих пор сценариях и программах оболочки.

Прежде чем оставить пример программы ctype, приведем ее версию, в которой команда wc не применяется, а все возможные условия обрабатываются в ветвях выбора одного оператора case:

```
$ cat ctype
#
# Классифицировать символ, заданный в качестве
# аргумента -- версия 2
#

if [ $# -ne 1 ]
then
      echo Usage: ctype char
      exit 1
fi

#
# Проверить, был ли набран только один символ
#

char=$1

case "$char"
in
        [0-9]   ) echo digit;;
        [a-z]   ) echo lowercase letter;;
        [A-Z]   ) echo uppercase letter;;
        ?       ) echo special character;;
        *       ) echo Please type a single character;;
esac
$
```

Напомним, что знак ? обозначает совпадение с любым одиночным символом. А поскольку мы уже проверили символы на предмет их принадлежности цифрам, прописным и строчным буквам, то при совпадении с данным шаблоном символ должен быть специальным. И наконец, если совпадение с данным шаблоном *не* происходит, это означает, что было введено больше одного символа, о чем выводится соответствующее сообщение:

```
$ ctype u
lowercase letter
$ ctype '>'
special character
$ ctype xx
Please type a single character
$
```

Возвращение к команде case

Когда знак | указывается между двумя шаблонами, он обозначает логическую операцию ИЛИ. Это означает, что в следующем выражении:

шаблон$_i$ | *шаблон*$_j$

определяется совпадение с любым из двух указанных шаблонов. Например, в выражении

```
-l | -list
```

определяется совпадение со значением **-l** или **-list**, а в выражении

```
dmd | 5620 | tty5620
```

совпадение со значением **dmd**, **5620** или **tty5620**. Принимая во внимание это обстоятельство, можно более точно и эффективно переписать самые разные управляющие структуры в ветвях выбора из оператора case.

Например, рассмотренную ранее программу greetings можно переписать, воспользовавшись оператором case вместо более громоздкой конструкции if-elif. На этот раз выгодно используется тот факт, что команда date с оператором **+%H** выводит значение часа в стандартном формате из двух цифр, как показано ниже.

```
$ cat greetings
#
# Программа для вывода приветствия - версия с оператором case
#

hour=$(date +%H)

case "$hour"
in
        0? | 1[01] ) echo "Good morning";;
        1[2-7]     ) echo "Good afternoon";;
        *          ) echo "Good evening";;
esac
$
```

Значение часа в формате из двух цифр, получаемое из команды date, сначала присваивается переменной оболочки hour, а затем выполняется оператор case. Далее значение переменной hour сопоставляется с первым шаблоном:

```
0? | 1[01]
```

С этим шаблоном совпадает любое значение, начинающееся с нуля, а затем символ любой цифры (т.е. от полуночи до 9 часов утра) или же любое значение, начинающееся с единицы, а далее нуль или единица (т.е. 10 или 11 часов утра). А со вторым шаблоном:

```
1[2-7]
```

совпадает значение, начинающееся с единицы, а далее символ любой цифры от двух до семи (т.е. от полудня до 5 часов вечера).

И в последней ветви выбора из оператора case проверяется совпадение с универсальным шаблоном, т.е. со всеми остальными цифрами (от 6 до 11 часов вечера). Ниже приведен пример выполнения программы greetings в новой версии с оператором case.

```
$ date
Wed Aug 28 15:45:12 EDT 2002
$ greetings
Good afternoon
$
```

Пустая команда :

В каждой совпадающей ветви выбора из оператора case и в каждом условном операторе if-then требуется результирующая команда. Но иногда возникает потребность вообще ничего не выполнять по заданному условию. Как же это сделать? С помощью встроенной в оболочку *пустой* команды, имеющей очень простую форму :. Нетрудно догадаться, что эта команда предназначена именно для того, чтобы вообще ничего не делать.

Как правило, пустая команда служит для того, чтобы удовлетворить требованию о наличии команды по заданному условию, особенно в условных операторах if. Допустим, требуется убедиться в том, что значение, хранящееся в переменной system, присутствует в файле /users/steve/mail/system, а если оно отсутствует — вывести сообщение об ошибке и выйти из программы. С этой целью можно было бы написать следующий условный оператор:

```
if grep "^$system" /users/steve/mail/systems > /dev/null
then
```

Но непонятно, что же указывать после условного оператора then. Ведь в данном случае требуется проверить отсутствие значения переменной $system в файле и ничего особенного не делать при удачном исходе выполнения команды grep. Оболочка требует, чтобы после условного оператора then была *непременно* указана команда. Именно здесь и приходит на помощь пустая команда:

```
if grep "^$system" /users/steve/mail/systems > /dev/null
then

else
        echo "$system is not a valid system"
        exit 1
fi
```

Если проверка условия проходит, то ничего делается. А если данная проверка не проходит, то выводится сообщение об ошибке и происходит выход из программы.

Откровенно говоря, данный конкретный пример можно было бы переделать, изменив на обратную структуру команды `grep` (например, воспользовавшись аргументом `!` команды `test`). Но иногда проще организовать проверку положительного, а не отрицательного условия, ничего при этом не делая. Но в любом случае очень полезным оказывается своевременный комментарий, который буквально творит чудеса, когда приходится читать исходный текст программы многие недели, а то и месяцы спустя.

Конструкции && и ||

В оболочке имеются две специальные конструкции, позволяющие выполнять последующую команду в зависимости от удачного или неудачного исхода выполнения предыдущей команды. На самом деле эти конструкции подобны условному оператору `if` и просто являются более краткой его формой. Так, если написать следующее выражение:

команда$_1$ && *команда*$_2$

в любом месте, где в оболочке предполагается обнаружить команду, то выполняется *команда*$_1$. Если эта команда возвращает (при удачном исходе) нулевой код завершения, выполняется *команда*$_2$. Если же *команда*$_1$ возвращает (при неудачном исходе) ненулевой код завершения, то *команда*$_2$ *не* выполняется и просто игнорируется.

Так, если написать следующее условное выражение:

```
sort bigdata > /tmp/sortout && mv /tmp/sortout bigdata
```

то команда `mv` будет выполнена только при удачном исходе выполнения команды `sort`. Следует, впрочем, заметить, что данное выражение равнозначно приведенному ниже условному оператору.

```
if sort bigdata > /tmp/sortout
then
      mv /tmp/sortout bigdata
fi
```

А в следующей команде:

```
[ -z "$EDITOR" ] && EDITOR=/bin/ed
```

проверяется значение переменной `$EDITOR`. Если в этой переменной содержится пустое значение, то ей присваивается значение **`/bin/ed`**.

Аналогичным образом действует конструкция **||**, за исключением того, что команда, указываемая в ней второй, выполняется лишь в том случае, если первая команда возвратит ненулевой код завершения. Так, если написать следующее условное выражение:

```
grep "$name" phonebook || echo "Couldn't find $name"
```

то команда echo будет выполнена лишь при неудачном исходе выполнения команды grep, т.е. в том случае, если значение переменной $name не удастся найти в файле phonebook. И это выражение равнозначно приведенному ниже условному оператору.

```
if grep "$name" phonebook
then

else
      echo "Couldn't find $name"
fi
```

В левой или правой части этих конструкций можно указать сложные последовательности команд. В левой части этих конструкций проверяется код завершения последней команды в конвейере. Следовательно, в приведенном ниже условном выражении команда echo выполняется лишь при неудачном исходе выполнения команды grep.

```
who | grep "^$name " > /dev/null || echo "$name's not logged on"
```

Конструкции **&&** и **||** можно также объединять в одной командной строке, как демонстрируется в следующем примере:

```
who | grep "^$name " > /dev/null && echo "$name's not logged on" \
    || echo "$name is logged on"
```

(Напомним, что наличие знака **** в конце строки сигнализирует оболочке о продолжении строки.) Первая команда echo выполняется при удачном исходе выполнения команды grep, а вторая — при неудачном.

Эти конструкции нередко применяются и в условных операторах if, как демонстрируется в следующем фрагменте кода из программы системной оболочки (в этом фрагменте обратите больше внимания на применение конструкции **&&**, чем на вызываемые команды):

```
if validsys "$sys" && timeok
then
      sendmail "$user@$sys" < $message
fi
```

Если команда validsys возвращает нулевой код завершения, выполняется команда timeok и далее проверяется код ее завершения в условном операторе if. Если этот код завершения оказывается нулевым, то выполняется команда sendmail. А если команда validsys возвращает ненулевой код завершения, то команда timeok *не* выполняется, далее в условном операторе if проверяется код неудачного завершения и в конечном итоге команда sendmail также не выполняется.

Конструкция **&&** в предыдущем примере обозначает логическую операцию И. Обе команды, указанные в условном операторе if, должны возвратить нулевой код завершения, чтобы была выполнена и команда sendmail. На самом деле приведенный выше фрагмент кода можно было бы написать следующим образом:

```
validsys "$sys" && timeok && sendmail "$user@$sys" < $message
```

С другой стороны, наличие конструкции **||** в условном операторе if обозначает логическую операцию ИЛИ, как показано ниже.

```
if endofmonth || specialrequest
then
      sendreports
fi
```

Если команда endofmonth возвращает нулевой код завершения, то выполняется команда specialrequest. А если она возвращает ненулевой код завершения, то выполняется команда sendreports. Таким образом, команда sendreports выполняется, если нулевой код завершения возвращает команда endofmonth или specialrequest.

Из главы 8 вы узнаете, как организовать сложные циклы, управляющие ходом выполнения программы. Но прежде поупражняйтесь в применении операторов if, case и конструкций **&&** и **||**, пояснявшихся в этой главе.

8

Повторное выполнение команд

В этой главе речь пойдет об организации программных циклов, позволяющих выполнять ряд команд указанное количество раз до тех пор, пока не будет удовлетворено конкретное конечное условие. Для организации программных циклов в оболочке предоставляются перечисленные ниже команды. Они рассматриваются по отдельности в соответствующих разделах этой главы.

- `for`
- `while`
- `until`

Команда `for`

Команда `for` служит для выполнения одной или нескольких команд указанное количество раз. Ниже приведена общая форма этой команды, реализующей оператор цикла `for`.

```
for var in слово1 слово2 ... словоn
do
        команда
        команда

done
```

Команды, указанные в промежутке между операторами `do` и `done`, называются *телом* цикла и выполняются столько раз, сколько указано после оператора `in`. В начале цикла первое *слово*$_1$ присваивается переменной цикла `var` и далее выполняется тело цикла. Затем переменной цикла `var` присваивается второе *слово*$_2$ и тело цикла выполняется снова.

Этот процесс продолжается до тех пор, пока переменной не будет присвоено последнее *слово*$_n$, а тело цикла — выполнено в последний раз. После этого слов в списке больше не остается, и на этом цикл `for` завершается. Вслед за этим сразу же выполняется оператор `done`.

Если в операторе `in` перечислено n слов, тело цикла будет выполнено в общем n раз, после чего цикл завершится. Ниже приведен пример цикла, выполняющегося три раза.

```
for i in 1 2 3
do
      echo $i
done
```

Чтобы опробовать этот цикл, наберите его непосредственно на терминале аналогично любой другой команде оболочки, как показано ниже.

```
$ for i in 1 2 3
> do
>      echo $i
> done
1
2
3
$
```

В ожидании ввода оператора done для завершения цикла for оболочка продолжает выводить вспомогательное приглашение на ввод команд. Как только пользователь введет оператор done, оболочка продолжит выполнение цикла. В данном примере после оператора in указаны три элемента (значения **1**, **2** и **3**), и поэтому тело цикла (в данном случае — единственная команда echo) будет выполнено в общем три раза.

Сначала переменной цикла i присваивается первое значение **1** из списка. Затем тело цикла выполняется первый раз. При этом значение переменной цикла i выводится на терминал. Затем переменной цикла i присваивается второе значение **2**, и команда echo выполняется снова, выводя это значение на терминал. Далее переменной цикла i присваивается третье значение **3**, и команда echo выполняется в третий раз, выводя это значение на терминал. В этот момент в списке больше не остается слов, поэтому выполнение цикла for завершается. После этого оболочка выводит приглашение на ввод команд, уведомляя о том, что цикл завершен.

А теперь вернемся к примеру программы run, рассматривавшейся в главе 6 и позволявшей обработать файл в последовательности команд tbl, nroff и lp:

```
$ cat run
tbl $1 | nroff -mm -Tlp | lp
$
```

Если же требуется обработать несколько файлов (например, memo1–memo4), то на терминале можно набрать следующий цикл:

```
$ for file in memo1 memo2 memo3 memo4
> do
>       run $file
> done
request id is laser1-33 (standard input)
request id is laser1-34 (standard input)
request id is laser1-35 (standard input)
request id is laser1-36 (standard input)
$
```

Переменной цикла `file` по очереди присваиваются значения **`memo1`**, **`memo2`**, **`memo3`** и **`memo4`**, и программа `run` выполняется со значением этой переменной в качестве аргумента. Выполнение данного цикла будет осуществлено так, как будто были введены четыре команды подряд:

```
$ run memo1
request id is laser1-33 (standard input)
$ run memo2
request id is laser1-34 (standard input)
$ run memo3
request id is laser1-35 (standard input)
$ run memo4
request id is laser1-36 (standard input)
$
```

В оболочке допускается подставлять имена файлов из списка слов, указанного в операторе `for`. Таким образом, предыдущий цикл можно переписать следующим образом:

```
for file in memo[1-4]
do
      run $file
done
```

Если по команде `run` требуется вывести все файлы из текущего каталога, для этого достаточно набрать следующий цикл:

```
for file in *
do
      run $file
done
```

Чтобы выполнить более сложное действие, допустим, в файле `filelist` содержится список файлов, которые требуется обработать по команде `run`, достаточно ввести следующий цикл:

```
files=$(cat filelist)
for file in $files
do
      run $file
done
```

или более лаконично:

```
for file in $(cat filelist)
do
      run $file
done
```

В качестве дальнейшего совершенствования можно внести соответствующие коррективы в саму программу run, и оператор for идеально подходит для данной цели, как показано ниже.

```
$ cat run
#
# Обработать файлы по команде nroff -- версия 2
#

for file in $*
do
      tbl $file | nroff -rom -Tlp | lp
done
$
```

Напомним, что специальная переменная оболочки $* обозначает *все* аргументы, введенные в командной строке. Если выполнить новую версию программы run, введя в командной строке следующее:

```
run memo1 memo2 memo3 memo4
```

то список, сохраняемый в переменной оболочки $* для цикла for, будет заменен введенными аргументами memo1, memo2, memo3 и memo4. Разумеется, для получения тех же самых результатов можно было бы ввести следующее:

```
run memo[1-4]
```

Переменная $@

Раз уж мы коснулись переменной $*, рассмотрим более подробно, каким образом действует она и родственная ей переменная $@. С этой целью рассмотрим следующую программу args, отображающую в отдельных строках все аргументы, введенные в командной строке:

```
$ cat args
echo Number of arguments passed is $#

for arg in $*
do
      echo $arg
done
$
```

А теперь опробуем эту программу следующим образом:

```
$ args a b c
Number of arguments passed is 3
a
b
c
```

```
$ args 'a b' c
Number of arguments passed is 2
a
b
c
$
```

Проанализируем более подробно второй из приведенных выше примеров. Несмотря на то что строка 'a b' была передана программе args как единый аргумент, она все равно разбивается на два аргумента в цикле for. Именно поэтому переменная $* заменяется в операторе цикла for списком значений a b c, а кавычки при этом отбрасываются. Таким образом, цикл for выполняется три раза.

Оболочка заменяет значение переменной $* значениями аргументов $1, $2 и т.д., но если вместо нее воспользоваться специальной переменой оболочки "$@", то программе будут переданы значения "$1", "$2" и т.д. Главное отличие состоит в том, что переменная $@ заключается в двойные кавычки, в отсутствие которых она действует таким же образом, как и переменная $*. Вернемся к программе args и заменим переменную $* без кавычек переменной "$@" в кавычках:

```
$ cat args
echo Number of arguments passed is $#
for arg in "$@"
do
      echo $arg
done
$
```

А теперь опробуем эту программу еще раз:

```
$ args a b c
Number of arguments passed is 3
a
b
c
$ args 'a b' c
Number of arguments passed is 2
a b
c
$ args                    Опробовать программу без аргументов
Number of arguments passed is 0
$
```

В последнем случае программе не было передано ни одного аргумента, и поэтому переменную "$@" *нечем* было заменить. А в конечном счете тело цикла так и не было выполнено.

Цикл for без списка

При организации циклов for оболочка распознает особое обозначение. Так, если опустить оператор in и следующий после его список, как показано ниже, оболочка воспримет последовательно все аргументы, введенные в командной строке.

```
for var
do
      команда
      команда

done
```

Это все равно, что составить следующий цикл:

```
for var in "$@"
do
      команда
      команда

done
```

Если принять во внимание эту особенность организации циклов for, то третью и последнюю версию программы args можно написать и опробовать следующим образом:

```
$ cat args
echo Number of arguments passed is $#
for arg
do
      echo $arg
done
$ args a b c
Number of arguments passed is 3
a
b
c
$ args 'a b' c
Number of arguments passed is 2
a b
c
$
```

Команда while

Второй для организации циклов является команда, реализующая оператор while. Ниже приведена его общая форма.

```
while команда1
do
        команда2
        команда3

done
```

В цикле while выполняется *команда*$_1$ и проверяется код ее завершения. Если этот код нулевой, то выполняются команды, находящиеся в промежутке между операторами do и done и образующие тело данного цикла. Затем *команда*$_1$ выполняется снова и опять проверяется код ее завершения.

Если этот код нулевой, то тело цикла while выполняется снова. Этот процесс продолжается до тех пор, пока *команда*$_1$ не возвратит ненулевой код своего завершения, и тогда данный цикл завершится. А выполнение продолжится с команды, следующей после оператора done.

Следует, однако, иметь в виду, что команды, находящиеся в теле цикла while, т.е. в промежутке между операторами do и done, могут быть так и не выполнены, если *команда*$_1$ возвратит ненулевой код своего завершения после первого же выполнения. Ниже приведена программа twhile, где производится простой подсчет до **5**.

```
$ cat twhile
i=1
while [ "$i" -le 5 ]
do
        echo $i
        i=$((i + 1))
done
$ twhile                Выполнить эту программу
1
2
3
4
5
$
```

В переменной цикла $i первоначально устанавливается значение **1**. Затем начинается цикл while, где при удачном исходе проверки выполняется блок кода, образующий тело данного цикла. Оболочка продолжает выполнение цикла while до тех пор, пока значение переменной цикла $i будет меньше или равно **5**. В самом теле цикла значение его переменной $i выводится на терминал и увеличивается на единицу.

Цикл while нередко применяется в сочетании с командой shift для обработки неизвестного количества аргументов, введенных в командной строке. Рассмотрим в качестве примера следующую программу, называемую prargs и выводящую каждый аргумент в отдельной строке:

```
$ cat prargs
#
# Вывести аргументы командной строки в отдельных строках
#

while [ "$#" -ne 0 ]
do
      echo "$1"
      shift
done
$ prargs a b c
a
b
c
$ prargs 'a b' c
a b
c
$ prargs *
addresses
intro
lotsaspaces
names
nu
numbers
phonebook
stat
$ prargs                    Без аргументов
$
```

До тех пор, пока количество аргументов не равно нулю, значение переменной оболочки `$1` выводится на терминал, а затем вызывается команда `shift`, сдвигающая значения переменных оболочки (т.е. значение переменной `$2` — в переменную `$1`, значение переменной `$3` — в переменную `$2` и т.д.), а также уменьшающая на единицу значение переменной `$#`. Как только будет выведен и сдвинут последний аргумент, значение переменной `$#` станет равным нулю, и в этот момент цикл `while` завершится. Следует, однако, иметь в виду, что если не предоставить программе `prargs` никаких аргументов, то команды `echo` и `shift` так и не будут выполнены в теле цикла `while`, поскольку в самом его начале значение переменной `$#` равно нулю.

Команда `until`

Цикл `while` выполняется до тех пор, пока проверочное выражение продолжает возвращать нулевой код завершения (`TRUE`). Совсем иначе действует цикл `until`. Он продолжает выполнять блок кода в своем теле до тех пор, пока проверочное выражение возвращает *ненулевой* код завершения, и прекратит выполнять этот блок кода, как только возвратится нулевой код завершения. Ниже приведена

общая форма команды, реализующей цикл until. Как и в цикле while, команды, находящиеся в теле цикла until, т.е. в промежутке между операторами do и done, могут быть так и не выполнены, если *команда*$_1$ возвратит нулевой код своего завершения при первом же выполнении.

```
until команда1
do
      команда
      команда

done
```

Несмотря на внешнее сходство обоих циклов, команда, реализующая цикл until, чаще всего применяется для написания программ, ожидающих наступления определенного события. Допустим, требуется выяснить, зарегистрировался ли пользователь sandy в системе, поскольку ему необходимо передать нечто важное. С этой целью данному пользователю можно было бы послать сообщение по электронной почте, но известно, что он привык читать свою почту в конце дня. Чтобы выяснить, зарегистрировался ли пользователь sandy в системе, можно воспользоваться программой on, упоминавшейся в главе 6:

```
$ on sandy
sandy is not logged on
$
```

Это простой, но неэффективный способ, поскольку данная программа выполняется только раз. Ее можно было бы, конечно, выполнять периодически в течение дня, но это не очень удобно. Поэтому лучше написать отдельную программу для постоянной проверки регистрации нужного пользователя в системе!

Допустим, такая программа называется waitfor и принимает в качестве единственного аргумента имя пользователя, вход которого в систему требуется контролировать. Но вместо того чтобы непрерывно проверять, зарегистрировался ли нужный пользователь в системе, это можно делать лишь один раз в минуту. И для этого достаточно воспользоваться командой sleep, приостанавливающей выполнение программы на указанное количество секунд. Так, следующая команда:

```
sleep n
```

приостанавливает выполнение программы на *n* секунд. А по истечении заданного интервала времени программа возобновит свое выполнение с того места, где она была приостановлена, т.е. с команды, следующей сразу же после команды sleep:

```
$ cat waitfor
#
# Ожидать до тех пор, пока указанный пользователь
# не зарегистрируется в системе
#
```

```
if [ "$#" -ne 1 ]
then
      echo "Usage: waitfor user"
      exit 1
fi
user="$1"
#
# Проверять каждые 60 секунд, зарегистрировался ли
# указанный пользователь в системе
#

until who | grep "^$user " > /dev/null
do
      sleep 60
done

#
# Если программа дойдет до этого места, значит,
# указанный пользователь зарегистрировался в системе
#

echo "$user has logged on"
$
```

После того, как будет проверено, что программе предоставлен только один аргумент, значение переменной `$1`, хранящей этот аргумент, присваивается переменной `user`. Затем начинается цикл `until`, который будет выполняться до тех пор, пока команда `grep` не возвратит нулевой код завершения, т.е. до того момента, когда указанный пользователь зарегистрируется в системе. А до тех пор, пока он не войдет в систему, тело цикла `until` (т.е. команда `sleep`) будет продолжаться, приостанавливая всякий раз выполнение программы на 60 секунд. По истечении заданного промежутка времени в одну минуту снова вызывается конвейер команд, указанный после команды `until`, и весь процесс повторяется снова.

По окончании цикла `until`, свидетельствующего о том, что нужный пользователь зарегистрировался в системе, на терминал выводится соответствующее сообщение. И на этом данная программа завершается, как показано ниже.

```
$ waitfor sandy          Проходит время
sandy has logged on
$
```

Разумеется, выполнять программу `waitfor` так, как показано выше, не очень практично, поскольку она связывает терминал до тех пор, пока пользователь `sandy` не войдет в систему. Поэтому данную программу лучше выполнять в фоновом режиме, чтобы пользоваться терминалом для других видов работ. С этой целью программу `waitfor` можно запустить на выполнение следующим образом:

```
$ waitfor sandy &          Выполнить программу в фоновом режиме
[1] 4392                   Номер задания и идентификатор процесса
$ nroff                    Выполнить другую работу
    ...
sandy has logged on        Это событие происходит некоторое время спустя
```

Теперь можно выполнить другую работу. А программа `waitfor` продолжит свое выполнение в фоновом режиме до тех пор, пока пользователь `sandy` не войдет в систему или текущий пользователь не выйдет из нее.

Примечание

По умолчанию все процессы автоматически завершаются, когда вы выходите из системы. Если же требуется, чтобы программа продолжила свое выполнение после того, как вы выйдете из системы, можете запустить ее с помощью команды **nohup** или запланировать ее выполнение по команде **at** или **cron**. За дополнительной справкой обращайтесь к руководству пользователя вашей системы Unix.

Программа `waitfor` проверяет факт регистрации нужного пользователя в системе лишь один раз в минуту, и поэтому она не потребляет много системных ресурсов на свое выполнение. Это соображение очень важно принимать во внимание, выполняя программы в фоновом режиме.

К сожалению, после того, как указанный пользователь войдет в систему, вполне возможно, что соответствующее уведомление об этом в одной строке будет пропущено. Так, если вы редактируете файл в экранном редакторе вроде `vi`, сообщение, извещающее о регистрации нужного пользователя в системе, может превратить изображение на экране в полный беспорядок, и прочитать это сообщение так и не удастся!

Поэтому сообщение, извещающее о регистрации нужного пользователя в системе, лучше отправить самому себе по электронной почте. А еще лучше предоставить пользователю программы `waitfor` возможность выбрать вариант отправки извещающего сообщения по электронной почте. Если выбран именно такой вариант, программа должна указывать, что сообщение будет отправлено по электронной почте, а иначе — выведено на терминал. С этой целью в следующей версии программы `waitfor` добавлен параметр **-m**:

```
$ cat waitfor
#
# Ожидать до тех пор, пока указанный пользователь
# не зарегистрируется в системе - версия 2
#

if [ "$1" = -m ]
then
      mailopt=TRUE
      shift
else
      mailopt=FALSE
fi
```

```
if [ "$#" -eq 0 -o "$#" -gt 1 ]
then
      echo "Usage: waitfor [-m] user"
      echo" -m means to be informed by mail"
      exit 1
fi

user="$1"
#
# Проверять каждую минуту, зарегистрировался ли
# указанный пользователь в системе
#

until who | grep "^$user " > /dev/null
do
      sleep 60
done

#
# Если программа дойдет до этого места, значит,
# указанный пользователь зарегистрировался в системе
#

if [ "$mailopt" = FALSE ]
then
      echo "$user has logged on"
else
      echo "$user has logged on" | mail steve
fi
$
```

При первой проверке в данной программе выясняется, был ли указан параметр **-m**. Если он был указан, переменной `mailopt` присваивается логическое значение `TRUE`, а затем выполняется команда `shift`, выполняющая сдвиг первого аргумента с именем контролируемого пользователя в переменную `$1` и уменьшающая на единицу значение переменной `$#`. Если же параметр **-m** не был указан, переменной `mailopt` присваивается логическое значение `FALSE`.

Выполнение программы продолжается далее таким же образом, как и в предыдущей ее версии, за исключением того, что на этот раз по завершении основного блока кода проверяется значение переменной `mailopt`, чтобы выяснить, куда именно направлять извещающее сообщение: по электронной почте или на терминал по команде `echo`. Ниже демонстрируется применение программы `waitfor` в новой версии.

```
$ waitfor sandy -m
Usage: waitfor [-m] user
       -m means to be informed by mail
$ waitfor -m sandy &
[1] 5435
```

```
$ vi newmemo                    Продолжить работу
    ...
you have mail
$ mail
From steve Wed Aug 28 17:44:46 EDT 2002
sandy has logged on

?d
$
```

Разумеется, программу `waitfor` можно было бы написать таким образом, чтобы она принимала параметр **-m** в качестве первого или второго аргумента. Но в традиционном синтаксисе Unix все параметры должны *непременно* предшествовать любым другим типам аргументов в командной строке.

Следует также иметь в виду, что в прежней версии программу `waitfor` можно было бы выполнить следующим образом:

```
$ waitfor sandy | mail steve &
[1] 5522
$
```

чтобы добиться такого же результата, как и с помощью дополнительного параметра **-m**, хотя это и менее изящный вариант.

В текущей версии данная программа всегда посылает извещающее сообщение по электронной почте адресату `steve`, что не особенно практично для какого-нибудь другого ее пользователя. Поэтому лучше определить сначала пользователя, выполняющего данную программу, а затем отправить ему извещающее сообщение по электронной почте, если он укажет параметр **-m**. Но как это сделать? С этой целью можно, например, выполнить сначала команду `who` с параметрами **am i** и получить в ответ имя текущего пользователя, а затем извлечь его по команде `cut` из результата, выводимого командой `who`, чтобы воспользоваться им в качестве имени получателя электронной почты.

Все это можно сделать в последнем условном операторе `if`, внеся приведенные ниже изменения. Теперь программу `waitfor` сможет выполнить любой пользователь, и он получит извещающее сообщение по своему правильно определенному имени адресата электронной почты.

```
if [ "$#" -eq 1 ]
then
      echo "$user has logged on"
else
      recipient =$(who am i | cut -c1-8)
      echo "$user has logged on" | mail $recipient
fi
```

Дополнительные сведения о циклах

В последующих разделах рассматриваются дополнительные возможности организации циклов.

Прерывание цикла

Иногда логика выполнения программы диктует немедленный выход из цикла. Чтобы выйти из цикла, но не из самой программы, достаточно указать в теле цикла следующую команду:

```
break
```

Когда выполняется команда `break`, управление немедленно передается за пределы цикла, где выполнение программы продолжается обычным образом. Таким способом можно, в частности, воспользоваться для прерывания бесконечного цикла, т.е. такого блока кода, который предназначен для бесконечного выполнения одних и тех же команд до тех пор, пока оно не будет прервано командой `break`.

В подобных случаях можно воспользоваться командой `true`, чтобы возвратить нулевой код завершения, и командой `false` — в противоположной ситуации, чтобы возвратить ненулевой код завершения. Так, если написать следующий цикл `while`:

```
while true
do

done
```

он будет выполняться бесконечно, поскольку команда `true` всегда возвращает нулевой код завершения.

А команда `false` всегда возвращает ненулевой код завершения, и поэтому следующий цикл `until`:

```
until false
do

done
```

будет также выполняться бесконечно.

Таким образом, команда `break` применяется для выхода из подобного рода бесконечных циклов. Как правило, это делается при обнаружении ошибочного условия или окончания обработки, как демонстрируется в следующем примере:

```
while true
do
      cmd=$(getcmd)
```

```
        if [ "$cmd" = quit ]
        then
                break
        else
                processcmd "$cmd"
        fi
done
```

В данном примере цикл `while` будет продолжать выполнение команд `getcmd` и `processcmd` до тех пор, пока значение переменной `cmd` не станет равным **`quit`**. Именно в этот момент будет выполнена команда `break`, что приведет к окончанию данного цикла.

Если команда `break` применяется в следующей форме:

```
break n
```

то происходит немедленный выход из *n* самых внутренних циклов. Таким образом, в следующем примере:

```
for file
do
        ...
        while [ "$count" -lt 10 ]
        do
                ...
                if [ -n "$error" ]
                then
                        break 2
                fi

        done

done
```

циклы `while` и `for` будут прерваны, если значение переменной `error` окажется непустым.

Пропуск или оставление команд в цикле

Команда `continue` подобна команде `break`, но она не приводит к прерыванию цикла, а только оставляет команды на текущем шаге цикла, пропуская его. В этом случае выполнение программы сразу же переходит к следующему шагу цикла, выполняя его, как обычно. Как и в команде `break`, в команде `continue` может быть указано дополнительное число *n*:

```
continue n
```

которое обозначает пропуск команд в самых внутренних циклах, после чего выполнение программы продолжается обычным образом:

```
for file
do
      if [ ! -e "$file" ]
      then
            echo "$file not found!"
            continue
      fi

      #
      # Обработать файл
      #


done
```

В данном примере каждое значение из списка, хранящегося в переменной `$file`, проверяется на предмет существования файла. Если файл не существует, выводится соответствующее сообщение, и дальнейшая обработка файла пропускается. Цикл продолжает выполняться со следующего значения в указанном списке.

Код из приведенного выше примера равнозначен написанию следующего фрагмента кода:

```
for file
do
      if [ ! -e "$file" ]
      then
            echo "$file not found!"
      else

            #
            # Обработать файл
            #


      fi
done
```

Выполнение цикла в фоновом режиме

Выполнение всего цикла может быть перенесено в фоновый режим. Для этого достаточно указать знак **&** после оператора `done`, как показано ниже.

```
$ for file in memo[1-4]
> do
>      run $file
> done &                Перенести выполнение цикла в фоновый режим
[1] 9932
$
request id is laser1-85 (standard input)
```

```
request id is laser1-87 (standard input)
request id is laser1-88 (standard input)
request id is laser1-92 (standard input)
```

Этот и последующие примеры вполне работоспособны, поскольку оболочка интерпретирует циклы как отдельные мини-программы. И все, что следует после таких операторов, закрывающих блок кода, как, например, done, fi или esac, может быть перенесено на выполнение в фоновый режим с помощью знака **&** или даже частичной формы организации конвейера команд.

В следующем примере весь вывод переадресовывается из цикла в файл output, за исключением результата, выводимого из команды echo, который направляется непосредственно на терминал:

```
for file
do
      echo "Processing file $file" > /dev/tty
      ...
done > output
```

Стандартный вывод ошибок можно также переадресовывать из цикла. Для этого достаточно указать форму переадресации 2> *имя_файла* после оператора done, как показано ниже. В данном примере стандартный вывод ошибок из всех команд будет переадресован в файл errors.

```
while [ "$endofdata" -ne TRUE ]
do
      ...
done 2> errors
```

Для вывода сообщений об ошибках на терминал даже в том случае, если их предполагается переадресовывать в файл или канал, служит следующая разновидность формы переадресации 2>:

```
echo "Error: no file" 1>&2
```

По умолчанию команда echo направляет результат своего выполнения в стандартный вывод (с дескриптором файла **1**), тогда как дескриптор файла **2** остается для вывода ошибок, который умолчанию не подлежит переадресации в файлы или каналы. Таким образом, приведенное выше обозначение означает, что сообщение об ошибке, выводимое командой echo, должно быть переадресовано из "файла **#1**" в "файл **#2**", т.е. в стандартный вывод. В этом можно убедиться, выполнив следующий фрагмент кода:

```
for i in "once"
do
      echo "Standard output message"
      echo "Error message" 1>&2
done > /dev/null
```

Конвейеризация ввода из вывода данных из цикла

Вывод из команды можно конвейеризировать в цикл, указав знаки канала между командами, выполняемыми в конвейере, и последующим циклом, а вывод из цикла — в последующую команду. Ниже приведен пример конвейеризации вывода из цикла `for` в команду `wc`.

```
$ for i in 1 2 3 4
> do
>       echo $i
> done | wc -l
       4
$
```

Ввод цикла в одной строке

Если часто приходится набирать циклы непосредственно в командной строке, то с этой целью можно опробовать приведенное ниже сокращенное обозначение для ввода всего цикла в одной строке. Для этого достаточно указать точку с запятой сначала вслед за последним элементом в списке, а затем после каждой команды в теле цикла, но только не после оператора `do`. Используя это сокращенное обозначение, следующий цикл:

```
for i in 1 2 3 4
do
      echo $i
done
```

может быть переписан таким образом:

```
for i in 1 2 3 4; do echo $i; done
```

Этот цикл можно набрать непосредственно в командной строке следующим образом:

```
$ for i in 1 2 3 4; do echo $i; done
1
2
3
4
$
```

Те же самые правила применяются к циклам `while` и `until`. Условные операторы `if` также могут быть набраны в одной строке в той же самой форме:

```
$ if [ 1 = 1 ]; then echo yes; fi
yes
$ if [ 1 = 2 ]; then echo yes; else echo no; fi
no
$
```

Следует, однако, иметь в виду, что после операторов then и else точки с запятыми не указываются. Многие программирующие на языке оболочки пользуются гибридной структурой, где условные операторы if структурированы таким образом:

```
if [ условие ] ; then
      команда
fi
```

Столь простое употребление точки с запятой может способствовать повышению удобочитаемости программ оболочки. И поэтому его стоит включить в арсенал своих средств форматирования исходного текста программ.

Команда getopts

Произведем дальнейшее расширение программы waitfor, добавив параметр **-t**, обозначающий, как часто (в секундах) осуществляется проверка регистрации нужного пользователя в системе. Теперь программа waitfor должна принимать два параметра, **-m** и **-t**. Допустим, эти параметры могут указываться в любом порядке в командной строке, но перед именем контролируемого пользователя. Таким образом, достоверные варианты ввода данной программы в командной строке выглядят следующим образом:

```
waitfor ann
waitfor -m ann
waitfor -t 600 ann
waitfor -m -t 600 ann
waitfor -t 600 -m ann
```

а недостоверные варианты — таким образом:

```
waitfor                 Отсутствует имя пользователя
waitfor -t600 ann       Требуется пробел после параметра -t
waitfor ann -m          Сначала должны быть указаны параметры
waitfor -t ann          Отсутствует аргумент после параметра -t
```

Как только вы начнете писать код, допускающий подобную гибкость в командной строке, то сразу же обнаружите, что он становится довольно сложным! Но не переживайте, ведь в оболочке имеется встроенная команда getopts, упрощающая обработку аргументов командной строки. Ниже приведена общая форма этой команды.

```
getopts параметры переменная
```

Строку параметров мы обсудим несколько позже, а до тех пор достаточно знать, что параметры, обозначаемые только буквами, указываются как таковые, а параметры, требующие дополнительных аргументов, дополняются завершающим двоеточием. Так, строка параметров «**ab:c**» означает, что в команде допускаются

параметры **-a**, **-b** и **-c**, но вместе с параметром **-b** требуется указать дополнительный аргумент.

Но команда `getopts` предназначена для выполнения в цикле, поскольку она упрощает выполнение таких действий, которые требуются для каждого из указанных параметров. На каждом шаге цикла команда `getopts` проверяет следующий аргумент командной строки и определяет его достоверность, выясняя, начинается ли он со знака "минус" и любых последующих букв, обозначаемых как *параметры*. В таком случае команда `getopts` сохраняет совпавшую букву параметра, которую содержит указанный список *переменная*, и возвращает нулевой код завершения. Ниже поясняется, что все это означает.

Если буква, указанная после знака "минус", отсутствует в списке *параметры*, команда `getopts` сохраняет знак вопроса в указанной *переменной*, прежде чем возвратит нулевой код завершения. Она направляет также в стандартный вывод ошибок сообщение о параметре, неверно указанном пользователем.

Если в командной строке больше не остается аргументов или же если текущий аргумент не начинается со знака "минус", команда `getopts` возвращает ненулевой код завершения, позволяя затем обработать любые последующие аргументы в сценарии или программе. Рассмотрим в этом контексте команду `ls -C /bin`, где `-C` — параметр, который может быть синтаксически проанализирован и обработан по команде `getopts`, а `/bin` — аргумент самой команды `ls`, обрабатываемый после всех начальных аргументов.

На первый взгляд, приведенное выше пояснение кажется несколько запутанным. Все это действительно не так просто, поэтому обратимся к конкретному примеру, чтобы продемонстрировать принцип действия команды `getopts`. Допустим, что в создаваемом сценарии требуется воспользоваться командой `getopts`, чтобы распознать параметры **-a**, **-i** и **-r**. В таком случае вызов этой команды может выглядеть следующим образом:

```
getopts "air" option
```

где первый аргумент **air** обозначает три допустимых параметра команды (**-a**, **-i** и **-r**), а второй аргумент **option** — имя переменной, используемой в команде `getopts` для хранения каждого обнаруженного при совпадении значения.

Команда `getopts` позволяет также группировать параметры вместе в командной строке. Для этого достаточно указать один знак "минус" и несколько последующих параметров. Так, команда `foo` в рассматриваемом здесь примере может быть выполнена следующим образом:

```
foo -a -r -i
```

или же так:

```
foo -ari
```

если воспользоваться простым свойством группирования.

На самом деле команда getopts намного более эффективна, чем было продемонстрировано до сих пор! Она, например, способна учесть и то обстоятельство, что параметру требуется дополнительный аргумент. Например, новый параметр **-t**, которым необходимо дополнить команду waitfor, должен быть указан с дополнительным аргументом.

Чтобы проанализировать надлежащим образом аргументы указанного пользователем параметра, команде getopts требуется, чтобы анализируемая команда вызывалась хотя бы с одним пробелом, отделяющим параметр от аргумента. В данном примере параметры не могут быть сгруппированы вместе.

Чтобы указать команде getopts на то, что параметру требуется аргумент, после буквы, обозначающей этот параметр, следует указать двоеточие в строке команды getopts. Таким образом, в программе waitfor, допускающей наличие параметров **-m** и **-t**, последнему из которых требуется дополнительный аргумент, команду getopts следует вызвать так:

```
getopts mt: option
```

Если команда getopts не обнаружит аргумент после того параметра, которому он требуется, она сохранит знак вопроса в указанной переменной и направит сообщение об ошибке в стандартный вывод ошибок. В противном случае она сохранит символ в указанной переменной, а введенный пользователем аргумент — в специальной переменной OPTARG.

И последнее замечание относительно команды getopts. Значение **1**, первоначально устанавливаемое в еще одной специальной переменной OPTIND, обновляется всякий раз, когда команда getopts возвращает результат, отражающий порядковый номер аргумента командной строки, который должен быть обработан *следующим* по порядку.

Чтобы разъяснить эту особенность команды getopts, рассмотрим приведенную ниже третью версию программы waitfor, где команда getopts применяется для обработки аргументов командной строки. В этой версии реализовано также упоминавшееся выше изменение для отправки сообщения по электронной почте пользователю, выполняющему данную программу.

```
$ cat waitfor
#
# Ожидать до тех пор, пока указанный пользователь
# не зарегистрируется в системе - версия 3
#

# Установить значения по умолчанию

mailopt=FALSE
interval=60
```

```
# Обработать параметры команды

while getopts mt: option
do
      case "$option"
      in
            m) mailopt=TRUE;;
            t) interval=$OPTARG;;
           \?) echo "Usage: waitfor [-m] [-t n] user"
                echo " -m means to be informed by mail"
                echo " -t means check every n secs."
                 exit 1;;
      esac
done

# Убедиться, что указано имя пользователя

if [ "$OPTIND" -gt "$#" ]
then
      echo "Missing user name!"
      exit 2
fi

shiftcount=$((OPTIND - 1))
shift $shiftcount
user=$1

#
# Проверять, зарегистрировался ли
# указанный пользователь в системе
#

until who | grep "^$user " > /dev/null
do
      sleep $interval
done

#
# Если программа дойдет до этого места, значит,
# указанный пользователь зарегистрировался в системе
#

if [ "$mailopt" = FALSE]
then
      echo "$user has logged on"
else
      runner=$(who am i | cut -cl-8)
      echo "$user has logged on" | mail $runner
fi

$ waitfor -m
Missing user name!
```

```
$ waitfor -x fred                    Недопустимый параметр
waitfor: illegal option -- x
Usage: waitfor [-m] [-t n] user
    -m means to be informed by mail
    -t means check every n secs.
$ waitfor -m -t 600 ann &            Проверять регистрацию в системе
                                     пользователя ann каждые 10 минут
[1] 5792
$
```

Проанализируем последнюю строку более подробно. Когда выполняется строка

```
waitfor -m -t 600 ann &
```

в цикле `while` происходит следующее: вызывается команда `getopts`, сохраняющая символ **m** в переменной `option`, устанавливающая значение **2** в переменной `OPTIND` и возвращающая нулевой код завершения.

Затем в операторе `case` определяется, что именно было сохранено в переменной `option`. Совпадение с символом **m** указывает на то, что выбран параметр, определяющий отправку сообщения по электронной почте, и поэтому в переменной `$mailopt` устанавливается логическое значение `TRUE`. (Обратите внимание на то, что знак **?** в операторе `case` специально экранирован. Благодаря этому исключается его специальное назначение в оболочке как символа совпадения с шаблоном.)

При выполнении во второй раз команда `getopts` сохраняет символ **t** в переменной `option`, следующий в командной строке аргумент (**600**) — в переменной `OPTARG`, а значение **3** — в переменной `OPTIND` и возвращает нулевой код завершения. Затем в операторе `case` проверяется совпадение с символом **t**, хранящимся в переменной `option`. Далее в коде, указанном в данной ветви оператора `case`, значение **600**, сохраненное в переменной `OPTARG`, копируется в переменную `interval`. А когда команда `getopts` выполняется в третий раз, она возвращает ненулевой код завершения, указывая на то, что она достигла конца списка операторов, указанных пользователем в командной строке.

После этого в данной программе сравниваются значения в переменных `OPTIND` и `$#`, чтобы проверить, было ли имя пользователя введено в командной строке. Если значение переменной `OPTIND` оказывается больше, чем значение переменной `$#`, это означает, что аргументов в командной строке больше не осталось, а пользователь программы забыл указать имя нужного ему пользователя в качестве аргумента. Конкретное количество позиций, на которые требуется произвести сдвиг, оказывается на единицу меньше, чем значение переменной `OPTIND`.

Остальная часть программы `waitfor` остается такой же, как и прежде. Единственное изменение состоит в применении переменной `interval` для хранения промежутка времени (в секундах), на который прерывается данная программа.

Если у вас голова идет кругом от команды `getopts`, не отчаивайтесь — ее применение будет еще не раз продемонстрировано в примерах программ оболочки из последующих глав, что поможет вам лучше раскрыть ее истинный потенциал. Ее изучение окажется полезным и для совершенствования в программировании на языке оболочки, поскольку синтаксический анализ вручную не одного, а нескольких аргументов в командной строке крайне неэффективен.

9

Ввод и вывод данных

В этой главе поясняется, каким образом данные вводятся с терминала или файла по команде `read` и направляются в отформатированном виде в стандартный вывод по команде `printf`.

Команда `read`

Общая форма команды `read` выглядит следующим образом:

```
read переменные
```

Когда выполняется эта команда, оболочка читает строку из стандартного ввода и присваивает первое слово переменной, перечисленной первой в списке *переменные*, второе слово — второй переменной и т.д. Если же слов в строке оказывается больше, чем перечисленных выше переменных, лишние слова присваиваются последней переменной. Например, по следующей команде:

```
read x y
```

строка читается из стандартного ввода, сохраняя первое слово в переменной `x`, а остальные слова — в переменной `y`. Таким образом, по следующей команде:

```
read text
```

вся строка читается и сохраняется в переменной оболочки `text`.

Пример программы копирования файлов

Попробуем применить команду `read` на практике, написав программу в виде упрощенной версии команды `cp`. Эта программа называется `mycp` и должна принимать два аргумента, обозначающие исходный и целевой файлы. Если целевой файл уже существует, программа `mycp` предупредит пользователя и предложит ему продолжить операцию копирования. При положительном ответе пользователя операция копирования будет продолжена, а иначе произойдет выход из данной программы, как показано ниже.

```
$ cat mycp
#
# Скопировать файл
#

if [ "$#" -ne 2 ] ; then
      echo "Usage: mycp from to"
```

```
      exit 1
fi

from="$1"
to="$2"

#
# Выяснить, существует ли целевой файл
#

if [ -e "$to" ] ; then
      echo "$to already exists; overwrite (yes/no)?"
      read answer

      if [ "$answer" != yes ] ; then
            echo "Copy not performed"
            exit 0
      fi
fi

#
# Целевой файл не существует или введен положительный ответ
#

cp $from $to    # продолжить копирование файла
$
```

А теперь оперативно проверим данную программу в действии:

```
$ ls -C                          Что за файлы? Параметр -C определяет
                                 вывод результата в несколько столбцов
Addresses     intro    lotsaspaces    mycp
names         nu       numbers        phonebook
stat
$ mycp                           Без аргументов
Usage: mycp from to
$ mycp names names2              Скопировать файл names
$ ls -l names*                   Произошло ли копирование успешно?
-rw-r--r--    1   steve   steve    43 Jul  20 11:12  names
-rw-r--r--    1   steve   steve    43 Jul  21 14:16  names2
$ mycp names numbers             Попытаться переписать существующий файл
numbers already exists; overwrite (yes/no)?
no
Copy not performed
$
```

Следует заметить, что если целевой файл уже существует, команда `echo` выводит приглашение дать положительный или отрицательный ответ. А следующая далее команда `read` вынуждает оболочку ожидать ответа пользователя. В данном примере программы демонстрируется, что сама оболочка не выводит приглашение, ожидая от пользователя ввода данных. Эта обязанность возлагается на автора программы.

Данные, вводимые пользователем, сначала сохраняются в переменной answer, а затем сравниваются с символьной строкой "yes", чтобы выяснить, следует ли продолжить процесс копирования файла. Кавычки, в которые заключена переменная answer в следующей проверке:

```
[ "$answer" != yes]
```

необходимы на тот случай, если пользователь просто нажмет клавишу <Enter>, не вводя никаких данных. В таком случае оболочка сохранит в переменной answer пустое значение. А без кавычек в результате данной проверки было бы выведено сообщение об ошибке.

Обратите также внимание на точку с запятой, которая позволяет указывать операторы then в одной строке с условным оператором if. Это типичный прием, к которому прибегают программирующие на языке оболочки, как упоминалось в предыдущей главе.

Употребление специальных управляющих символов в команде echo

Некоторое неудобство при использовании программы mycp возникает в связи с тем, что данные, введенные пользователем после выполнения команды echo, отображаются в следующей строке. Это происходит потому, что команда echo автоматически добавляет знак новой строки вслед за последним аргументом.

Правда, этот знак нетрудно подавить, если указать специальный *управляющий* символ \c в конце команды echo. Тем самым команде echo предписывается пропустить знак новой строки после отображения последнего аргумента. Так, если внести в вызов команды echo из программы mycp следующие изменения:

```
echo "$to already exists; overwrite (yes/no)? \c"
```

введенные пользователем данные отобразятся в той же самой строке. Следует, однако, иметь в виду, что управляющий символ \c интерпретируется командой echo, а не оболочкой. Это означает, что он должен быть экранирован знаком обратной косой черты, чтобы его правильно восприняла команда echo.

Примечание

Оболочки в некоторых системах Linux и Mac OS X не интерпретируют управляющие символы в команде **echo**, и поэтому приведенная выше команда **echo** отобразит следующий результат:

```
newfile.txt already exists; overwrite (yes/no)? \c
```

Если в результате проверки выявится, что оболочка действует именно таким образом, вызовы команды **echo** в этих конкретных местах программы следует заменить на обращения к отдельной программе **/bin/echo**, и тогда они будут выполнены правильно. А все остальные вызовы обычной команды **echo** следует оставить без изменения.

Команда echo способна интерпретировать и другие специальные символы, перечисленные в табл. 9.1. Если же этого не происходит, см. приведенное выше примечание. Каждый такой символ должен быть экранирован знаком обратной косой черты.

Таблица 9.1. Специальные управляющие символы, употребляемые в команде echo

Символ	Что отображает
\b	Возврат на одну позицию
\c	Строка без завершающего знака новой строки
\f	Перевод страницы
\n	Перевод строки или новая строка
\r	Перевод каретки
\t	Табуляция
\\	Обратная косая черта
\0*nnn*	Символ со значением *nnn* в коде ASCII, где *nnn* — восьмеричное число, состоящее из одной–трех цифр

Усовершенствованная версия программы mycp

Допустим, в текущем каталоге имеется программа prog1 и ее требуется скопировать в каталог bin. Это нетрудно сделать с помощью обычной команды cp, указав имя исходного файла и целевой каталог для копирования этого файла. Но вернемся снова к программе mycp и выясним, что произойдет, если ввести в командной строке следующее:

```
mycp prog1 bin
```

Проверка наличия файла в каталоге bin с помощью операции **-e** пройдет успешно, программа mycp выведет сообщение о том, что файл уже существует, и перейдет в режим ожидания положительно или отрицательного ответа. Но ведь это потенциально опасная ошибка, особенно если пользователь введет положительный ответ! Если же в качестве второго аргумента указан *каталог*, то в программе mycp вместо этого должно быть проверено, существует ли исходный файл из переменной from в указанном каталоге.

Именно эта проверка и производится в приведенной ниже следующей версии программы mycp. В ней также применяется видоизмененная команда echo, включающая управляющий символ \c для подавления завершающего знака новой строки.

```
$ cat mycp
#
# Копировать файл - версия 2
#
```

```
if [ "$#" -ne 2 ] ; then
      echo "Usage: mycp from to"
      exit 1
fi
from="$1"
to="$2"

#
# Выяснить, является целевой файл каталогом
#

if [ -d "$to" ] ; then
      to="$to/$(basename $from)"
fi

#
# Выяснить, существует ли уже целевой файл
#

if [ -e "$to" ] ; then
      echo "$to already exists; overwrite (yes/no)? \c"
      read answer
      if [ "$answer" != yes ] ; then
            echo "Copy not performed"
            exit 0
      fi
fi

#
# Целевой файл не существует или введен положительный ответ
#

cp $from $to      # продолжить копирование файла
$
```

Если целевой файл оказывается каталогом, значение переменной `$to` уточняется и в ней сохраняется имя целевого файла вместе с именем каталога: `$to/$(basename $from)`. Этим гарантируется, что последующая проверка существования обыкновенного файла из переменной `$from` будет выполнена относительно файла в каталоге, а не самого каталога.

Команда `basename` возвращает базовое имя файла по значению ее аргумента, отбрасывая все составляющие, относящиеся к каталогам (например, по команде `basename /usr/bin/troff` получается имя файла `troff`, как, впрочем, и по команде `basename troff`). Этой дополнительной мерой гарантируется, что копия файла из исходного места будет сделана в нужное место назначения. Так, если ввести в командной строке

```
mycp /tmp/data bin
```

где `bin` — каталог, это означает, что файл `/tmp/data` требуется скопировать в файл `bin/tmp/data`.

Ниже приведены некоторые примеры выполнения программы myсp в новой ее версии. Обратите внимание на результат действия управляющего символа \c.

```
$ ls                          Проверить текущий каталог
bin
prog1
$ ls bin                      Заглянуть в каталог bin
lu
nu
prog1
$ mycp prog1 prog2            Простой случай
$ mycp prog1 bin              Копировать файл в тот же каталог
bin/prog1 already exists; overwrite (yes/no)? yes
$
```

Окончательная версия программы mycp

В последней рассматриваемой здесь версии программа myсp становится практически равноценной стандартной для Linux команде cp, допуская переменное количество аргументов. Напомним, что в стандартной команде cp прежде имени каталога может быть указано любое количество файлов, как показано ниже.

```
cp prog1 prog2 greetings bin
```

С целью видоизменить программу myсp таким образом, чтобы она принимала любое количество файлов в сходном формате, можно применить следующий подход.

1. Получить из командной строки каждый аргумент, кроме последнего, а затем сохранить его в переменной оболочки filelist.
2. Сохранить последний аргумент в переменной to.
3. Если переменная to не содержит каталог, проверить подсчитанное количество аргументов, которых должно быть ровно два.
4. Если же переменная to содержит каталог, проверить, существует ли каждый файл из переменной filelist в целевом каталоге. Если файл не существует в этом каталоге, добавить его имя к переменной copylist. А если файл существует, то запросить у пользователя разрешение на перезапись этого файла. И если пользователь даст положительный ответ (yes), то добавить имя этого файла к переменной copylist.
5. Если значение переменной copylist окажется непустым, скопировать файлы в целевой каталог, хранящийся в переменной to.

Если этот алгоритм покажется не совсем ясным, приведенный ниже исходный текст и подробное пояснение данной программы помогут прояснить дело. Обратите внимание на сообщение о правильном применении видоизмененной версии данной программы.

```
$ cat mycp
#
# Копировать файл - окончательная версия
#

numargs=$#          # сохранить для последующего применения
filelist=
copylist=
#
# Обработать аргументы, сохранив в переменной filelist
# все, кроме последнего аргумента
#

while [ "$#" -gt 1 ] ; do
       filelist="$filelist $1"
       shift
done
to="$1"

#
# Если количество аргументов меньше или больше двух, а
# последний аргумент не является каталогом, вывести
# сообщение об ошибке
#

if [ "$numargs" -lt 2 -o "$numargs" -gt 2 -a ! -d "$to" ] ; then
      echo "Usage: mycp file1 file2"
      echo " mycp file(s) dir"
      exit 1
fi

#
# Перебрать каждый файл в переменной filelist
#

for from in $filelist ; do
      #
      # Выяснить, является целевой файл каталогом
      #

      if [ -d "$to" ] ; then
            tofile="$to/$(basename $from)"
      else
            tofile="$to"
      fi

      #
      # Добавить имя файла к переменной copylist
      # если файл еще не существует или же если
      # пользователь дает разрешение на его перезапись
```

```
      if [ -e "$tofile" ] ; then
          echo "$tofile already exists; overwrite (yes/no)? \c"
          read answer

          if [ "$answer" = yes ] ; then
                copylist="$copylist $from"
          fi
      else
          copylist="$copylist $from"
      fi
done

#
# А теперь сделать копию. Сначала убедиться в наличии
# файлов для копирования
#

if [ -n "$copylist" ] ; then
      cp $copylist $to          # продолжить копирование файлов
fi
$
```

А теперь рассмотрим некоторые примеры применения программы mycp, прежде чем пояснять ее исходный текст.

```
$ ls -C                  Выяснить, какие файлы имеются в наличии
bin lu names prog1
prog2
$ ls bin                    И что находится в каталоге bin
lu
nu
prog1
$ mycp                      Без аргументов
Usage: mycp file1 file2
       mycp file(s) dir
$ mycp names prog1 prog2    Последний аргумент не является каталогом
Usage: mycp file1 file2
       mycp file(s) dir
$ mycp names prog1 prog2 lu bin        Правильное применение
bin/prog1 already exists; overwrite (yes/no)? yes
bin/lu already exists; overwrite (yes/no)? no
$ ls -l bin                            Выяснить, что же произошло
total 5
-rw-r--r--   1   steve   steve   543  Jul  19  14:10  lu
-rw-r--r--   1   steve   steve   949  Jul  21  17:11  names
-rw-r--r--   1   steve   steve   38   Jul  19  09:55  nu
-rw-r--r--   1   steve   steve   498  Jul  21  17:11  prog1
-rw-r--r--   1   steve   steve   498  Jul  21  17:11  prog2
$
```

В последнем случае файл prog1 был перезаписан, тогда как файл lu — не перезаписан по запросу пользователя. Когда данная программа начинает выполняться, она сохраняет количество своих аргументов в переменной $numargs. Это делается потому, что переменные аргументов изменяются по команде shift далее в программе.

Далее начинается цикл, который выполняется до тех пор, пока количество аргументов остается больше единицы. Назначение данного цикла — получить последний аргумент в командной строке. В ходе этого процесса остальные аргументы сохраняются в переменной оболочки filelist, где в конечном итоге оказывается список всех копируемых файлов. В следующем операторе:

```
filelist="$filelist $1"
```

к предыдущему значению переменной filelist добавляется пробел и значение переменной $1, а полученный результат сохраняется обратно в переменной filelist. Затем выполняется команда shift, сдвигающая все аргументы на одну позицию. В конечном итоге значение переменной $# окажется равным единице, и на этом цикл завершится. В этот момент переменная filelist будет содержать разделяемый пробелами список всех копируемых файлов, а переменная $1 — последний аргумент, который обозначает имя целевого файла или каталога.

Чтобы стало яснее, как действует этот алгоритм, рассмотрим последовательные шаги цикла while, выполнив программу mycp следующим образом:

```
mycp names prog1 prog2 lu bin
```

На рис. 9.1 показано, каким образом изменяются значения переменных на каждом шаге данного цикла. Так, в первой строке показано состояние переменных перед началом цикла.

$#	$1	$2	$3	$4	$5	filelist
5	names	prog1	prog2	lu	bin	null
4	prog1	prog2	lu	bin		names
3	prog2	lu	bin			names prog1
2	lu	bin				names prog1 prog2
1	bin					names prog1 prog2 lu

Рис. 9.1. Порядок обработки аргументов командной строки

По окончании цикла последний аргумент, хранившийся в переменной $1, сохраняется в переменной to. Далее производится проверка с целью убедиться, что в командной строке было введено по крайней мере два аргумента. И если их было введено больше двух, то последний аргумент обозначает целевой каталог. Если

же ни одно из этих условий не удовлетворяется, то для пользователя на терминал выводится сообщение о правильном употреблении рассматриваемой здесь программы, после чего программа завершается с кодом состояния **1**.

После этого в цикле `for` проверяется каждый файл в списке, чтобы выяснить, существует ли он уже в целевом каталоге. Если он существует, то пользователь, как и прежде, получает приглашение дать ответ относительно копирования данного файла. Если пользователь решит перезаписать существующий файл или же если этот файл не существует, его имя добавляется к переменной оболочки `copylist`. В данном случае применяется такой же прием, как и при накапливании аргументов в переменной `filelist`.

По окончании цикла `for` переменная `copylist` содержит список всех копируемых файлов. В крайнем случае этот список может оказаться пустым, если пользователь решит не перезаписывать ни один из указанных файлов, уже существующих в целевом каталоге. Следовательно, проверяется, содержит ли переменная `copylist` непустое значение, и в этом случае выполняется копирование файлов.

Уделите немного времени анализу логики окончательной версии программы `mycp`, так как в ней демонстрируются многие средства, рассмотренные до сих пор в данной книге. Свое понимание данной программы вы сможете проверить на примерах, приведенных в конце этой главы.

Управляемая через меню программа ведения телефонного справочника

Команда `read` полезна еще и тем, что она позволяет писать управляемые через меню программы оболочки. В качестве примера вернемся к рассмотрению трех упоминавшихся ранее программ ведения телефонного справочника (`add`, `lu` и `rem`) и создадим так называемую *обертку* — программу, упрощающую применение других программ. На этот раз мы создадим программу-обертку `rolo`, сокращенно названную от Rolodex — фирменного наименования вращающегося каталога с карточками, используемыми для хранения контактной информации.

Когда программа `rolo` вызывается, она отображает сначала список вариантов выбора, сделанного пользователем, а затем выполняет соответствующую программу в зависимости от конкретного выбора после приглашения ввести требующиеся аргументы. Ниже приведен исходный текст программы `rolo`.

```
$ cat rolo
#
# rolo - программа для поиска, ввода и удаления
# сведений об абонентах из телефонного справочника
#
# Отобразить меню
#
```

```
echo '
      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): \c'

#
# Прочитать и обработать выбранный вариант
#

read choice
echo ""
case "$choice"
in
      1) echo "Enter name to look up: \c"
         read name
         lu "$name";;
      2) echo "Enter name to be added: \c"
         read name
         echo "Enter number: \c"
         read number
         add "$name" "$number";;
      3) echo "Enter name to be removed: \c"
         read name
         rem "$name";;
      *) echo "Bad choice";;
esac
$
```

Обратите внимание, как по одной команде `echo` отображается все многострочное меню. С этой целью выгодно используется то обстоятельство, что кавычки сохраняют форматирование и встраиваемые знаки новой строки. А команда `read` получает вариант, выбранный пользователем из меню, и сохраняет его в переменной `choice`.

В операторе `case` определяется, какой именно вариант был выбран из меню. Так, если выбран вариант **1**, это означает, что пользователю требуется найти какого-то абонента в телефонном справочнике. И в этом случае пользователю предлагается ввести имя искомого абонента, которое служит в качестве аргумента при последующем вызове программы `lu`. Обратите также внимание на кавычки, в которые заключена переменная `name` в приведенной ниже строке. Они необходимы для того, чтобы программа `lu` интерпретировала два или более слова, введенных пользователем, как единый аргумент.

```
lu "$name"
```

Аналогичная последовательность действий происходит, если пользователь выберет вариант **2** или **3** из меню. При выборе первых трех вариантов из меню вызываются упоминавшиеся в предыдущих главах программы `lu`, `add` и `rem` соответственно. Ниже приведены некоторые примеры применения программы `rolo`.

```
$ rolo

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 2
      Enter name to be added: El Coyote
      Enter number: 212-567-3232
$ rolo       Попробовать еще раз

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 1

      Enter name to look up: Coyote
      El Coyote 212-567-3232
$ rolo       Попробовать снова

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 4
Bad choice
$
```

Если введен неверный вариант выбора, данная программа выводит сообщение `"Bad choice"` (Неудачный выбор) и на этом завершается. Более удобный для пользователя подход состоит в том, чтобы повторять приглашение на ввод варианта выбора до тех пор, пока пользователь не сделает правильный выбор из меню. С этой целью всю программу следует заключить в цикл `until`, который будет выполняться до тех пор, пока не будет сделан правильный выбор.

Еще одно изменение в программе `rolo` должно отражать наиболее вероятное ее применение для поиска абонента в телефонном справочнике. Вместо того чтобы набирать **`rolo`**, выбирать вариант **1**, а затем вводить имя искомого абонента, намного проще набрать непосредственно **`lu имя`**

С этой целью снабдим программу rolo аргументами командной строки, чтобы эффективно пользоваться ими. Если указаны какие-нибудь аргументы, программа rolo по умолчанию предположит, что запрашивается поиск абонента, и сразу же вызовет программу lu, передав ей все свои аргументы. Так, если пользователю требуется произвести быстрый поиск, он может набрать **rolo** и далее имя искомого абонента. Если же пользователю требуется отобразить все меню, ему достаточно набрать лишь **rolo**.

Оба упомянутых изменения (циклическое выполнение до тех пор, пока не будет сделан правильный выбор, а также быстрый поиск абонента) внесены в версию 2 программы rolo, приведенную ниже.

```
$ cat rolo
#
# rolo - программа для поиска, ввода и удаления сведений
# об абонентах из телефонного справочника - версия 2
#


#
# Если предоставлены аргументы, произвести поиск
#

if [ "$#" -ne 0 ] ; then
      lu "$@"
      exit
fi

validchoice=""           # установить пустое значение

#
# Выполнить цикл до тех пор, пока не будет
# сделан правильный выбор
#

until [ -n "$validchoice" ]
do

      #
      # Отобразить меню
      #

      echo '

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): \c'
```

```
        #
        # Прочитать и обработать выбранный вариант
        #

        read choice
        echo

        case "$choice"
        in
              1) echo "Enter name to look up: \c"
                 read name
                 lu "$name"
                 validchoice=TRUE;;
              2) echo "Enter name to be added: \c"
                 read name
                 echo "Enter number: \c"
                 read number
                 add "$name" "$number"
                 validchoice=TRUE;;
              3) echo "Enter name to be removed: \c"
                 read name
                 rem "$name"
                 validchoice=TRUE;;
              *) echo "Bad choice";;
        esac
done
$
```

Если значение переменной `$#` не равно нулю, программа `lu` вызывается непосредственно с аргументами, введенными в командной строке, после чего рассматриваемая здесь программа завершается. В противном случае цикл `until` выполняется до тех пор, пока значение переменной `$validchoice` остается пустым. Напомним, что единственный способ присвоить ей другое значение состоит в том, чтобы выполнить следующую операцию:

```
validchoice=TRUE
```

в ветвях выбора **1**, **2** или **3** из оператора `case`. В противном случае программа продолжит свое выполнение в цикле `until`. Ниже приведены некоторые примеры применения программы `rolo` в ее новой версии.

```
$ rolo Bill                    Быстрый поиск абонента
Billy Bach 201-331-7618
$ rolo                         Воспользоваться на этот раз меню

        Would you like to:

              1. Look someone up
              2. Add someone to the phone book
              3. Remove someone from the phone book
```

```
      Please select one of the above (1-3): 4
      Bad choice

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 0
      Bad choice

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 1
      Enter name to look up: Tony
      Tony Iannino 973-386-1295
$
```

Переменная $$ и временные файлы

Если программу `rolo` одновременно выполняют два или несколько пользователей, работающих в системе, это может вызвать определенные осложнения. Проанализируйте исходный текст программы `rem` и попробуйте выявить причину этих осложнений. Они возникают в связи с применением временного файла `/tmp/phonebook` для создания новой версии файла телефонного справочника. Конкретные операторы из программы `rem`, где могут возникнуть подобные осложнения, выглядят следующим образом:

```
grep -v "$name" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
```

Если несколько пользователей одновременно обращаются к программе `rolo` с целью удалить запись из файла `phonebook`, этот файл может быть испорчен, поскольку для этой цели одновременно используется один и тот же временный файл. Откровенно говоря, вероятность возникновения подобного осложнения невелика, но исключить его полностью нельзя, а следовательно, оно может рано или поздно произойти.

В связи с этим в рассматриваемой здесь программе необходимо решить два важных вопроса. Первый связан с тем, каким образом на компьютерах фактически реализована многозадачность путем подкачки и откачки отдельных программ из процессора. В итоге программа может быть откачена из процессора в любой момент своего выполнения — даже посредине последовательности команд. Если обратиться к приведенным выше операторам из программы `rem`, то что может

произойти, если выполнение одного экземпляра данной программы прервется в промежутке между этими двумя операторами, а в другом ее экземпляре будут выполнены те же самые операторы? Во втором экземпляре данной программы будет перезаписан результат вызова команды `grep` из первого ее экземпляра, что неприемлемо для обоих ее пользователей!

Второй вопрос связан с так называемым *состоянием гонок* — ситуацией, в которой несколько одновременных вызовов программы могут вызвать серьезные осложнения. Чаще всего состояние гонок возникает в связи с временными файлами, но оно может произойти и с подпроцессами и файлами блокировки, обсуждаемыми далее в книге. А до тех пор достаточно иметь в виду, что при написании программ оболочки, которые могут одновременно выполняться несколькими пользователями, следует убедиться, что для каждого пользователя выделен отдельный временный файл.

Выходом из рассмотренного выше затруднения может стать создание временных файлов в начальном каталоге пользователя вместо каталога `/tmp` или выбор однозначного имени временного файла, которое должно отличаться при каждом вызове программы. С этой целью можно вставить однозначный идентификатор процесса (PID) в имя временного файла при каждом вызове программы. И сделать это нетрудно по ссылке на специальную переменную `$$`.

```
$ echo $$
4668
$ ps
  PID    TTY    TIME    COMMAND
  4668   co     0:09    sh
  6470   co     0:03    ps
$
```

При подстановке, выполняемой оболочкой, ссылка на переменную `$$` заменяется номером идентификатора процесса самой исходной оболочки. А поскольку каждому процессу в системе присваивается однозначный идентификатор, то с помощью значения переменной `$$` удается исключить возможность использования одного и того же имени временного файла в разных процессах. Чтобы устранить упомянутые выше осложнения в связи с одновременным выполнением программы `rolo` несколькими пользователями, включая и возможное состояние гонок, достаточно заменить приведенные ранее строки кода из программы `rem` следующими строками:

```
grep -v "$name" phonebook > /tmp/phonebook$$
mv /tmp/phonebook$$ phonebook
```

Каждый пользователь будет выполнять программу `rolo` как отдельный процесс, и поэтому в каждом ее экземпляре будет использоваться однозначное имя временного файла. Подобным образом устраняются рассмотренные выше осложнения.

Код завершения, возвращаемый командой `read`

Команда `read` возвращает нулевой код завершения, если только не возникнет условие в связи с обнаружением конца файла. Если данные поступают с терминала, это означает, что пользователь нажал комбинацию клавиш <Ctrl+D>. А если данные поступают из файла, то это означает, что в файле больше не осталось данных для чтения. Этим упрощается составление цикла для чтения строк данных из файла или с терминала.

В следующем примере демонстрируется программа `addi`, где читаются строки, состоящие из пар чисел, которые складываются, а полученные суммы направляются в стандартный вывод:

```
$ cat addi
#
# Сложить пары целых чисел, читаемых из стандартного ввода
#

while read n1 n2
do
      echo $(( $n1 + $n2))
done
$
```

Цикл `while` выполняется при условии, что команда `read` возвратит нулевой код завершения, а это произойдет лишь в том случае, если для чтения все еще имеются данные. В теле данного цикла прочитанные из строки значения (предположительно целочисленные, поскольку никакой проверки ошибочного ввода данных не производится) складываются вместе, а полученный результат направляется в стандартный вывод по команде `echo`. Ниже приведен пример применения программы `addi`.

```
$ addi
10 25
35
-5 12
7
123 3
126
<Ctrl+D>
$
```

Стандартный ввод для программы `addi` может быть переадресован из файла, а стандартный вывод — в другой файл (и, конечно, канал), как показано ниже.

```
$ cat data
1234 7960
593 -595
395 304
3234 999
```

```
-394 -493
$ addi < data > sums
$ cat sums
9194
-2
699
4233
-887
$
```

Ниже приведен пример еще одной программы, которая называется number и является упрощенной версией стандартной команды nl в системе Unix. Она принимает в качестве аргументов один или несколько файлов и отображает их содержимое в отдельно пронумерованных строках. Если же аргументы не предоставляются, данная программа читает данные из стандартного ввода.

```
$ cat number
#
# Пронумеровать строки, введенные из файлов, указанных
# в качестве аргументов, или же из стандартного ввода,
# если файлы не предоставлены
#

lineno=1

cat $* |
while read line
do
      echo "$lineno: $line"
      lineno=$((lineno + 1))
done
$
```

Первоначально в переменной lineno, где хранится подсчитанное количество строк, устанавливается значение **1**. Затем аргументы, введенные при вызове программы number, передаются команде cat с целью направить их сообща в стандартный вывод. Если же никаких аргументов не предоставлено, значение переменной $* окажется пустым, поэтому команде cat не будет передано никаких аргументов. Это, в свою очередь, приведет к чтению данных из стандартного ввода. А результат, выводимый из команды cat, по каналу направляется циклу while.

Каждая строка, вводимая по команде read, отображается эхом на терминал и предваряется текущим значением переменной lineno, которое увеличивается на единицу на каждом шаге цикла while. Ниже приведены некоторые примеры применения программы number.

```
$ number phonebook
1: Alice Chebba      973-555-2015
2: Barbara Swingle   201-555-9257
```

```
3: Billy Bach          201-555-7618
4: El Coyote           212-555-3232
5: Liz Stachiw         212-555-2298
6: Susan Goldberg      201-555-7776
7: Teri Zak            201-555-6000
8: Tony Iannino        973-555-1295
$ who | number                    Попробовать прочитать данные
                                  из стандартного ввода
1: root     console   Jul  25  07:55
2: pat      tty03     Jul  25  09:26
3: steve    tty04     Jul  25  10:58
4: george   tty13     Jul  25  08:05
$
```

Однако программа `number` не вполне справляется с обработкой строк, содержащих знаки обратной косой черты или начальные пробелы. Именно это обстоятельство демонстрируется в следующем примере:

```
$ number
          Here are some backslashes: \ \*
1: Here are some backslashes: *
$
```

В данном примере начальные пробелы удаляются из любой прочитанной строки. Знаки обратной косой черты также интерпретируются оболочкой при чтении строки. Чтобы предотвратить интерпретацию знаков обратной косой черты, можно указать параметр **`-r`** при вызове команды `read`. Так, если заменить следующую строку из программы `number`:

```
while read line
```

на строку

```
while read -r line
```

то результат выполнения данной программы в новой версии будет выглядеть лучше:

```
$ number
          Here are some backslashes: \ \*
1: Here are some backslashes: \ \*
$
```

В главе 11 поясняется, как можно сохранить начальные пробелы и проконтролировать синтаксический анализ входных данных.

Команда `printf`

Несмотря на то что команда `echo` вполне пригодна для вывода простых сообщений, иногда требуется вывести *отформатированный* результат, выровняв,

например, данные в столбцах. Для решения подобных задач в системах Unix предоставляется команда `printf`. Те, у кого имеется некоторый опыт программирования на языке C или C++, обнаружат немало сходств этой команды с функцией `printf()`.

Общая форма команды `printf` выглядит следующим образом:

```
printf "формат" arg1 arg2 ...
```

где *формат* — это форматирующая строка с подробными сведениями о том, как отображать значения последующих аргументов. Форматирующая строка представлена одним аргументом и, вероятнее всего, содержит специальные символы и пробелы, поэтому ее целесообразно всегда заключать в кавычки.

Те символы в форматирующей строке, которые не предваряются знаком процентов (**%**), направляются непосредственно в стандартный вывод. Поэтому в своей простейшей форме команда `printf` может действовать подобно команде `echo`, при условии, что в конце каждой выводимой строки указан специальный символ `\n`, обозначающий переход на новую строку, как демонстрируется в следующем примере:

```
$ printf "Hello world!\n"
Hello world!
$
```

Символы, предваряемые знаком процента, называются *спецификаторами формата* и сообщают команде `printf` порядок отображения соответствующих аргументов. Для каждого знака процента в форматирующей строке должен быть указан соответствующий аргумент, за исключением специального спецификатора формата **%%**, обозначающего вывод одного знака процента. Итак, начнем рассмотрение форматированного вывода по команде `printf` со следующего простого примера:

```
$ printf "This is a number: %d\n" 10
This is a number: 10
$
```

Команда `printf` не добавляет автоматически, подобно команде `echo`, знак новой строки в конце выводимого результата, но в то же время она воспринимает все управляющие символы, которые были перечислены в табл. 9.1. Так, если добавить управляющий символ `\n` в конце формирующей строки, он будет благополучно выведен, а приглашение на ввод команд появится, как и предполагалось, в следующей строке.

Несмотря на то что приведенный выше простой пример может быть реализован и с помощью команды `echo`, он помогает продемонстрировать, каким образом спецификатор формата (**%d**) интерпретируется командой `printf`, которая анализирует форматирующую строку и выводит каждый ее символ до тех пор,

пока не встретится знак процента. Затем команда `printf` читает символ **d** и пытается заменить спецификатор **%d** следующим аргументом, который предоставлен команде `printf` и должен быть целочисленным. После того как аргумент (**10**) будет направлен в стандартный вывод, команда `printf` продолжит анализ форматирующей строки и, обнаружив управляющий символ \n, выведет знак новой строки. Различные символы спецификаторов формата сведены в табл. 9.2.

Таблица 9.2. Символы спецификаторов формата в команде `printf`

Символ	Применение для форматирования вывода
%d	Целые числа
%u	Целые числа без знака
%o	Целые числа в восьмеричном формате
%x	Целые числа в шестнадцатеричном формате с использованием букв **a-f**
%X	Целые числа в шестнадцатеричном формате с использованием букв **A-F**
%c	Одиночные символы
%s	Символьные строки в буквальном виде
%b	Символьные строки, содержащие управляющие символы, экранированные знаками обратной черты
%%	Знаки процентов

Все пять символов спецификаторов формата, перечисленных первыми в табл. 9.2, служат для отображения целых чисел. В частности, спецификатор формата **%d** служит для отображения целых чисел со знаком, а спецификатор формата **%u** — для отображения целых чисел без знака, хотя им можно воспользоваться и для отображения отрицательных целых чисел в положительном представлении. По умолчанию целые числа, отображаемые в восьмеричном или шестнадцатеричном формате, не содержат префикс **0** или **0x**, хотя это положение можно исправить, как будет показано далее.

Символьные строки выводятся с помощью спецификатора формата **%s** или **%b**. В частности, спецификатор формата **%s** служит для вывода символьных строк в буквальном виде, т.е. без обработки любых управляющих символов, экранируемых знаками обратной косой черты, а спецификатор формата **%b** — для принудительной интерпретации управляющих символов в строковом аргументе команды `printf`.

Для большей ясности ниже приведен ряд примеров применения команды `printf`.

```
$ printf "The octal value for %d is %o\n" 20 20
The octal value for 20 is 24
$ printf "The hexadecimal value for %d is %x\n" 30 30
The hexadecimal value for 30 is 1e
$ printf "The unsigned value for %d is %u\n" -1000 -1000
```

```
The unsigned value for -1000 is 4294966296
$ printf "This string contains a backslash
Äescape: %s\n" "test\nstring"
This string contains a backslash escape: test\nstring
$ printf "This string contains an interpreted
Äbackslash escape: %b\n" "test\nstring"
This string contains an interpreted backslash escape: test string
$ printf "A string: %s and a character: %c\n" hello A
A string: hello and a character: A
$
```

В последнем примере применения команды `printf` спецификатор формата **%c** служит для отображения единственного символа. Если же соответствующий аргумент оказывается длиннее одного символа, то отображается только первый символ:

```
$ printf "Just the first character: %c\n" abc
a
$
```

Общая форма преобразования спецификаторов формата выглядит следующим образом:

```
%[флаги][ширина][.точность]тип
```

где *тип* — символ спецификатора формата из табл. 9.2. Как видите, для обозначения спецификатора формата требуется только знак процента и *тип*, а остальные параметры, называемые *модификаторами*, необязательны. Достоверными считаются следующие *флаги*: **-**, **+**, **#**, а также пробел.

В частности, флаг **-** обозначает выравнивание выводимого значения по левому краю, что станет понятнее при дальнейшем рассмотрении модификатора *ширина*. Флаг **+** предписывает команде `printf` предварять целые числа знаком **+** или **-** (по умолчанию только отрицательные числа выводятся со знаком). Флаг **#** предписывает команде `printf` предварять восьмеричные числа префиксом **0**, а шестнадцатеричные — префиксом **0x** или **0X**, обозначаемым как **%#x** или **%#X** соответственно. И наконец, символ пробела предписывает команде `printf` предварять положительные целые числа пробелом, а отрицательные целые числа — знаком **-** для целей выравнивания.

Ниже приведен ряд дополнительных примеров применения команды `printf`. Обратите особое внимание на форматирующие строки!

```
$ printf "%+d\n%+d\n%+d\n" 10 -10 20
+10
-10
+20
$ printf "% d\n% d\n% d\n" 10 -10 20
10
-10
```

```
20
$ printf "%#o %#x\n" 100 200
0144 0xc8
$
```

Как видите, применение знака **+** или пробела в качестве флага приводит к аккуратному выравниванию столбца с положительными и отрицательными числами по левому краю. Модификатор *ширина* — это положительное число, обозначающее минимальную *ширину поля* для вывода аргумента. Если флаг **-** не указан, аргумент в этом поле выравнивается по правому краю, как показано ниже.

```
$ printf "%20s%20s\n" string1 string2
             string1             string2
$ printf "%-20s%-20s\n" string1 string2
string1             string2
$ printf "%5d%5d%5d\n" 1 10 100
    1   10  100
$ printf "%5d%5d%5d\n" -1 -10 -100
   -1   -10 -100
$ printf "%-5d%-5d%-5d\n" 1 10 100
1    10   100
$
```

Модификатор *ширина* может оказаться полезным для выравнивания столбцов текста или чисел. (Следует иметь в виду, что знаки и префиксы **0**, **0x**, **0X** в числах считаются частью ширины аргумента.) В общем, модификатор *ширина* обозначает *минимальную* ширину поля, но если ширина аргумента превышает величину, указанную в модификаторе *ширина*, то может произойти переполнение и ничего вообще не будет выведено. В этом нетрудно убедиться, выполнив приведенную ниже команду. Что произойдет в вашей системе при попытке выполнить ее?

```
printf "%-15.15s\n" "this is more than 15 chars long"
```

Модификатор *.точность* — это положительное число, обозначающее минимальное количество цифр в числах, отображаемых с помощью спецификаторов формата **%d**, **%u**, **%o**, **%x** и **%X**. В итоге выводимое числовое значение *заполняется нулями* слева, как демонстрируется в следующем примере:

```
$ printf "%.5d %.4X\n" 10 27
00010 001B
$
```

Что же касается символьных строк, то модификатор *.точность* обозначает максимальное количество символов, выводимых из строки. Если же символьная строка оказывается длиннее, чем количество символов, указанных в модификаторе *.точность*, то она усекается справа. Это очень важно иметь в виду при выравнивании многострочного текста, поскольку некоторые данные могут быть утрачены, если отдельное значение окажется больше заданной ширины поля:

```
$ printf "%.6s\n" "Ann Smith"
Ann Sm
$
```

Модификаторы *ширина* и *.точность* можно сочетать вместе, чтобы обозначить как ширину поля, так и заполнение чисел начальными нулями или усечение символьных строк. Такое сочетание обоих модификаторов демонстрируется в следующих примерах:

```
$ printf ":%#10.5x:%5.4x:%5.4d\n" 1 10 100
:   0x00001: 000a: 0100
$ printf ":%9.5s:\n" abcdefg
:    abcde:
$ printf ":%-9.5s:\n" abcdefg
:abcde    :
$
```

И наконец, если все изложенные выше правила форматирования вывода покажутся довольно сложными, то вместо модификатора *ширина* или *.точность* ради простоты можно указать знак *. В этом случае аргумент, предшествующий выводимому значению, должен быть числом, обозначающим ширину поля или точность представления соответственно. Если же знак * употребляется вместо обоих модификаторов, то выводимому значению должны предшествовать два числовых аргумента, обозначающих ширину поля и точность представления, как демонстрируется в приведенном ниже примере.

```
$ printf "%*s%*.*s\n" 12 "test one" 10 2 "test two"
    test one        te
$ printf "%12s%10.2s\n" "test one" "test two"
    test one        te
$
```

Как видите, обе команды `printf` в данном примере дают одинаковый результат. В первом случае числовой аргумент **12** служит для обозначения ширины первой символьной строки, тогда как аргумент **10** — для обозначения ширины второй символьной строки и, наконец, аргумент **2** — для обозначения точности представления второй символьной строки. А во втором случае эти числовые аргументы указаны непосредственно в спецификаторе формата.

Безусловно, спецификаторы формата, употребляемые в команде `printf`, довольно сложны. Тем не менее их истинный потенциал позволяет преобразовать относительно неструктурированный вывод из команды `echo` в требуемый формат вывода результатов из программ оболочки. Поэтому команду `printf` стоит изучить досконально, чтобы знать, как пользоваться ею при разработке собственных изощренных программ. Различные модификаторы, употребляемые в спецификаторах формата, перечислены в табл. 9.3.

Таблица 9.3. Модификаторы, употребляемые в спецификаторах формата

Модификатор	Назначение
флаги	
-	Выравнивает значение по левому краю
+	Предваряет целочисленное значение знаком + или -
пробел	Предваряет положительное целочисленное значение символом пробела
#	Предваряет целочисленное значение в восьмеричном формате префиксом 0, а в шестнадцатеричном формате — префиксом **0x** или **0X**
ширина	Обозначает минимальную ширину поля; а знак * — применение последующего аргумента для указания ширины поля
точность	Обозначает минимальное количество цифр, отображаемых в целых числах; максимальное количество символов, отображаемых в строках; а знак * — применение последующего аргумента для указания точности представления

Продолжим далее исследование потенциальных возможностей команды `printf`. Ниже приведен пример простой программы, где команда `printf` применяется для выравнивания двух столбцов чисел, выводимых из файла.

```
$ cat align
#
# Выровнять два столбца чисел
# (подходит для чисел длиной до 12 цифр, включая знак)
#

cat $* |
while read number1 number2
do
      printf "%12d %12d\n" $number1 $number2
done
$ cat data
1234 7960
593 -595
395 304
3234 999
-394 -493
$ align data
        1234         7960
         593         -595
         395          304
        3234          999
        -394         -493
$
```

Дополнительные примеры применения команды `printf` будут продемонстрированы в главах 11, 13 и 14.

10

Рабочая среда

Когда вы входите в свою систему, будь то новенькая версия приложения Mac OS X Terminal, чистая установка Linux или сервер Unix в личном кабинете, вы, по существу, получаете свою копию программы оболочки. Эта исходная оболочка поддерживает так называемую *среду* — конфигурацию, отличающуюся от других пользователей в системе. Эта среда поддерживается с момента вашей регистрации в системе и до тех пор, пока вы не выйдете из нее. В этой главе речь пойдет о среде оболочки и будет показано, каким образом она связана с написанием и выполнением программ.

Локальные переменные

Наберите следующую программу под названием `vartest` на своем компьютере:

```
$ cat vartest
echo :$x:
$
```

Программа `vartest` состоит из единственной команды `echo`, отображающей значение переменной `x`, окруженной двоеточиями.

Присвойте любое значение переменной `x`, введя его с терминала, например, следующим образом:

```
$ x=100
```

Вопрос: что, на ваш взгляд, отобразится, если теперь выполнить программу `vartest`? Ответ:

```
$ vartest
::
$
```

Программе `vartest` ничего неизвестно о значении переменной `x`. Следовательно, по умолчанию она принимает пустое значение. Переменная `x`, которой было присвоено значение **100** в исходной оболочке, называется *локальной*. Ниже поясняется, почему она получила именно такое наименование.

Рассмотрим в качестве примера еще одну программу `vartest2`:

```
$ cat vartest2
x=50
echo :$x:
$ x=100
$ vartest2              Выполнить эту программу
:50:
$
```

В данной программе значение переменной x изменяется с **50** на **100**. В связи с этим возникает следующий вопрос: какое значение сохранится в переменной x после выполнения данной программы? Ответ:

```
$ echo $x
100
$
```

Как видите, программа `vartest2` не изменила значение переменной x, где было установлено значение **100** в интерактивной оболочке.

Подоболочки

Необычное на первый взгляд поведение, проявляемое программами `vartest` и `vartest2`, обусловлено тем, что эти программы выполняются исходной оболочкой в отдельных *подоболочках*. По существу, *подоболочка* является совершенно новой оболочкой, предназначенной лишь для выполнения требуемой программы.

Когда вы обращаетесь к исходной оболочке с запросом выполнить программу `vartest`, она запускает новую оболочку для выполнения этой программы. Всякий раз, когда действует новая оболочка, она выполняет собственную среду со своими локальными переменными. Подоболочке неизвестны локальные переменные, которым присваиваются значения в исходной (*родительской*) оболочке. Более того, в подоболочке нельзя изменить значение переменной из родительской оболочки, о чем свидетельствует результат выполнения программы `vartest2`.

Рассмотрим происходящий при этом процесс, чтобы стало понятнее, что собой представляет *соблюдение области действия* переменных в программах оболочки. Перед выполнением программы `vartest2` в оболочке переменной x было присвоено значение **100** (рис. 10.1).

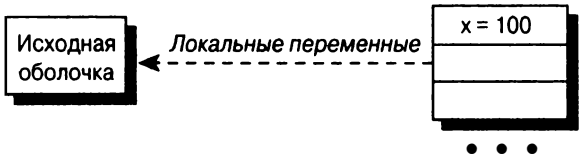


Рис. 10.1. Процесс выполнения операции **x=100** в исходной оболочке

Когда же вызывается программа `vartest2`, исходная оболочка запускает подоболочку для выполнения данной программы, передавая ей первоначально *пустой* список локальных переменных (рис. 10.2).

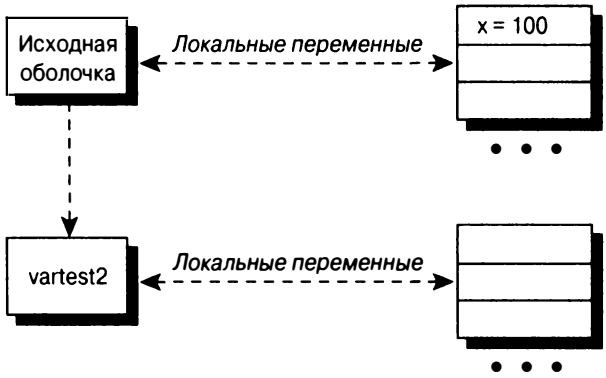


Рис. 10.2. Процесс выполнения программы **`vartest2`** в исходной оболочке

После выполнения первой команды в программе `vartest2` локальная переменная `x`, *существующая в среде подоболочки*, будет иметь значение **50** (рис. 10.3). Но значение переменной `x` в родительской оболочке останется без изменения.

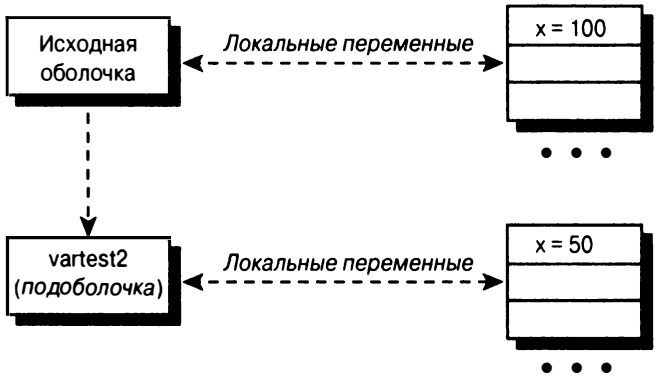


Рис. 10.3. Процесс выполнения операции **`x=50`** в программе **`vartest2`**

Когда программа `vartest2` завершает свое выполнение, подоболочка исчезает вместе с *любыми переменными*, которым были присвоены *значения* в данной программе. На самом деле это не такое уж и большое затруднение, как может показаться на первый взгляд. Нужно лишь понять, что среда исходной оболочки отличается от среды подоболочек, что и отражается на результатах выполнения в них программ.

Экспортируемые переменные

Чтобы сделать значение переменной доступным для подоболочки, эту переменную можно *экспортировать* по команде `export`. Форма этой команды довольно проста и выглядит следующим образом:

```
export переменные
```

где *переменные* — список имен переменных, которые требуется экспортировать. Значения экспортируемых переменных будут далее переданы любым подоболочкам, выполняемым после команды `export`.

Ниже приведен пример программы `vartest3`, помогающей наглядно продемонстрировать отличия локальных переменных от экспортируемых.

```
$ cat vartest3
echo x = $x
echo y = $y
$
```

Присвойте сначала значения переменным `x` и `y` в исходной оболочке, а затем выполните программу `vartest3`:

```
$ x=100
$ y=10
$ vartest3
x =
y =
$
```

Как видите, переменные `x` и `y` являются локальными, поэтому их значения не передаются подоболочке, в которой выполняется программа `vartest3`. Этого и следовало ожидать. А теперь экспортируйте переменную `y` и попытайтесь снова выполнить программу `vartest3`, как показано ниже. На этот раз переменная `y` окажется доступной программе `vartest3`, поскольку она была экспортирована.

```
$ export y          Сделать переменную y доступной для подоболочек
$ vartest3
x =
y = 10
$
```

Всякий раз, когда выполняется подоболочка, экспортируемые переменные, по существу, "копируются" в подоболочку, тогда как с локальными переменными этого не происходит (рис. 10.4).

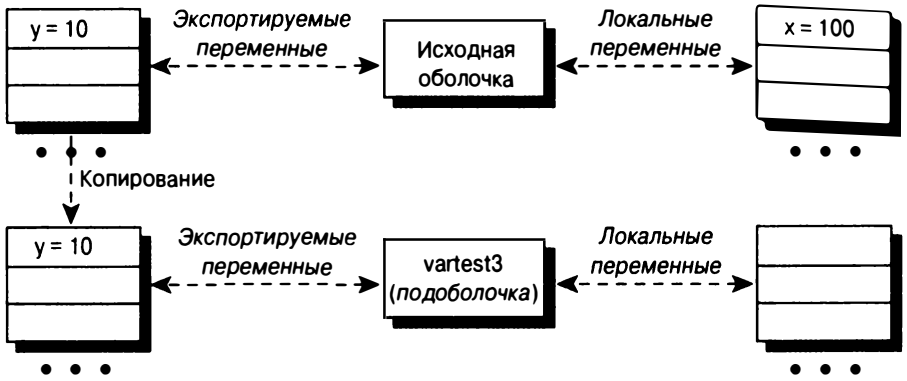


Рис. 10.4. Процесс выполнения программы **vartest3**

А теперь самое время задать следующий вопрос: что произойдет, если значение экспортированной переменной изменится в подоболочке, т.е. окажется ли она доступной в родительской оболочке по завершении подоболочки? Чтобы ответить на этот вопрос, рассмотрим следующую программу под названием `vartest4`:

```
$ cat vartest4
x=50
y=5
$
```

Допустим, что значения переменных `x` и `y` не изменились и что переменная `y` по-прежнему экспортируется из предыдущего примера, как показано ниже.

```
$ vartest4
$ echo $x $y
100 10
$
```

В подоболочке не удалось изменить значение как локальной переменной `x`, что не удивительно, так и экспортируемой переменной `y`. В ней удалось изменить лишь свою локальную копию переменной `y`, экземпляр которой был получен при выполнении этой подоболочки (рис. 10.5). А когда исчезнет подоболочка, то вместе с ней исчезнут значения не только локальных переменных, но и присвоенные в ней значения экспортируемых переменных. В действительности, как только переменные оказываются в подоболочке, они становятся *локальными*. Именно поэтому в подоболочке *нельзя* изменить значение переменной из родительской оболочки.

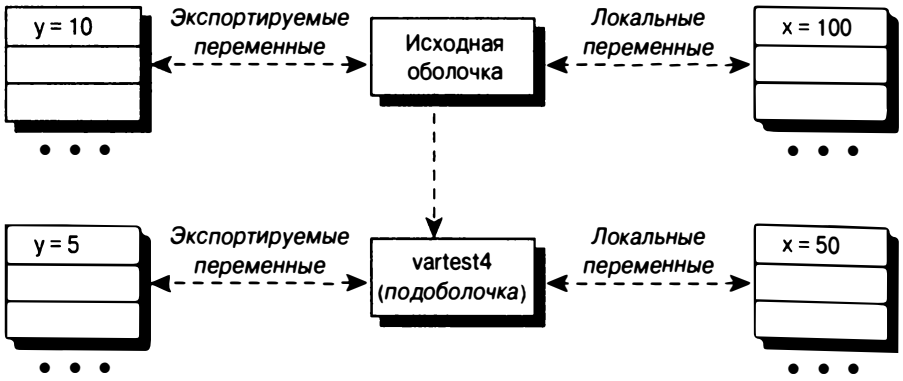


Рис. 10.5. Процесс выполнения программы **vartest4**

В том случае, если в одной программе оболочки вызывается другая программа оболочки (например, программа `rolo` вызывает программу `lu`), то процесс повторяется, когда экспортируемые переменные из прежней подоболочки копируются в новую подоболочку. Эти экспортируемые переменные могли бы быть экспортированы из исходной оболочки или же вновь экспортированы из самой подоболочки.

Как только переменная будет экспортирована, она останется экспортированной и во все выполняемые далее подоболочки.

Рассмотрим следующую видоизмененную версию программы `vartest4`:

```
$ cat vartest4
x=50
y=5
z=1
export z
vartest5
$
```

а также программу `vartest5`:

```
$ cat vartest5
echo x = $x
echo y = $y
echo z = $z
$
```

Когда выполняется программа `vartest4`, экспортируемая переменная `y` будет скопирована в среду подоболочки. Программа `vartest4` сначала устанавливает значение **50** в переменной `x`, изменяет на **5** значение в переменной `y` и устанавливает значение **1** в переменной `z`, а затем экспортирует переменную `z`, значение которой становится доступным для любой последующей подоболочки.

Такой подоболочкой служит программа `vartest5`, и когда она выполняется, оболочка копирует в ее среду экспортируемые переменные `y` и `z` из программы `vartest4`. Именно этим объясняется следующий результат:

```
$ vartest4
x =
y = 5
z = 1
$
```

Весь этот процесс наглядно показан на рис. 10.6.

Рис. 10.6. Процесс выполнения подоболочки

Таким образом, локальные и экспортируемые переменные действуют следующим образом.

1. Любая переменная, которая не экспортируется, является локальной, причем о ее существовании не будет известно подоболочкам.
2. Экспортируемые переменные и их значения копируются в среду подоболочки, где они могут быть доступны и изменены. Тем не менее подобные изменения не оказывают никакого влияния на переменные в родительской оболочке.
3. Экспортируемые переменные сохраняют свои свойства не только для непосредственно порождаемых подоболочек, но и для подоболочек, порождаемых этими подоболочками, и так далее по цепочке.

4. Переменная может быть экспортирована в любой момент до или после присваивания ей значения, но она принимает свое значение в момент экспорта, а последующие изменения в ней не отслеживаются.

Команда `export -p`

Если ввести команду **`export -p`**, то в итоге будет получен список переменных и их значений, экспортируемых в рабочую оболочку, как показано ниже.

```
$ export -p
export LOGNAME=steve
export PATH=/bin:/usr/bin:.
export TIMEOUT=600
export TZ=EST5EDT
export y=10
$
```

Как видите, в типичной исходной оболочке имеется немало экспортируемых переменных. Например, в системе Mac OS команда `export -p` выдаст список из 22 переменных. Обратите внимание на то, что в приведенном выше примере переменная `y` появляется из предыдущего примера экспериментирования с экспортом наряду с другими переменными, которые были экспортированы, когда вы вошли в систему и тем самым запустили свою исходную оболочку. Но что означают перечисленные выше переменные с именами, набранными прописными буквами? Рассмотрим их более подробно.

Переменные `PS1` и `PS2`

Последовательность символов, употребляемых в оболочке в качестве приглашения на ввод команд, хранятся в переменной окружения `PS1`. Но значение этой переменной можно изменить на любое другое, и с этого момента новое значение будет использоваться далее в оболочке, как демонстрируется в следующем примере:

```
$ echo :$PS1:
:$ :
$ PS1="==> "
==> pwd
/users/steve
==> PS1="I await your next command, master: "
I await your next command, master: date
Wed Sep 18 14:46:28 EDT 2002
I await your next command, master: PS1="$ "
$                          Вернуться к обычному приглашению
```

По умолчанию в качестве вспомогательного приглашения, когда команду требуется ввести в нескольких строках, употребляется знак **>**, который хранится в

переменной PS2. Содержимое и этой переменной можно изменить по своему усмотрению, например, следующим образом:

```
$ echo :$PS2:
:> :
$ PS2="=======> "
$ for x in 1 2 3
=======> do
=======> echo $x
=======> done
1
2
3
$
```

Как только вы выйдете из системы, все изменения, внесенные в упомянутых выше переменных, исчезнут, как, впрочем, и все изменения в любых других переменных оболочки. Если внести изменение в переменную PS1, ее новое значение будет использоваться далее в сеансе работы в оболочке. Но когда вы войдете в систему в следующий раз, то увидите прежнее приглашение, если только не сохраните новое значение переменной PS1, введя его в файл .profile, рассматриваемый далее в этой главе.

Совет

У приглашения, хранящегося в переменной **PS1**, имеется собственный язык со специальными последовательностями операций, производящих подсчет команд, выбор текущего каталога, определение времени суток и выполняющих многое другое. Ознакомиться с этим языком можно, прочитав раздел "Prompting" (Выдача приглашения) на оперативной странице руководства по оболочке Bash или Sh.

Переменная HOME

Ваш начальный каталог находится там, где вы размещаете его всякий раз, когда входите в систему. Этот каталог автоматически устанавливается также в специальной переменной оболочки HOME, когда вы регистрируетесь в системе, как демонстрируется в следующем примере:

```
$ echo $HOME
/users/steve
$
```

Этой переменной вы можете воспользоваться в своих программах с целью определить начальный каталог. Именно таким образом она употребляется в других программах системы Unix. Переменная HOME употребляется также в команде cd для перехода в требующийся целевой каталог, когда эта команда вводится без аргументов, как показано ниже.

```
$ pwd                    Отобразить текущий каталог
/usr/src/lib/libc/port/stdio
$ cd
$ pwd
/users/steve             Начальный каталог пользователя
$
```

Содержимое переменной HOME можно изменить на любое другое, как показано ниже. Но такое изменение может повлиять на действие программ, опирающихся на данную переменную.

```
$ HOME=/users/steve/book      Изменить содержимое переменной
$ pwd
/users/steve
$ cd
$ pwd                         Выяснить, что же произошло
/users/steve/book
$
```

Содержимое переменной HOME можно изменить. Но делать этого все же не рекомендуется, если только вы не готовы к тому, что положение дел может очень быстро оказаться весьма шатким.

Переменная PATH

Вернемся, однако, к программе rolo из главы 9, выполнив ее следующим образом:

```
$ rolo Liz
Liz Stachiw          212-775-2298
$
```

Допустим, что для приведения дел в более организованный порядок эта программа была создана в подкаталоге /bin пользователя steve:

```
$ pwd
/users/steve/bin
$
```

Перейдите в какой-нибудь другой каталог в файловой системе. Например, по следующей команде:

```
$ cd            Перейти в начальный каталог
$
```

А теперь попытайтесь найти абонента Liz в телефонном справочнике:

```
$ rolo Liz
sh: rolo: not found
$
```

Не удалось! Что же произошло?

Всякий раз, когда вы вводите имя программы, оболочка производит ее поиск в списке каталогов до тех пор, пока не найдет запрашиваемую программу. Как только программа будет найдена, она будет запущена на выполнение. Этот список каталогов для поиска пользовательских программ хранится в переменной оболочки PATH и автоматически устанавливается в ней, когда вы входите в систему. Чтобы выяснить в любой момент, что установлено в этой переменной, достаточно воспользоваться командой echo следующим образом:

```
$ echo $PATH
/bin:/usr/bin:.
$
```

Скорее всего, в переменной PATH установлено совсем другое значение, но в этом нет ничего страшного. Ведь это всего лишь отклонение от конфигурации системы. Важно лишь иметь в виду, что каталоги должны разделяться двоеточием (:) и что поиск в них запрашиваемых команд или программ оболочка производит по порядку слева направо.

В предыдущем примере перечислены три каталога: /bin, /usr/bin и . (точка, как известно, обозначает текущий каталог). Поэтому всякий раз, когда вводится имя программы, оболочка производит ее поиск в каталогах, перечисленных в переменной PATH, до тех пор, пока не найдет совпадающий исполняемый файл. Так, если ввести имя программы **rolo**, оболочка попробует найти ее исполняемый файл по пути /bin/rolo, затем по пути /usr/bin/rolo и, наконец, по пути ./rolo к текущему каталогу. Как только оболочка найдет искомый файл, она запустит его на выполнение. А если оболочке не удастся найти исполняемый файл rolo ни в одном из каталогов, перечисленных в переменной PATH, то она выдаст сообщение об ошибке "not found" (не найдено).

Чтобы поиск начинался, прежде всего, с текущего каталога, точку следует указать в переменной PATH первой, как показано ниже. Следует, однако, иметь в виду, что из соображений безопасности поиск *не* должен производиться в текущем каталоге пользователя прежде, чем в системных каталогах.

```
.:/bin:/usr/bin
```

Это правило следует соблюдать во избежание нарушений защиты системы с помощью так называемых *троянских коней*. Допустим, кто-нибудь создаст свою версию команды su, которая позволяет перейти в корневой каталог или установить привилегии суперпользователя по приглашению ввести пароль администратора системы, а затем разместит ее в текущем каталоге, ожидая до тех пор, пока другой пользователь перейдет в этот каталог и выполнит данную команду. Если этот текущий каталог указан в переменной PATH первым, то будет выполнена видоизмененная версия команды su. Но дело в том, что эта версия выдаст приглашение на ввод пароля, отправит его по электронной почте злонамеренному

пользователю, удалит себя и выведет безобидное сообщение об ошибке. В результате повторного вызова и ввода пароля учетная запись администратора окажется нарушенной, а пользователь даже не осознает, что же на самом деле произошло! Ловко, не так ли?

Точка (.), обозначающая текущий каталог, служит необязательным, но удобным наглядным напоминанием. Например, следующее значение переменной PATH:

```
:/bin:/usr/bin
```

равнозначно предыдущему. Но здесь и далее текущий каталог будет обозначаться точкой ради большей ясности.

Если вас беспокоят троянские кони, у вас всегда есть возможность переопределить порядок поиска, заданный в переменной PATH, явно указав путь к исполняемому файлу программы. Так, если вы введете следующий путь:

```
/bin/date
```

оболочка перейдет непосредственно к каталогу /bin, чтобы выполнить команду date. В этом случае содержимое переменной PATH будет проигнорировано, как, впрочем, и в том случае, если вы введете путь

```
../bin/lu
```

или

```
./rolo
```

В последнем случае предписывается выполнить программу rolo, исполняемый файл которой находится в текущем каталоге. Такой способ нередко применяется при разработке программ оболочки, поскольку он дает возможность опустить точку (.) в переменной PATH.

Теперь должно быть понятно, почему программу rolo нельзя было выполнить из начального каталога пользователя steve. В переменную PATH не был включен путь /users/steve/bin, и поэтому оболочке не удалось найти программу rolo. Чтобы устранить данное препятствие, проще всего включить путь к этому каталогу в переменную PATH следующим образом:

```
$ PATH=/bin:/usr/bin:/users/steve/bin:.
$
```

Теперь *любая* программа из каталога /users/steve/bin будет выполнена независимо от местоположения текущего каталога в файловой системе.

```
$ pwd                Отобразить текущий каталог
/users/steve
$ rolo Liz
grep: can't open phonebook
$
```

Что же теперь не так? Оболочка правильно обнаружила исполняемый файл программы rolo, но команде grep не удалось найти файл данных phonebook.

Если внимательно проанализировать исходный текст программы rolo, то можно обнаружить, что источником сообщения об ошибке в команде grep служит программа lu. Ниже приведен исходный текст программы lu в ее текущей версии.

```
$ cat /users/steve/bin/lu
#
# Найти абонента в телефонном справочнике - версия 3
#

if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments"
      echo "Usage: lu name"
      exit 1
fi

grep "$name" phonebook
$
```

Команда grep пытается найти файл phonebook в текущем каталоге, которым в настоящий момент является каталог /users/steve. Но дело в том, что рассматриваемая здесь программа выполняется безотносительно к каталогу, в котором находится она и ее файл данных.

В переменной PATH указываются только те каталоги, в которых ищутся исполняемые файлы программ, вызываемых из командной строки, но не любые другие типы файлов. Поэтому местонахождение файла данных phonebook должно быть точно определено, чтобы им было удобнее пользоваться в программе lu.

Этот недостаток, присущий не только программе lu, но и программам rem и add, можно исправить несколькими способами. В частности, перед вызовом команды grep в программе lu можно перейти в каталог /users/steve/bin, как показано ниже. И тогда команда grep обнаружит файл phonebook, поскольку он находится в новом текущем каталоге.

```
   ...
cd /users/steve/bin
grep "$1" phonebook
```

Такой способ пригоден в том случае, если приходится немало манипулировать различными файлами в отдельном каталоге. Для этого достаточно перейти сначала в нужный каталог по команде cd, а затем непосредственно обратиться ко всем требующимся файлам.

Второй, более распространенный способ состоит в том, чтобы указать полный путь к файлу phonebook в команде grep:

```
   ...
grep "$1" /users/steve/bin/phonebook
```

Допустим, что и другим пользователям требуется предоставить возможность применять программы rolo, а также связанные с ней программы lu, add и rem. С этой целью каждому из них можно было бы предоставить отдельную копию данной программы, но тогда в системе появилось бы несколько ее копий, обновлять которые было бы неудобно, если бы потребовалось внести даже незначительные изменения в программу. Поэтому лучше иметь единственную копию программы rolo, но предоставить доступ к ней другим пользователям.

В этой связи возникает следующее затруднение: если изменить все ссылки на файл phonebook таким образом, чтобы явно ссылаться на свой телефонный справочник, то и все остальные будут пользоваться *этим же* телефонным справочником. Поэтому лучше, чтобы у каждого пользователя был свой файл phonebook, доступный в его начальном каталоге по ссылке $HOME/phonebook.

Чтобы воспользоваться столь распространенным соглашением по программированию на языке оболочки, следует определить в программе rolo переменную PHONEBOOK, установив в ней удобное для многих пользователей значение $HOME/phonebook. Если экспортировать переменную PHONEBOOK, ее значение будет использоваться в программах lu, rem и add, выполняемых как подоболочки в программе rolo, для ссылки на отдельную версию файла phonebook соответствующего пользователя.

Преимущество такого подхода заключается, в частности, в следующем: если изменить местоположение файла phonebook, то достаточно внести изменение только в одну переменную в программе rolo, чтобы остальные три программы и дальше работали без сучка и задоринки. С учетом этих соображений ниже приведены новые версии программ rolo, lu, add и rem.

```
$ cd /users/steve/bin
$ cat rolo
#
# rolo - программа для поиска, ввода и удаления сведений
# об абонентах из телефонного справочника
#

# Установить в переменной PHONEBOOK ссылку на файл
# телефонного справочника и экспортировать ее, чтобы
# сделать доступной в других программах
#

PHONEBOOK=$HOME/phonebook
export PHONEBOOK

if [ ! -f "$PHONEBOOK" ] ; then
      echo "No phone book file in $HOME!"
      exit 1
fi
```

```
#
# Если предоставлены аргументы, произвести поиск
#

if [ "$#" -ne 0 ] ; then
      lu "$@"
      exit
fi
validchoice=""           # установить пустое значение

#
# Выполнить цикл до тех пор, пока не будет
# сделан правильный выбор
#

until [ -n "$validchoice" ]
do
      #
      # Отобразить меню
      #

      echo '
      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): \c'


      #
      # Прочитать и обработать выбранный вариант
      #

      read choice
      echo

      case "$choice"
      in
            1) echo "Enter name to look up: \c"
               read name
               lu "$name"
               validchoice=TRUE;;
            2) echo "Enter name to be added: \c"
               read name
               echo "Enter number: \c"
               read number
               add "$name" "$number"
               validchoice=TRUE;;
```

```
            3) echo "Enter name to be removed: \c"
               read name
               rem "$name"
               validchoice=TRUE;;
            *) echo "Bad choice";;
      esac
done
$ cat add
#
# Программа для ввода абонента в файл телефонного справочника
#

if [ "$#"_ -ne 2 ] ;then
      echo "Incorrect number of arguments"
      echo "Usage: add name number"
      exit 1
fi

echo "$1        $2" >> $PHONEBOOK
sort -o $PHONEBOOK $PHONEBOOK
$ cat lu
#
# Найти абонента в телефонном справочнике
#

if [ "$#" -ne 1 ] ; then
      echo "Incorrect number of arguments"
      echo "Usage: lu name"
      exit 1
fi

name=$1
grep "$name" $PHONEBOOK
if [ $? -ne 0 ] ; then
      echo "I couldn't find $name in the phone book"
fi
$ cat rem
#
# Удалить абонента из телефонного справочника
#

if [ "$#" -ne 1 ] ; then
      echo "Incorrect number of arguments"
      echo "Usage: rem name"
      exit 1
fi

name=$1

#
# Выявить количество совпавших записей
#
```

```
matches=$(grep "$name" $PHONEBOOK | wc -1)

#
# Если совпавших записей больше одной, вывести сообщение,
# а иначе – удалить запись
#

if [ "$matches" -gt 1 ] ; then
      echo "More than one match; please qualify further"
elif [ "$matches" -eq 1 ] ; then
      grep -v "$name" $PHONEBOOK > /tmp/phonebook$$
      mv /tmp/phonebook$$ $PHONEBOOK
else
      echo "I couldn't find $name in the phone book"
fi
$
```

В данном случае был применен еще один специальный прием. Ради большего удобства для пользователей в конце программы `lu` была введена проверка с целью выяснить, успешно ли была выполнена команда `grep`. Если же поиск не дает никаких результатов, то выводится соответствующее сообщение. А теперь проверим программу `rolo`:

```
$ cd                          Перейти в начальный каталог
$ rolo Liz                    Произвести быстрый поиск
No phonebook file in /users/steve! Forgot to move it
$ mv /users/steve/bin/phonebook .
$ rolo Liz                    Попробовать еще раз
Liz Stachiw 212-775-2298
$ rolo                        Попробовать с выбором варианта из меню
      Would you like to:
            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book

      Please select one of the above (1-3): 2

Enter name to be added: Teri Zak
Enter number: 201-393-6000
$ rolo Teri
Teri Zak 201-393-6000
$
```

Как видите, программы `rolo`, `lu` и `add` действуют исправно. Остается лишь проверить программу `rem`, чтобы убедиться и в ее исправности. Если же программу `lu`, `rem` или `add` требуется выполнить отдельно, это можно сделать, определив сначала переменную `PHONEBOOK`, а затем экспортировав ее, как демонстрируется в следующем примере:

```
$ PHONEBOOK=$HOME/phonebook
$ export PHONEBOOK
$ lu Harmon
I couldn't find Harmon in the phone book
$
```

Если упомянутые выше программы предполагается выполнять отдельно, в каждой из них следует ввести проверку установки правильного значения в переменной PHONEBOOK.

Текущий каталог пользователя

Ваш текущий каталог является также частью среды вашей оболочки. Рассмотрим в качестве примера следующую небольшую программу оболочки под названием cdtest:

```
$ cat cdtest
cd /users/steve/bin
pwd
$
```

В этой программе сначала выполняется команда cd для перехода в каталог /users/steve/bin, а затем вызывается команда pwd для проверки изменений в местоположении пользователя. Попробуем выполнить эту программу следующим образом:

```
$ pwd              Выяснить текущий каталог
/users/steve
$ cdtest
/users/steve/bin
$
```

А теперь попробуйте ответить на следующий любопытный вопрос: если вызвать команду pwd, то в каком каталоге вы окажетесь: /users/steve или /users/steve/bin? Ответ:

```
$ pwd
/users/steve
$
```

Оказывается, что выполнение команды cd в программе cdtest не оказывает никакого влияния на ваш текущий каталог. А поскольку текущий каталог является частью текущей среды, то когда команда cd выполняется из оболочки, она оказывает влияние только на смену каталога в данной оболочке. Перейти же в текущий каталог родительской оболочки из подоболочки *никак нельзя*.

Когда команда cd вызывается на выполнение, она сменяет текущий каталог, а также устанавливает в переменной PWD полный путь к новому текущему каталогу. В итоге следующая команда:

```
echo $PWD
```

дает такой же результат, как и команда pwd:

```
$ pwd
/users/steve
$ echo $PWD
/users/steve
$ cd bin
$ echo $PWD
/users/steve/bin
$
```

Кроме того, команда cd устанавливает в переменной OLDPWD полный путь к каталогу, который был прежде текущим. И в некоторых случаях это может быть удобно.

Переменная CDPATH

Переменная CDPATH действует таким же образом, как и переменная PATH. Она определяет список каталогов, в которых оболочка производит поиск всякий раз, когда выполняется команда cd. Такой поиск производится лишь в том случае, если указанный каталог не задан по полному пути и если значение переменной CDPATH оказывается непустым. Так, если ввести следующую команду:

```
cd /users/steve
```

оболочка сменит текущий каталог на /users/steve. Но если ввести команду

```
cd memos
```

оболочка воспользуется значением переменной CDPATH, чтобы найти каталог memos. И если значение переменной CDPATH окажется следующим:

```
$ echo $CDPATH
.:/users/steve:/users/steve/docs
$
```

оболочка произведет поиск каталога memos сначала в текущем каталоге данного пользователя, а если не найдет его — в каталоге /users/steve. Если же и эта попытка окажется неудачной, то поиск будет произведен в каталоге /users/steve/docs, как в последнем прибежище искомого каталога. Если указанный каталог найден не относительно текущего каталога, то команда cd выведет полный путь к нему, чтобы пользователю стало понятно, куда именно он перешел.

```
$ cd /users/steve
$ cd memos
/users/steve/docs/memos
$ cd bin
/users/steve/bin
$ pwd
/users/steve/bin
$
```

Как и в переменной PATH, употреблять точку для обозначения текущего каталога в переменной CDPATH совсем не обязательно. Поэтому следующий путь:

```
:/users/steve:/users/steve/docs
```

равнозначен приведенному ниже пути

```
.:/users/steve:/users/steve/docs
```

Благоразумное использование переменной CDPATH позволяет немало сэкономить на наборе путей к каталогам, особенно если иерархия каталогов оказывается довольно глубокой и приходится часто перемещаться не только в этой, но и в других иерархиях. В отличие от переменной PATH, текущий каталог, вероятнее, всего следует указать в переменной CDPATH первым. Это дает возможность употреблять переменную CDPATH самым естественным образом. Если же текущий каталог не указан в ней первым, он может в конечном итоге оказаться неожиданным!

И еще один момент: переменная CDPATH не устанавливается, когда вы входите в систему. Вам придется устанавливать в ней явным образом последовательный ряд каталогов, в которых оболочка будет искать указанное имя файла.

Дополнительные сведения об подоболочках

Как известно, ни значение переменной, ни текущий каталог из родительской оболочки не могут быть изменены в подоболочке. Допустим, требуется написать программу для установки значений в ряде переменных, которыми необходимо воспользоваться после входа в систему. В качестве примера рассмотрим файл vars следующей программы:

```
$ cat vars
BOOK=/users/steve/book
UUPUB=/usr/spool/uucppublic
DOCS=/users/steve/docs/memos
DB=/usr2/data
$
```

Если вызвать программу vars, значения, присвоенные указанным в ней переменным, по существу, исчезнут по окончании данной программы, поскольку она выполняется в подоболочке, как показано ниже. И в этом нет ничего удивительного.

```
$ vars
$ echo $BOOK

$
```

Команда .

Чтобы разрешить упомянутую выше дилемму, в оболочке имеется команда-точка (.), общая форма которой приведена ниже.

файл

Назначение этой команды — выполнить содержимое указанного *файла* в *текущей* оболочке. Это означает, что команды из указанного *файла* выполняются в текущей оболочке так, как будто они были просто набраны в командной строке, но не в подоболочке. Переменная PATH употребляется в оболочке для поиска указанного *файла*, как и при выполнении любых других программ.

```
$ . vars                    Выполнить программу vars в текущей оболочке
$ echo $BOOK
/users/steve/book           Ура!
$
```

Для выполнения программы подоболочка не порождается, и поэтому любая переменная, которой присваивается значение, сохраняется даже по завершении программы. Так, если имеется программа db со следующими командами:

```
$ cat db
DATA=/usr2/data
RPTS=$DATA/rpts
BIN=$DATA/bin

cd $DATA
$
```

то в результате выполнения программы db по команде-точке будет получен следующий любопытный результат:

```
$ pwd
/users/steve
$ . db
$
```

На этот раз в данной программе определяются три переменные, DATA, RPTS и BIN, в текущей оболочке и происходит переход к каталогу, указанному в переменной DATA:

```
$ pwd
/usr2/data
$
```

Работая над несколькими проектами, можно создавать программы вроде db для специальной настройки своей среды нужным образом. В такую программу можно также включить определения других переменных, изменить приглашения и сделать многое другое. Например, может возникнуть потребность заменить на DB приглашение в переменной PS1, чтобы знать, что переменные базы данных были установлены; ввести в переменную PATH каталог, где находятся программы, связанные с базой данных; а в переменной указать соответствующие каталоги, чтобы сделать их легко доступными по команде cd.

С другой стороны, если внести изменения подобного рода, то может возникнуть потребность выполнить программу db в подоболочке, а не в текущей

оболочке. Ведь в последнем случае все видоизмененные переменные останутся по завершении работы с базой данных.

Самое лучшее — запустить *новую* оболочку из подоболочки со всеми видоизмененными переменными и обновленными в среде установками. А по завершении работы можно выйти из новой оболочки, нажав комбинацию клавиш <Ctrl+D>. Рассмотрим, как этот прием действует на примере приведенной ниже новой версии программы db.

```
$ cat db
#
# Установить и экспортировать переменные,
# связанные с базой данных
#

HOME=/usr2/data
BIN=$HOME/bin
RPTS=$HOME/rpts
DATA=$HOME/rawdata

PATH=$PATH$BIN
CDPATH=:$HOME:$RPTS

PS1="DB: "

export HOME BIN RPTS DATA PATH CDPATH PS1

#
# Запустить новую оболочку
#

/bin/sh
$
```

Сначала в переменной HOME устанавливается начальный каталог /usr2/data, а затем в переменных BIN, RPTS и DATA определяются каталоги относительно начального каталога в переменной HOME, что целесообразно в том случае, если переместить структуру каталогов в какое-нибудь другое место. Для этого достаточно изменить в программе значение переменной HOME. Затем вносятся изменения в переменную PATH, чтобы включить в путь каталог bin с базой данных, а в переменной CDPATH установить путь для поиска в текущем каталоге, начальном каталоге, указанном в переменной HOME, а также в каталоге, указанном в переменной RPTS, где могут содержаться и подкаталоги.

После экспорта этих переменных вызывается стандартная оболочка /bin/sh. С этой точки зрения в новой оболочке обрабатываются любые введенные пользователем команды до тех пор, пока пользователь не введет команду exit или нажмет комбинацию клавиш <Ctrl+D>. После выхода из оболочки управление возвращается программе db, которая, в свою очередь, возвращает управление исходной оболочке.

```
$ db                        Выполнить программу
DB: echo $HOME
/usr2/data
DB: cd rpts                 Попробовать воспользоваться переменной CDPATH
/usr2/data/rpts             Действует!
DB: ps                      Выяснить, какие процессы выполняются
PID TTY TIME COMMAND
123 13 0:40 sh              Исходная оболочка
761 13 0:01 sh              Подоболочка, выполняющая программу db
765 13 0:01 sh              Новая оболочка, выполняемая из программы db
769 13 0:03 ps
DB: exit                    Завершить пока что программу
$ echo $HOME
/users/steve                Вернуться в нормальный режим
$
```

Процесс выполнения программы db наглядно демонстрируется на рис. 10.7, где ради простоты показаны только экспортируемые переменные, представляющие наибольший интерес, а не все переменные, существующие в среде.

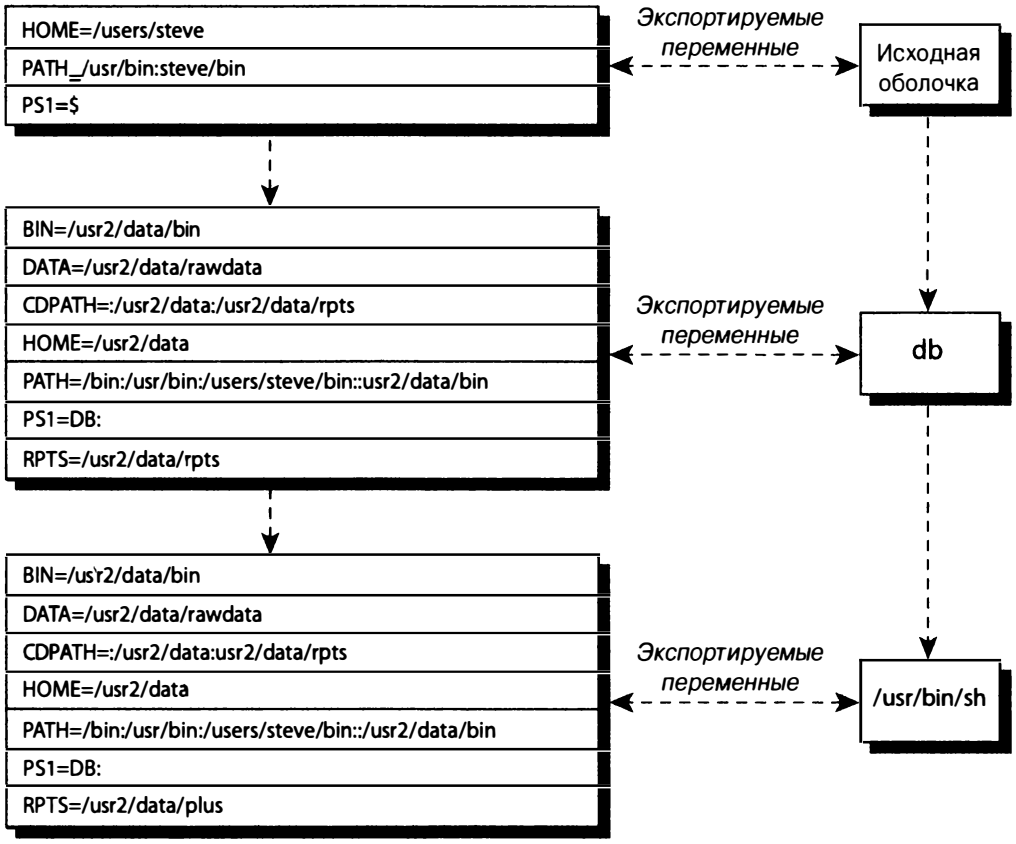


Рис. 10.7. Процесс выполнения программы **db**

Команда exec

Как только процесс оболочки завершится в программе db, завершится и все остальное, о чем свидетельствует отсутствие в данной программе команд после оболочки /bin/sh. Вместо того чтобы ожидать в программе db завершения подоболочки, можно воспользоваться командой exec для замены текущей программы (db) новой программой (/bin/sh). Ниже приведена общая форма команды exec.

```
exec программа
```

Программа, выполняемая по команде exec, заменяет текущую программу, и поэтому в системе остается на один выполняемый процесс меньше, что помогает ей работать быстрее. Программа, выполняемая по команде exec, запускается быстрее благодаря особенностям выполнения процессов в системах Unix.

Чтобы воспользоваться командой exec в программе db, достаточно заменить последнюю строку в данной программе следующей строкой:

```
exec /bin/sh
```

После выполнения команды в этой строке программа db будет заменена оболочкой /bin/sh. Это означает, что теперь не имеет никакого смысла выполнять какие-нибудь команды после команды exec, поскольку они вообще не будут выполнены.

Командой exec можно также воспользоваться для закрытия и повторного открытия стандартного вывода с любым файлом, содержимое которого требуется прочитать. Чтобы заменить стандартный ввод, например, на *входной_файл*, можно воспользоваться приведенной ниже формой команды exec. Любые последующие команды, читающие данные из стандартного ввода, будут читать их из указанного *входного_файла*.

```
exec < входной_файл
```

Переадресация стандартного вывода выполняется аналогичным образом. Например, по команде

```
exec > report
```

весь последующий вывод стандартно переадресовывается в заданный файл report. Следует, однако, иметь в виду, что в обоих приведенных выше примерах команда применялась не для запуска новой программы на выполнение, а лишь для переназначения стандартного ввода или вывода.

Если стандартный ввод переназначается по команде exec, а в дальнейшем его требуется переназначить в каком-нибудь другом месте, для этого достаточно вызвать команду exec снова. Так, для переназначения стандартного ввода обратно

с терминала достаточно ввести приведенную ниже команду. Аналогичным образом переназначается и стандартный вывод.

```
exec < /dev/tty
```

Конструкции (. . .) и { . . . ; }

Иногда требуется сгруппировать вместе ряд команд. Например, программу `plotdata` требуется выполнить в фоновом режиме после программы `sort`. Но эта задача никак не связана с каналом, а требует лишь выполнить ее одну за другой.

Оказывается, что ряд команд можно сгруппировать вместе, заключив их в круглые или фигурные скобки. Первая конструкция приводит к выполнению команд в подоболочке, а вторая конструкция — в текущей оболочке. Ниже приведены некоторые примеры, демонстрирующие принцип действия этих конструкций.

```
$ x=50
$ (x=100)              Выполнить эту строку в подоболочке
$ echo $x
50                     Ничего не изменилось
$ { x=100; }           Выполнить эту строку в текущей оболочке
$ echo $x
100
$ pwd                  Определить местонахождение
/users/steve
$ (cd bin; ls)         Перейти в каталог bin и выполнить команду ls
add
greetings
lu
number
phonebook
rem
rolo
$ pwd
/users/steve           Без изменений
$ { cd bin; }          Это должно привести к смене каталога
$ pwd
/users/steve/bin
$
```

Если все команды, заключаемые в фигурные скобки, должны располагаться в одной строке, после пробела следует указать левую квадратную скобку, а точку с запятой — вслед за последней командой. Обратите в этой связи особое внимание на конструкцию `{ cd bin; }` в приведенных выше примерах.

Иначе действуют круглые скобки. Как демонстрируется в следующем примере, круглые скобки удобны для выполнения команд, не оказывая влияние на текущую среду:

```
(cd bin; ls)
```

Круглыми скобками можно пользоваться и в других целях, перенося, например, команды в фоновый режим:

```
$ (sort 2016data -0 2016data; plotdata 2016data) &
[1] 3421
$
```

Круглые скобки группируют команды `sort` и `plotdata` вместе, а следовательно, эти команды можно перенести в фоновый режим, сохранив порядок их выполнения. Ввод и вывод можно также направить по каналу или переадресовать как из этих конструкций, так и в них, что очень удобно для программирования на языке оболочки.

В следующем примере команда `.ls 2`, по существу, добавляется в начале файла `memo`, прежде чем быть переданной команде `nroff` на обработку:

```
$ { echo ".ls 2"; cat memo; } | nroff -Tlp | lp
```

А в следующей последовательности все сообщения, направляемые из трех программ в стандартный вывод ошибок, накапливаются в файл `errors`:

```
$ { prog1; prog2; prog3; } 2> errors
```

В качестве завершающего примера вернемся к программе `waitfor`, рассмотренной в главе 8. Напомним, что эта программа служит для периодической проверки регистрации пользователя, указанного в системе. Там же упоминалось, что было бы также неплохо, если бы данная программа была способна "переноситься" в фоновый режим. Теперь понятно, как это сделать. Для этого следует заключить цикл `until` и последующие команды в круглые скобки и перенести всю эту группу в фоновый режим, как показано ниже.

```
$ cat waitfor
#
# Ожидать до тех пор, пока указанный пользователь
# не зарегистрируется в системе - версия 4
#
# Установить значения по умолчанию
#

mailopt=FALSE
interval=60

# Обработать параметры команды

while getopts mt: option
do
      case "$option"
      in
            m) mailopt=TRUE;;
            t) interval=$OPTARG;;
```

```
            \?) echo "Usage: mon [-m] [-t n] user"
                echo" -m means to be informed by mail"
                echo" -t means check every n secs."
                exit 1;;
      esac
done

# Убедиться, что указано имя пользователя

if [ "$OPTIND" -gt "$#" ] ; then
      echo "Missing user name!"
      exit 2
fi

shiftcount=$(( OPTIND - 1 ))
shift $shiftcount
user=$1

#
# Перенести все последующие команды в фоновый режим
#

(
      #
      # Проверять, зарегистрировался ли
      # указанный пользователь в системе
      #

      until who | grep "^$user " > /dev/null
      do
            sleep $interval
      done

      #
      # Если программа дойдет до этого места, значит,
      # указанный пользователь зарегистрировался в системе
      #

      if [ "$mailopt" = FALSE] ; then
            echo "$user has logged on"
      else
            runner=$(who am i | cut -c1-8)
            echo "$user has logged on" | mail $runner
      fi
) &
$
```

В действительности данную программу можно было бы полностью заключить в круглые скобки. Но мы решили организовать проверку и синтаксический анализ аргументов прежде, чем переносить остальную часть программы на выполнение в фоновый режим. Как демонстрируется в следующем примере, идентификатор

процесса не выводится оболочкой, когда команда переносится из программы оболочки в фоновый режим:

```
$ waitfor fred
$                          Приглашение возвращается, а следовательно,
                           можно продолжить работу
   ...
fred has logged on
```

Другой способ передачи переменных подоболочке

Если значение переменной требуется передать подоболочке, это можно сделать другим способом, помимо экспорта переменной. Для этого достаточно предварить имя команды или программы в командной строке операцией присваивания значений одной или нескольким переменным. Так, в следующем примере:

```
DBHOME=/uxn2/data DBID=452 dbrun
```

переменные DBHOME и DBID сначала размещаются с присвоенными им значениями в среде команды dbrun, а затем выполняется сама команда dbrun. Но эти переменные не будут доступны в текущей оболочке, поскольку они специально созданы для выполнения команды dbrun.

В действительности приведенный выше пример, где команда предваряется переменными с присвоенными им значениями, подобен составлению следующей конструкции:

```
(DBHOME=/uxn2/data; DBID=452; export DBHOME DBID; dbrun)
```

Ниже приведен краткий пример применения такого способа передачи переменных подоболочке.

```
$ cat foo1
echo :$x:
foo2
$ cat foo2
echo :$x:
$ foo1
::
::                  Переменная x недоступна в программе foo1 или foo2
$ x=100 foo1        Попробовать передать ее таким способом
:100:               Теперь переменная x доступна программе foo1
:100:               и в ее подоболочках
$ echo :$x:
::                  Но она по-прежнему недоступна в текущей оболочке
$
```

А в остальном переменные, определяемые подобным способом, действуют как обычные переменные, экспортируемые в подоболочку. Но они исчезают из вызывающей оболочки после выполнения строки кода, в которой определяются.

Файл .profile

В главе 2 пояснялась последовательность входа в систему, выполняемая прежде, чем оболочка отобразит приглашение на ввод команд. Но перед выводом такого приглашения оболочка производит поиск в системе и чтение двух специальных файлов.

Первым из них является файл /etc/profile, настраиваемый системным администратором. Как правило, в этом файле организуется проверка сообщений, поступивших по электронной почте с выводом сообщения "You have mail" (У вас есть почта); задается используемая по умолчанию маска создания файлов пользователем, сокращенно называемая *umask*; устанавливается стандартное значение переменной PATH и делается все остальное, что должно произойти при входе пользователя в систему, как считает ее администратор.

Но еще больший интерес представляет второй файл .profile, автоматически выполняемый в начальном каталоге пользователя. В большинстве систем Unix стандартный файл .profile устанавливается при создании учетной записи пользователя. Итак, приступим к анализу его содержимого. Ниже приведен пример довольно скромного файла .profile, где просто устанавливается и экспортируется переменная PATH.

```
$ cat $HOME/.profile
PATH="/bin:/usr/bin:/usr/lbin:.:"
export PATH
$
```

У вас есть возможность внести изменения в файл .profile, чтобы ввести команды, которые требуется выполнить при входе в систему, включая и каталог, с которого следует начать работу, проверку пользователей, зарегистрированных в системе, а также установку любых предпочитаемых вами псевдонимов системы. В этот файл можно даже ввести команды, переопределяющие установки — как правило, переменных окружения, задаваемых в файле /etc/profile.

А если у вас имеется возможность изменить текущий каталог в файле .profile, то и не удивительно, что исходная оболочка выполняет оба упомянутых выше файла по следующим командам:

```
$ . /etc/profile
$   .profile
$
```

как только вы войдете в систему. Это также означает, что изменения, которые вы вносите в файл .profile для настройки своей среды, остаются до тех пор, пока вы не выйдете из оболочки.

Большинство пользователей Unix обращаются к своему файлу .profile, чтобы внести немало изменений в свою среду командной строки. В качестве примера

ниже приведен пример файла .profile, где устанавливается переменная PATH, включая каталог текущего пользователя bin; задается переменная CDPATH; изменяются первостепенные и вспомогательные приглашения на ввод команд; изменяется символ стирания на "забой" (комбинацию клавиш <Ctrl+h>) по команде stty; а также выводится приветственное сообщение с помощью программы greetings, упоминавшейся в главе 7.

```
$ cat $HOME/.profile
PATH=/bin:/usr/bin:/usr/lbin:$HOME/bin:.:
CDPATH=.:$HOME:$HOME/misc:$HOME/documents

PS1="=> "
PS2="====> "

export PATH CDPATH PS1 PS2

stty echoe erase CTRL-h

echo
greetings
$
```

Ниже показано, как выглядит последовательность входа в систему с помощью приведенного выше файла .profile.

```
login: steve
Password:
Good morning            Вывод приветствия из программы greetings
=>                      Новое приглашение из переменной PS1
```

Переменная TERM

И хотя многие команды и программы (вроде ls и echo) в системе Unix опираются на режим командной строки, имеется целый ряд полноэкранных команд (вроде vi), требующих досконального знания настроек и возможностей терминала. Подобная информация хранится в переменной окружения TERM и обычно не требует вашего внимания, поскольку программа Terminal или SSH автоматически устанавливает в этой переменной такое оптимальное значение, чтобы все работало без сучка и задоринки.

Тем не менее некоторые пользователи по старинке считают, что в переменной TERM требуется установить конкретное значение вроде ansi, vt100 или xterm, чтобы полноэкранные программы работали надлежащим образом. В подобных случаях переменную TERM рекомендуется настраивать в файле .profile.

Написав несложный блок кода, можно даже вывести приглашение на ввод значения переменной TERM в процессе регистрации в системе, как показано ниже.

```
echo "What terminal are you using (xterm is the default)? \c"
read TERM
if [ -z "$TERM" ]
then
      TERM=xterm
fi
export TERM
```

В зависимости от введенного типа терминала, может возникнуть потребность выполнить такие действия, как, например, установка функциональных клавиш или знаков табуляции на терминале. Даже если вы всегда пользуетесь одним и тем же типом терминала, установите переменную `TERM` в своем файле `.profile`.

Любопытно, что в системах Mac OS X и Ubuntu Linux для настройки программы Terminal в переменной `TERM` по умолчанию указывается значение `xterm-256color`, а в системе Solaris Unix — значение `vt100`. Во многих сторонних (терминальных) программах вроде `telnet/SSH` в переменной `TERM` устанавливается значение `ansi`.

Переменная `TZ`

Переменная `TZ` применяется в команде `date` и некоторых стандартных функциях из библиотеки C для определения текущего часового пояса. Безусловно, пользователи могут удаленно регистрироваться в системе через Интернет, и поэтому вполне возможно, чтобы разные пользователи системы работали в разных часовых поясах. В переменной `TZ` проще всего установить наименование часового пояса с помощью трех или больше буквенных символов и числа, обозначающего количество часов, которые требуется прибавить к местному времени для достижения *всемирного скоординированного времени* (UTC), называемого также временем по Гринвичу. Это число может быть как положительным, когда местный часовой пояс находится на запад от нулевого меридиана, так и отрицательным, когда местный часовой пояс находится на восток от нулевого меридиана. Например, восточное поясное время в Северной Америке может быть указано следующим образом:

```
TZ=EST5
```

Команда `date` вычисляет правильное время, исходя из этой информации, указывая при необходимости наименование часового пояса в выводимом результате, как демонстрируется в следующем примере:

```
$ TZ=EST5 date
Wed Feb 17 15:24:09 EST 2016
$ TZ=xyz3 date
Wed Feb 17 17:46:28 xyz 2016
$
```

После числа, обозначающего часовой пояс, может быть также указано второе наименование часового пояса. И если оно указано, то предполагается летнее время, которое автоматически корректируется в команде `date` при переходе на летнее время, т.е. на один час вперед от стандартного времени. Если после этого числа следует наименование часового пояса с учетом перехода на летнее время, то это значение используется для вычисления летнего времени, исходя из всемирного скоординированного времени, таким же образом, как и описанное выше число.

Чаще всего часовой пояс обозначается как `EST5EDT` или `MST7MDT` с учетом перехода на летнее время, хотя этот переход происходит не во всех странах. Переменная `TZ` обычно устанавливается в файле `/etc/profile` или `.profile`. Если она не установлена, то для конкретной реализации используется стандартный часовой пояс — как правило, всемирное скоординированное время.

Следует также иметь в виду, что для установки часового пояса во многих системах Linux может быть указана конкретная географическая местность. Так, в следующем примере устанавливается текущее время в Тихуане, Мексика:

```
TZ="America/Tijuana" date
```

11

Дополнительные сведения о параметрах

В этой главе приводятся дополнительные сведения о переменных и параметрах. Формально параметры включают в себя аргументы, передаваемые программе (т.е. *позиционные* параметры), специальные переменные оболочки вроде `$#` и `$?`, а также обыкновенные переменные, называемые также *ключевыми* параметрами.

Позиционным параметрам нельзя присваивать значения непосредственно, хотя с помощью команды `set` можно переназначать их значения. Как известно, для присваивания значений переменным служит следующая простая форма:

```
переменная=значение
```

Ниже приведена более общая форма, позволяющая одновременно присваивать значения нескольким переменным.

```
переменная=значение переменная=значение...
```

В следующем примере демонстрируется одновременное присваивание значений трем переменным:

```
$ x=100 y=200 z=50
$ echo $x $y $z
100 200 50
$
```

Подстановка значений параметров

В простейшей форме для подстановки значения параметра его имя предваряется знаком денежной единицы, как, например, `$i` или `$9`.

Конструкция ${*параметр*}

Во избежание возможного конфликта из-за символов, указываемых после имени параметра, последнее можно заключить в фигурные скобки, как показано ниже.

```
mv $file ${file}x
```

По этой команде значение переменной x добавляется в конце имени файла, указанного в переменной `$file`. То же самое нельзя написать так:

```
mv $file $filex
```

поскольку оболочка попытается подставить значение переменной `filex` в качестве второго аргумента, а не имя `file` плюс буква x.

Как упоминалось в главе 6, для доступа к позиционным параметрам, начиная с 10-го и далее, номер требующегося параметра следует заключить в фигурные скобки, используя ту же самую форму обозначения, например `${11}`. Но если заключить имя переменной в фигурные скобки, то с ней можно сделать нечто большее, чем обычно.

Конструкция ${*параметр*:-*значение*}

В данной конструкции предписывается использовать значение указанного *параметра*, если оно не является пустым, а иначе — подставить заданное *значение*. Например, в следующей команде:

```
echo Using editor ${EDITOR:-/bin/vi}
```

оболочка воспользуется значением параметра EDITOR, если оно не является пустым, а иначе — значением `/bin/vi`. Это все равно, что написать следующее:

```
if [ -n "$EDITOR" ]
then
      echo Using editor $EDITOR
      else
      echo Using editor /bin/vi
fi
```

По команде

```
${EDITOR:-/bin/ed} /tmp/edfile
```

запускается программа, имя которой хранится в переменной EDITOR (это, вероятно, текстовый редактор), а если значение переменной EDITOR оказывается пустым, запускается программа, доступная по пути `/bin/ed`.

Однако это не приводит к изменению значения переменной. Поэтому даже после выполнения приведенной выше команды переменная EDITOR будет и далее иметь пустое значение, если оно было таковым изначально. Ниже приведен простой пример, демонстрирующий применение рассматриваемой здесь конструкции.

```
$ EDITOR=/bin/ed
$ echo ${EDITOR:-/bin/vi}
/bin/ed
$ EDITOR=          Установить пустое значение в данной переменной
$ echo ${EDITOR:-/bin/vi}
/bin/vi
$
```

Конструкция ${*параметр*:=*значение*}

Эта конструкция подобна предыдущей, но если заданный *параметр* имеет пустое значение, то указанное *значение* не только используется, но и присваивается заданному *параметру* (обратите внимание на знак **=** в данной конструкции). Тем не менее подобным способом нельзя присваивать значения позиционным параметрам. Это означает, что заданный *параметр* не может иметь числовое значение.

Данная конструкция, как правило, применяется с целью проверить, было ли установлено значение в экспортируемой переменной. И если оно не было установлено, то переменной присваивается значение, используемое по умолчанию. Так, приведенная ниже конструкция означает, что если переменной уже присвоено значение PHONEBOOK, оно остается без изменения, а иначе — в ней устанавливается значение $HOME/phonebook.

```
${PHONEBOOK:=$HOME/phonebook}
```

Следует, однако, иметь в виду, что приведенную выше конструкцию нельзя применять в качестве отдельной команды. Ведь после произведенной подстановки оболочка попытается выполнить результат такой подстановки как команду, что и демонстрируется в следующем примере:

```
$ PHONEBOOK=
$ ${PHONEBOOK:=$HOME/phonebook}
sh: /users/steve/phonebook: cannot execute
$
```

Чтобы воспользоваться данной конструкцией как обособленной командой, следует применить пустую команду. Так, если написать следующее:

```
  ${PHONEBOOK:=$HOME/phonebook}
```

оболочка по-прежнему выполнит постановку, вычислив остальную часть командной строки, но ничего не выполнит, поскольку указана пустая команда. Ниже приведены некоторые примеры такого применения рассматриваемой здесь конструкции.

```
$ PHONEBOOK=
$ : ${PHONEBOOK:=$HOME/phonebook}
$ echo $PHONEBOOK                          Выяснить, присвоено ли
                                           переменной $PHONEBOOK
                                           значение /users/steve/phonebook
$ : ${PHONEBOOK:=foobar}                   Значение переменной $PHONEBOOK
                                           не должно измениться
$ echo $PHONEBOOK
/users/steve/phonebook It didn't
$
```

Но, как правило, в программах оболочки обозначение **:=** применяется для первой ссылки на переменную в условном операторе или команде echo. Оно дает такой же результат, как и пустая команда.

Конструкция ${*параметр*:?*значение*}

Если в этой конструкции заданный *параметр* имеет непустое значение, то оболочка подставит его. В противном случае оболочка направит указанное *значение* в стандартный вывод ошибок и на этом завершит свое выполнение. Но это совсем не означает выход из системы, если данная конструкция была выполнена в исходной оболочке. Если же *значение* не было указано, оболочка направит используемое по умолчанию и приведенное ниже сообщение в стандартный вывод ошибок.

```
prog: parameter: parameter null or not set
(программа: параметр: параметр пуст или не задан)
```

Ниже приведен пример применения данной конструкции.

```
$ PHONEBOOK=
$ : ${PHONEBOOK:?"No PHONEBOOK file"}
No PHONEBOOK file
$ : ${PHONEBOOK:?}                    Значение не задано
sh: PHONEBOOK: parameter null or not set
$
```

С помощью данной конструкции нетрудно проверить, были ли установлены переменные, требующиеся в программе, и не являются ли их значения пустыми:

```
  ${TOOLS:?} ${EXPTOOLS:?} ${TOOLBIN:?}
```

Конструкция ${*параметр*:+*значение*}

В данной конструкции указанное *значение* подставляется, если заданный *параметр* имеет непустое значение, а иначе вообще ничего не подставляется. Таким образом, данная конструкция по своему действию противоположна конструкции ${*параметр*:=*значение*}, как демонстрируется в следующем примере:

```
$ traceopt=T
$ echo options: ${traceopt:+"trace mode"}
options: trace mode
$ traceopt=
$ echo options: ${traceopt:+"trace mode"}
options:
$
```

В той части любой из рассматриваемых здесь конструкций, где указывается *значение*, может быть подставлена команда, поскольку она выполняется только в том случае, если требуется ее значение. На первый взгляд это может показаться несколько сложным, поэтому рассмотрим следующий пример:

```
WORKDIR=${DBDIR:-$(pwd)}
```

В данном примере переменной WORKDIR присваивается значение параметра DBDIR, если оно непустое. В противном случае выполняется команда pwd, а результат присваивается переменной WORKDIR. Таким образом, команда pwd выполняется *лишь* в том случае, если значение параметра DBDIR оказывается пустым.

Конструкции для сопоставления с шаблоном

В оболочке по стандарту POSIX предоставляются четыре конструкции для подстановки параметров, где осуществляется сопоставление с шаблоном. В некоторых довольно старых оболочках подобные средства не поддерживаются, но вам вряд ли придется иметь дело с этими оболочками в современных версиях систем Unix, Linux или Mac OS.

Такие конструкции принимают два аргумента: имя переменной (или номер позиционного параметра) и заданный шаблон. Оболочка производит поиск содержимого указанной переменной на совпадение с заданным шаблоном. Если такое совпадение обнаружено, оболочка воспользуется значением переменной в командной строке, а *совпавшая часть* шаблона будет удалена. Если же совпадение с шаблоном *не* обнаружено, то в командной строке используется все содержимое переменной. Но в обоих случаях содержимое переменной остается без изменения.

Термин *шаблон* употребляется здесь потому, что оболочка допускает пользоваться теми же символами сопоставления с шаблоном, что и при подстановке имен файлов и выбора в ветвях оператора case. К их числу относится знак *****, обозначающий совпадение с нулевым или бо́льшим количеством символов; знак **?**, обозначающий совпадение с любым одиночным символом; последовательность символов **[. . .]**, обозначающая совпадение с любым символом из указанного множества; а также последовательность символов **[! . . .]**, обозначающая совпадение с любым символом, отсутствующим в указанном множестве.

При написании конструкции

```
${переменная%шаблон}
```

оболочка анализирует содержимое указанной *переменной*, чтобы выяснить, *оканчивается* ли оно заданным *шаблоном*. Если это именно так, то используется содержимое указанной *переменной*, где справа удалено самое *краткое* совпадение с заданным *шаблоном*.

Если применяется следующая конструкция:

```
${переменная%%шаблон}
```

то и в этом случае оболочка анализирует содержимое указанной *переменной*, чтобы выяснить, *оканчивается* ли оно заданным *шаблоном*. Но на этот раз из содержимого указанной *переменной* удаляется самое *длинное* совпадение с шаблоном справа. Такая конструкция уместна лишь в том случае, если в заданном

шаблоне употребляется знак *****. В противном случае обе конструкции со знаками **%** и **%%** ведут себя одинаково.

Аналогично приведенная ниже конструкция обусловливает сопоставление с заданным шаблоном, но слева, а не справа. Если обнаружено совпадение с заданным *шаблоном*, в командной строке используется значение указанной *переменной*, где слева удалено совпадение с заданным *шаблоном*.

```
${переменная#шаблон}
```

И наконец, следующая конструкция:

```
${переменная##шаблон}
```

действует аналогично предыдущей конструкции, но на этот раз из содержимого указанной *переменной* удаляется самое длинное совпадение с заданным *шаблоном* слева.

Напомним, что во всех четырех упомянутых выше конструкциях никаких изменений в указанной переменной не происходит. Влияние оказывается только на то, что используется в командной строке. Напомним также, что совпадения с шаблоном *привязываются*. Так, в конструкциях со знаками **%** и **%%** значения переменных должны *оканчиваться* заданным шаблоном. А в конструкциях со знаками **#** и **##** значения переменных должны *начинаться* заданным шаблоном. В ряде приведенных ниже примеров демонстрируется применение рассматриваемых здесь конструкций.

```
$ var=testcase
$ echo $var
testcase
$ echo ${var%e}            Удалить символ e справа
testcas
$ echo $var                Значение переменной не изменяется
testcase
$ echo ${var%s*e}          Удалить самое краткое совпадение справа
testca
$ echo ${var%%s*e}         Удалить самое длинное совпадение
te
$ echo ${var#?e}           Удалить самое краткое совпадение слева
stcase
$ echo ${var#*s}           Удалить самое краткое совпадение слева
$ echo ${var##*s}          Удалить самое длинное совпадение слева
e
$ echo ${var#test}         Удалить строку test слева
case
$ echo ${var#teas}         Совпадение не обнаружено
testcase
$
```

Имеется немало практических примеров применения рассматриваемых здесь конструкций. Так, в следующем примере проверяется, оканчивается ли символами **.o** имя файла, сохраненное в переменной `file`:

```
if [ ${file%.o} != $file ] ; then
      # файл оканчивается на .o

fi
```

В качестве еще одного примера ниже приведена программа оболочки, действующая подобно команде `basename` в системе Unix.

```
$ cat mybasename
echo ${1##*/}
$
```

Эта программа отображает свой аргумент, где удалены все символы вплоть до последнего знака **/**:

```
$ mybasename /usr/spool/uucppublic
uucppublic
$ mybasename $HOME
steve
$ mybasename memos
memos
$
```

Но этим перечень конструкций, доступных в оболочке для подстановки значений параметров, не исчерпывается.

Конструкция ${#*переменная*}

Допустим, требуется выяснить, сколько символов хранится в переменной. С этой целью можно воспользоваться конструкцией ${#*переменная*}, как демонстрируется в следующем примере:

```
$ text='The shell'
$ echo ${#text}
9
$
```

Совет

Все конструкции для подстановки значений параметров, описываемые в этой главе, перечислены в табл. А.3 приложения А.

Переменная $0

Всякий раз, когда выполняется программа оболочки, имя этой программы сохраняется оболочкой в специальной переменной $0. Это может быть удобно в самых разных случаях, в том числе и тогда, когда программа доступна под разными именами команд по жестким ссылкам в файловой системе. Это дает возможность выяснить программным путем, какая именно команда была выполнена.

Эта переменная чаще всего применяется для отображения сообщений об ошибках, поскольку такие сообщения основываются на фактическом имени программы, а не на том, которое жестко закодировано в самой программе. Если имя программы доступно по ссылке на переменную $0, то в результате переименования программы автоматически обновляется сообщение об ошибке, не требуя дополнительного редактирования исходного текста программы, как демонстрируется в следующем примере:

```
$ cat lu
#
# Найти абонента в телефонном справочнике
#

if [ "$#" -ne 1 ] ; then
      echo "Incorrect number of arguments"
      echo "Usage: $0 name"
      exit 1
fi

name=$1
grep "$name" $PHONEBOOK

if [ $? -ne 0 ] ; then
      echo "I couldn't find $name in the phone book"
fi

$ PHONEBOOK=$HOME/phonebook
$ export PHONEBOOK
$ lu Teri
Teri Zak 201-393-6000
$ lu Teri Zak
Incorrect number of arguments
Usage: lu name
$ mv lu lookup              Переименовать программу
$ lookup Teri Zak           Выяснить, что произойдет теперь
Incorrect number of arguments
Usage: lookup name
$
```

В некоторых системах Unix в переменной $0 автоматически сохраняется полный путь к программе, включая и все каталоги, что может привести к появлению неудобочитаемых сообщений об ошибках. В подобных случаях следует воспользоваться конструкцией $(basename $0) или ${0##*/}, чтобы вычленить путь к программе, оставив только ее имя.

Команда set

Доступная в оболочке команда set служит двум целям: для установки различных параметров оболочки и переназначения позиционных параметров $1, $2 и т.д.

Параметр -x

Как упоминалось вкратце в главе 7, команда `sh -x ctype` служит для отладки и устранения ошибок в программах оболочки. А команда `set` позволяет включать и выключать режим трассировки отдельных параметров отлаживаемой программы.

Так, если указать в исходном тексте программы следующую команду:

```
set -x
```

то активизируется режим трассировки, а это означает, что данные обо всех выполняемых далее командах будут направлены оболочкой в стандартный вывод ошибок после имени файла, переменной и подстановки команд, а также осуществляемой переадресации ввода-вывода. Имена трассируемых команд предваряются знаками "плюс", как демонстрируется в следующем примере:

```
$ x=*
$ set -x          Установить режим трассировки команд
$ echo $x
+ echo add greetings lu rem rolo
add greetings lu rem rolo
$ cmd=wc
+ cmd=wc
$ ls | $cmd -l
+ ls
+ wc -l
      5
$
```

Режим трассировки можно выключить в любой момент, просто выполнив команду `set` с параметром **+x**:

```
$ set +x
+ set +x
$ ls | wc -l
           5              Вернуться в обычный режим
$
```

Следует, однако, иметь в виду, что режим трассировки *не* передается подоболочке. Тем не менее трассировку самой подоболочки можно осуществить, указав имя программы после параметра **-x** команды `sh` следующим образом:

```
sh -x rolo
```

или введя последовательный ряд команд `set -x` и `set +x` в исходный текст самой программы. На самом деле в исходном тексте программы можно указать любое количество команд `set -x` и `set +x`, чтобы включать и выключать режим трассировки по мере надобности!

Команда `set` без аргументов

Если вызвать команду `set` без аргументов, то в конечном итоге будет получен упорядоченный в алфавитном порядке список всех переменных (локальных или экспортируемых), существующих в текущей среде, как демонстрируется в следующем примере:

```
$ set                          Показать все переменные
CDPATH=:/users/steve:/usr/spool
EDITOR=/bin/vi
HOME=/users/steve
IFS=

LOGNAME=steve
MAIL=/usr/spool/mail/steve
MAILCHECK=600
PATH=/bin:/usr/bin:/users/steve/bin:.:
PHONEBOOK=/users/steve/phonebook
PS1=$
PS2=>
PWD=/users/steve/misc
SHELL=/usr/bin/sh
TERM=xterm
TMOUT=0
TZ=EST5EDT
cmd=wc
x=*
$
```

Переназначение позиционных параметров с помощью команды `set`

Напомним, что присвоить новое значение позиционному параметру или переназначить его никак нельзя. Попытку переназначить, например, значение **100** позиционного параметра `$1` логически можно обозначить следующим образом:

```
1=100
```

Но из этого ничего не выйдет. Позиционные параметры устанавливаются после вызова программы оболочки.

Тем не менее, чтобы изменить значение позиционного параметра, можно воспользоваться следующим ловким приемом: если слова задаются в команде `set` как аргументы командной строки, эти слова будут присвоены позиционным параметрам `$1`, `$2` и т.д. А предыдущие значения, хранившиеся в позиционных параметрах, будут утрачены. В таком случае по следующей команде из программы оболочки:

```
set a b c
```

аргумент `a` будет присвоен позиционному параметру `$1`, аргумент `b` — позиционному параметру `$2`, тогда как аргумент `c` — позиционному параметру `$3`. Ниже приведен более сложный пример переназначения позиционных параметров.

```
$ set one two three four
$ echo $1:$2:$3:$4
one:two:three:four
$ echo $#           Эта ссылка должна быть на значение 4
4
$ echo $*           А на что делается эта ссылка?
one two three four
$ for arg; do echo $arg; done
one
two
three
four
$
```

После выполнения команды `set` все действует, как и предполагалось. В частности, переменные `$#`, `$*` и цикл `for` без списка — все эти элементы отражают изменения в значениях позиционных параметров.

Команда `set` нередко применяется подобным образом для "синтаксического анализа" данных читаемых из файла или терминала. В качестве примера ниже приведена программа `words`, подсчитывающая количество слов, набранных в строке (т.е. "слов" в том смысле, в каком они определяются в оболочке).

```
$ cat words
#
# Подсчитать слова в строке
#

read line
set $line
echo $#
$ words                          Выполнить программу
Here's a line for you to count.
7
$
```

Данная программа читает вводимые пользователем данные, сохраняя сначала прочитанную строку в переменной оболочки `line`, а затем выполняя следующую команду:

```
set $line
```

В итоге каждое слово, сохраняемое в переменной `line`, присваивается соответствующему позиционному параметру. А в переменной `$#` устанавливается значение, обозначающее количество присвоенных слов, которое равно количеству слов в прочитанной строке.

Параметр --

Описанный выше способ вполне работоспособен. Но что, если пользователь начнет по какой-нибудь причине ввод данных со знака **-?** Рассмотрим следующий пример:

```
$ words
-1 + 5 = 4
words: -1: bad option(s)
$
```

В данном случае произошло следующее: после того, как введенная пользователем строка была прочитана и присвоена переменной `line`, в оболочке была выполнена приведенная ниже команда.

```
set $line
```

А после подстановки значения переменной `line` эта команда приобрела следующий вид:

```
set -1 + 5 = 4
```

В ходе выполнения данной команды оболочка обнаружила знак **-** и восприняла его как неверно введенный параметр **-1**. Именно этим и объясняется появление приведенного выше сообщения об ошибке.

Еще один недостаток программы `words` проявляется в том случае, если передать ей строку, полностью состоящую из символов пробела или пустую строку, как демонстрируется в следующем примере:

```
$ words
            Просто нажать клавишу <Enter>
CDPATH=.:/users/steve:/usr/spool
EDITOR=/bin/vi
HOME=/users/steve
IFS=

LOGNAME=steve
MAIL=/usr/spool/mail/steve
MAILCHECK=600
PATH=/bin:/usr/bin:/users/steve/bin:.:
PHONEBOOK=/users/steve/phonebook
PS1=$
PS2=>
PWD=/users/steve/misc
SHELL=/usr/bin/sh
TERM=xterm
TMOUT=0
TZ=EST5EDT
cmd=wc
x=*
0
$
```

В данном случае оболочка обнаружила команду `set` без аргументов. Следовательно, она вывела список всех переменных, установленных в среде данного пользователя.

Чтобы застраховаться от подобных осложнений, команду set следует применять с параметром **--**, который предписывает ей интерпретировать в качестве параметров любые последующие знаки дефиса или слова, обнаруживаемые в командной строке в форме аргументов. Такая мера предохраняет также от вывода всех переменных, установленных в среде данного пользователя, если в командной строке отсутствуют аргументы, как это произошло в приведенном выше примере, где была введена пустая строка.

Таким образом, в строку с командой set в программе words необходимо внести следующее изменение:

```
set -- $line
```

Если дополнить программу words циклом while, где выполняется вполне определенная целочисленная арифметическая операция, как показано ниже, эта программа сможет подсчитывать общее количество слов, прочитанных из стандартного ввода. По существу, она станет разновидностью команды wc -w.

```
$ cat words
#
# Подсчитать все слова, прочитанные из стандартного ввода
#

count=0
while read line
do
      set -- $line
      count=$(( count + $# ))
done

echo $count
$
```

Прочитав каждую строку из стандартного ввода, команда set присвоит новые данные из этой строки позиционным параметрам, а в переменной $# установится количество слов в данной строке. Параметр **--** указан в команде set на тот случай, если читаемые строки будут начинаться со знака **-**, окажутся пустыми или не будут вообще состоять из буквенно-цифровых символов.

Далее значение переменной $# прибавляется к значению переменной count и читается следующая строка. После выхода из цикла в связи с обнаружением конца файла отображается значение переменной count, т.е. общее количество прочитанных слов. Ниже демонстрируется применение новой версии программы words в сравнении с командой wc -w.

```
$ words < /etc/passwd
567
$ wc -w < /etc/passwd              Проверить полученный результат
                                   с помощью команды wc
567
$
```

Безусловно, в программе words применяется довольно необычный способ подсчета количества слов в файле. Но в то же время в ней демонстрируется, насколько универсальной может быть команда set, о чем известно далеко не всем пользователям систем Unix. В качестве еще одного примера применения данной команды ниже приведен краткий способ подсчета количества файлов в каталоге.

```
$ set *
$ echo $#
8
$
```

Такой способ позволяет намного быстрее подсчитать количество файлов в каталоге, чем следующая последовательность команд:

```
ls | wc -l
```

поскольку в нем применяются только команды, встроенные в оболочку. В целом программы оболочки будут выполняться намного быстрее, если постараться реализовать в них как можно больше операций с помощью команд, встроенных в оболочку.

Другие параметры команды set

Команда set принимает ряд других параметров, каждый из которых активизируется предваряющим знаком **-** и дезактивируется предваряющим знаком **+**. Чаще всего применяется рассмотренный выше параметр **-x**, а остальные параметры данной команды перечислены в табл. А.9 приложения А.

Переменная IFS

В оболочке имеется специальная переменная IFS, сокращенное имя которой обозначает *внутренний разделитель полей* (internal field separator). Значение этой переменной употребляется в оболочке при синтаксическом анализе ввода из команды read, вывода из команды, подставляемой с помощью механизма обратных кавычек, а также при подставке значений переменных. Короче говоря, переменная IFS содержит ряд символов, употребляемых в качестве пробельных разделителей. Если эта переменная введена в командной строке, оболочка интерпретирует ее значение как пробельный символ, обыкновенно применяемый для разделения слов.

Приведенное выше пояснение назначения переменной IFS может показаться не вполне вразумительным! Следовательно, чтобы выяснить, какие же именно символы хранятся в данной переменной, ее значение достаточно передать по каналу из команды echo команде od (т.е. восьмеричный вывод содержимого) с параметром **-b** (т.е. побайтовое отображение), как показано ниже.

```
$ echo "$IFS" | od -b
0000000 040 011 012 012
0000004
$
```

В первом столбце приведенных выше чисел указано смещение относительно начала ввода. А последующие числа являются восьмеричными эквивалентами символов, прочитанных командой od. Первым их них указано восьмеричное число **040**, обозначающее символ пробела в коде ASCII. Далее следует восьмеричное число **011**, обозначающее символ табуляции в коде ASCII, а затем восьмеричное число **012**, обозначающее символ новой строки в том же самом коде. После этого следует еще один символ новой строки, добавленный командой echo. Нет ничего удивительного в том, что именно этот ряд символов содержится в переменной IFS. Ведь все они относятся к категории пробельных символов, не раз упоминавшихся в данной книге.

Как отмечалось ранее, оболочка удаляет начальные пробельные символы из любой строки, прочитанной командой read. Но если установить в переменной IFS только символ новой строки перед выполнением команды read, то начальный символ новой строки останется в прочитанной строке, поскольку оболочка не воспримет его как разделитель полей.

```
$ read line                 Выполнить команду read "старым" способом
          Here's a line
$ echo "$line"
Here's a line
$ IFS="
> "                         Установить в переменной IFS только
                            знак новой строки
$ read line                 Выполнить команду read снова
          Here's a line
$ echo "$line"
          Here's a line                 Начальные пробелы остались
$
```

Чтобы установить в переменной IFS только знак новой строки, в данном примере была введена открывающая кавычка, после которой сразу же была нажата клавиша <Enter>, а с новой строки — закрывающая кавычка. Между этими кавычками никаких дополнительных символов не вводится, поскольку они будут сначала сохранены в переменной IFS, а затем употреблены в оболочке.

А теперь попробуем установить в переменной IFS нечто более видимое — знак двоеточия:

```
$ IFS=:
$ read x y z
123:345:678
$ echo $x
123
```

```
$ echo $z
678
$ list="one:two:three"
$ for x in $list; do echo $x; done
one
two
three
$ var=a:b:c
$ echo "$var"
a:b:c
$
```

В первом из приведенных выше примеров в переменной IFS был установлен знак двоеточия, и поэтому, прочитав строку, оболочка сначала разделила ее на три слова (**123**, **345** и **678**), а затем сохранила их в переменных x, y и z. В следующем примере переменная IFS была использована при подстановке значения переменной list в цикле for. А в последнем примере демонстрируется тот факт, что переменная IFS не используется, когда переменной var присваивается значение.

Изменение значения в переменной IFS нередко выполняется вместе с командой set, как демонстрируется в следующем примере:

```
$ line="Micro Logic Corp.:Box 174:Hackensack, NJ 07602"
$ IFS=:
$ set $line
$ echo $#                    Сколько параметров было установлено?
3
$ for field; do echo $field; done
Micro Logic Corp.
Box 174
Hackensack, NJ 07602
$
```

Такой способ оказывается эффективным потому, что в нем применяются все встроенные в оболочку команды, заметно ускоряющие выполнение программ. Именно этот способ применяется в окончательной версии программы rolo, представленной в главе 13.

Ниже приведена программа, которая называется number2 и является окончательной версией программы number, предназначенной для нумерации строк и представленной в главе 9. В этой программе введенные строки направляются в стандартный вывод с предшествующим номером строки, а содержимое переменной IFS видоизменяется с целью сохранить и воспроизвести начальные пробелы и остальные пробельные символы, в отличие от предыдущей версии данной программы. Обратите также внимание на применение команды printf для выравнивания номеров строк по правому краю.

```
$ cat number2
#
# Пронумеровать строки, введенные из файла, заданного
# в качестве аргумента, или из стандартного ввода, если
# файл не указан — окончательная версия
#
# Видоизменить содержимое переменной IFS, чтобы
# сохранить начальные пробелы при вводе строк

IFS='
'   # В кавычки заключается лишь знак новой строки

lineno=1

cat $* |
while read -r line
do
      printf "%5d:%s\n" $lineno "$line"
      lineno=$(( lineno + 1 ))
done
```

Ниже приведен пример выполнения программы `number2`.

```
$ number2 words
    1:#
    2:# Подсчитать все слова, прочитанные из стандартного ввода
    3:#
    4:
    5:count=0
    6:while read line
    7:do
    8:  set -- $line
    9:  count=$(( count + $# ))
    10:done
    11:
    12:echo $count
$
```

Значение переменной `IFS` оказывает влияние на порядок интерпретации вводимых данных в оболочке. И если его требуется изменить в программе, то целесообразно сохранить прежнее значение данной переменной (например, в переменной `OIFS`), а затем восстановить его по окончании соответствующих операций.

Команда `readonly`

Эта команда применяется для обозначения тех переменных, значения которых не подлежат последующему изменению. Например, в следующей команде:

```
readonly PATH HOME
```

переменные PATH и HOME обозначаются как доступные только для чтения. При последующих попытках присвоить значение любой из этих переменных оболочка выдаст сообщение об ошибке, как демонстрируется в следующем примере:

```
$ PATH=/bin:/usr/bin:.:
$ readonly PATH
$ PATH=$PATH:/users/steve/bin
sh: PATH: is read-only
$
```

В данном примере демонстрируется следующее: после того, как переменная PATH была сделана доступной только для чтения, оболочка вывела сообщение об ошибке при попытке присвоить ей значение. Чтобы перечислить переменные, доступные только для чтения, достаточно ввести команду `readonly -p` без аргументов:

```
$ readonly -p
readonly PATH=/bin:/usr/bin:.:
$
```

Атрибут "только для чтения" не передается от переменной к подоболочкам. А после того, как переменная станет доступной в оболочке только для чтения, "отменить" этот ее атрибут нельзя никоим образом.

Команда `unset`

Иногда требуется удалить определение переменной из своей среды. С этой целью достаточно ввести команду `unset`, указав далее имена переменных:

```
$ x=100
$ echo $x
100
$ unset x              Удалить переменную x из текущей среды
$ echo $x
$
```

Но по команде `unset` из среды нельзя удалить переменную, доступную только для чтения. Более того, из среды нельзя удалить переменные IFS, MAILCHECK, PATH, PS1 и PS2.

12

Невыясненные вопросы

В этой главе рассматриваются команды и средства оболочки, не охваченные в предыдущих главах. Они не рассматриваются здесь в каком-то обоснованном порядке, и поэтому вы вольны воспользоваться материалом этой главы, чтобы расширить свои знания приемов и методов программирования на языке оболочки.

Команда `eval`

В этом разделе описывается `eval` — одна из самых необычных команд оболочки. Ее общая форма выглядит следующим образом:

```
eval командная_строка
```

где *командная_строка* — это обычная командная строка, вводимая с терминала. Но если указать такую командную строку после команды `eval`, то оболочка просмотрит ее *дважды*, прежде чем выполнять ее. Это может быть удобно, если в сценарии составляется команда, которую, помимо всего прочего, требуется вызвать.

В простейшем случае применение команды `eval` не дает никакого эффекта, как демонстрируется в следующем примере:

```
$ eval echo hello
hello
$
```

А теперь рассмотрим следующий пример, где команда `eval` *не* применяется:

```
$ pipe="|"
$ ls $pipe wc -l
|: No such file or directory
wc: No such file or directory
-1: No such file or directory
$
```

Сообщения об ошибках выводятся из команды `ls` потому, что значение переменной `pipe` и результат последующего вызова команды `wc -l` интерпретируются как аргументы команды `ls`. Оболочка берет на себя заботу о каналах и переадресации ввода-вывода *перед* подстановкой значения переменной, и поэтому она вообще не интерпретирует символ канала в переменной `pipe`. Но если указать

команду `eval` перед рассматриваемой здесь последовательностью команд, то можно добиться желаемого результата:

```
$ eval ls $pipe wc -l
16
$
```

Когда оболочка просматривает командную строку в первый раз, она подставляет знак **|** в качестве значения переменной `pipe`. А команда `eval` вынуждает оболочку просмотреть командную строку во второй раз. В этот момент знак **|** распознается оболочкой как символ канала, и все остальное продолжается далее, как и предполагалось.

Команда `eval` нередко применяется в тех программах оболочки, где командные строки составляются в одной или нескольких переменных. И команда `eval` оказывается весьма полезной в том случае, если эти переменные содержат любые символы, которые интерпретируются в оболочке. В частности, ограничители команд (**;**, **|**, **&**), знаки переадресации ввода-вывода (**<**, **>**), а также знаки кавычек должны быть указаны непосредственно в командной строке, чтобы иметь особое назначение в оболочке.

В качестве следующего примера рассмотрим программу `last`, единственное назначение которой — отобразить аргумент, переданный последним. Напомним, что в программе `mycp`, рассматривавшейся в главе 9, данная задача выполняется смещением всех аргументов до тех пор, пока не останется один аргумент. Но того же самого результата можно добиться и с помощью команды `eval` следующим образом:

```
$ cat last
eval echo \$$#
$ last one two three four
four
$ last *                        Получить последний файл
zoo_report
$
```

Когда оболочка просматривает в первый раз командную строку

```
echo \$$#
```

знак обратной косой черты предписывает ей проигнорировать следующий далее знак **$**. После этого она обнаруживает специальную переменную **$#** и поэтому подставляет ее значение в командной строке. В итоге анализируемая команда примет следующий вид:

```
echo $4
```

Знак обратной косой черты удаляется оболочкой после первого просмотра командной строки. А при втором ее просмотре оболочка подставляет сначала значение позиционного параметра $4, а затем выполняет команду echo.

Тем же самым способом можно было бы воспользоваться и в том случае, если бы в командной строке имелась переменная arg, содержавшая, например, цифру, и при этом требовалось отобразить позиционный параметр по ссылке на переменную arg. В таком случае можно было бы написать следующую строку:

```
eval echo \$$arg
```

Единственный недостаток такого подхода заключается в том, что подобным способом оказываются доступными только первые девять позиционных параметров. Ведь для доступа к позиционным параметрам свыше 10-го потребуется конструкция ${***n***}. Поэтому сделаем еще одну попытку:

```
eval echo \${$arg}
```

Команда eval, по существу, применяется для создания "указателей" на переменные, как демонстрируется в следующем примере:

```
$ x=100
$ ptrx=x
$ eval echo \$$ptrx          Разыменовать указатель ptrx
100
$ eval $ptrx=50              Сохранить значение 50 в переменной,
                             на которую указывает переменная ptrx
$ echo $x                    Выяснить, к чему это привело
50
$
```

Команда wait

Если перенести команду на выполнение в фоновый режим, она будет выполняться в подоболочке независимо от текущей оболочки (в этом случае говорят, что задание выполняется *асинхронно*). Но иногда требуется подождать завершения фонового процесса, называемого также *порожденным*, поскольку он порожден *родительским* процессом в текущей оболочке, прежде чем продолжить выполнение. Например, крупное задание по сортировке может быть отправлено на выполнение в фоновый режим, а для доступа к отсортированным данным придется ждать завершения этого задания. Именно для этого и предназначена команда wait, общая форма которой выглядит следующим образом:

```
wait идентификатор_процесса
```

где *идентификатор_процесса* обозначает тот процесс, завершения которого требуется подождать. Если идентификатор процесса не указан, оболочка ожидает завершения всех порожденных процессов. В этом случае выполнение текущей

оболочки будет приостановлено до тех пор, пока не завершится ожидаемый процесс или ряд процессов.

Команду `wait` можно попробовать запустить на выполнение с терминала следующим образом:

```
$ sort big-data > sorted_data &          Отправить задание на
                                         сортировку в фоновый режим
[1] 3423                                 Номер задания и идентификатор
                                         процесса от оболочки
$ date                                   Выполнить другое задание
Wed Oct 2 15:05:42 EDT 2002
$ wait 3423                              А теперь ожидать завершения
                                         задания на сортировку
$
```

Переменная $!

Если в фоновом режиме выполняется лишь один процесс, то для ожидания его завершения достаточно указать команду `wait` без аргументов. Но если в фоновом режиме выполняется несколько процессов и требуется подождать завершения только того процесса, который был запущен самым последним, то для доступа к его идентификатору можно обратиться к специальной переменной `$!`. Таким образом, следующая команда:

```
wait $!
```

означает ожидание завершения того процесса, который был последним отправлен на выполнение в фоновом режиме. Идентификаторы процессов можно также хранить в ряде промежуточных переменных для последующего доступа, как демонстрируется в следующем примере:

```
prog1 &
pid1=$!
...
prog2 &
pid2=$!
...
wait $pid1      # ожидать завершения программы prog1
...
wait $pid2      # ожидать завершения программы prog2
```

Чтобы проверить, выполняется ли запущенный процесс по-прежнему, достаточно вызвать команду `ps` с параметром **`-p`** и идентификатором данного процесса.

Команда `trap`

Когда вы нажимаете клавишу <DELETE> или <BREAK> на терминале во время выполнения программы, ее выполнение, как правило, приостанавливается и вы

получаете приглашение ввести следующую команду. Но это не всегда желательно в программах оболочки. Ведь в конечном итоге у вас может оказаться целый ряд временных файлов, которые не будут удалены, как это обычно происходит при нормальном завершении программы.

При нажатии клавиши <DELETE> выполняющейся программе посылается так называемый *сигнал*. В программах можно указать действие, которое следует выполнить при получении данного сигнала, вместо того чтобы опираться на такие выполняемые по умолчанию действия, как немедленный выход из текущего процесса.

Обработка сигнала в программе оболочки выполняется по команде `trap`, общая форма которой выглядит следующим образом:

```
trap команды сигналы
```

где *команды* обозначает одну или больше команд, которые будут выполняться всякий раз, когда получаются указанные *сигналы*.

Наиболее употребительные мнемонические имена и номера, присваиваемые разным типам сигналов, перечислены в табл. 12.1. Более полный их перечень можно найти в описании команды `trap`, приведенном в приложении А.

Таблица 12.1. Наиболее употребительные мнемонические имена и номера сигналов

Сигнал	Мнемоническое имя	Когда генерируется
0	**EXIT**	При выходе из оболочки
1	**HUP**	При зависании программы
2	**INT**	При прерывании (например, в результате нажатия клавиши <DELETE> или комбинации клавиш <Ctrl+c>)
15	**TERM**	При получении сигнала о завершении программы (по умолчанию этот сигнал посылается командой **kill**)

В приведенном ниже примере выполнения команды `trap` демонстрируется, каким образом сначала удаляются некоторые файлы, а затем происходит выход из программы, если кто-нибудь попытается прервать программу с терминала.

```
trap "rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit" INT
```

Как только эта команда `trap` будет выполнена, оба файла, `work1$$` и `dataout$$`, будут автоматически удалены, если программа получит сигнал `SIGINT` (под номером **2**). Если же пользователь прервет выполнение программы по завершении данной команды `trap`, то оба указанных в ней временных файла будут удалены из файловой системы. Команда `exit`, следующая после команды `rm`, необходима потому, что без нее программа продолжит свое выполнение с того места, где был получен сигнал.

Сигнал номер **1** (SIGHUP или просто HUP) генерируется при зависании программы. Первоначально это было связано с установлением коммутируемых соединений, но теперь это, как правило, означает неожиданный разрыв соединения, аналогичного подключению к Интернету. Так, если вернуться к приведенному выше примеру команды trap, то, добавив сигнал SIGINT (или просто INT) в перечень сигналов, оба указанных в ней файла можно удалить и в случае зависания, как показано ниже. Если теперь линия связи зависнет или же пользователь прервет обработку, нажав клавишу <DELETE> или комбинацию клавиш <Ctrl+c>, оба указанных файла будут удалены.

```
trap "rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit'' INT HUP
```

Последовательность команд, указанная в команде trap и называемая *обработчиком прерываний*, должна быть заключена в кавычки. Следует также иметь в виду, что оболочка просматривает командную строку, когда выполняется команда trap, и делает это снова, когда получается один из сигналов, перечисленных в данной команде.

В приведенном выше примере значения переменных WORKDIR и $$ подставляются в момент выполнения команды trap. Если же требуется, чтобы подстановка произошла в момент получения сигнала, последовательность команд следует заключить в одиночные кавычки:

```
trap 'rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit' INT HUP
```

Команда trap может быть использована и для того, чтобы сделать разрабатываемые программы более удобными для пользователей. Так, при рассмотрении программы rolo в следующей главе в нее будет внесено изменение, предусматривающее перехват сигнала при нажатии комбинации клавиш <Ctrl+c>, а также возврат к основному меню пользователя, а не к полному выходу из программы.

Команда trap без аргументов

Выполнение команды trap без аргументов приводит к отображению любых определенных или видоизмененных обработчиков прерываний, как демонстрируется в следующем примере:

```
$ trap 'echo logged off at $(date) >>$HOME/logoffs' EXIT
$ trap                          Перечислить измененные прерывания
trap - 'echo logged off at $(date) >>$HOME/logoffs' EXIT
$ Ctrl+d                        Выйти из системы
login: steve                    Снова войти в систему
Password:
$ cat $HOME/logoffs             Выяснить, что произошло
logged off at Wed Oct 2 15:11:58 EDT 2002
$
```

Прерывание должно устанавливаться в данном примере всякий раз, когда происходит выход из оболочки (после получения сигнала **0** или EXIT). А поскольку это прерывание было установлено в исходной оболочке, то обработчик прерываний используется при выходе из системы для записи времени наступления данного события в файле $HOME/logoffs. В данном случае последовательность команд заключена в одиночные кавычки для того, чтобы предотвратить выполнение команды date в оболочке при определении прерывания.

Затем в данном примере выполняется команда trap без аргументов, перечисляющая новое действие, которое должно быть выполнено по сигналу **0** или EXIT. Когда же пользователь steve выходит из системы и снова входит в нее, в файле $HOME/logoffs проверяется, была ли выполнена команда echo и сработало ли прерывание.

Игнорирование сигналов

Если в команде trap указан пустой список команд, заданный сигнал будет проигнорирован при его получении. Например, в команде

```
trap "" SIGINT
```

указывается, что сигнал прерывания должен быть проигнорирован. При выполнении отдельной операции может возникнуть потребность проигнорировать определенные сигналы, чтобы не прерывать ее.

Следует, однако, иметь в виду, что команда trap позволяет указывать сигналы по их номеру, сокращенному (INT) или полному наименованию (SIGINT). В связи с этим рекомендуется пользоваться мнемоническими именами сигналов, чтобы повысить удобочитаемость исходного текста программ. Хотя это, конечно, дело личных предпочтений.

В приведенном выше примере в качестве первого аргумента следовало указать пустое значение, чтобы проигнорировать сигнал. Но это не равнозначно написанию следующей команды, которая имеет отдельное назначение:

```
trap 2
```

Если сигнал игнорируется, он игнорируется и во всех подоболочках. Но если указать действие для обработки прерывания, то при получении данного сигнала во всех подоболочках будет автоматически выполнено действие, заданное *по умолчанию*, а не новая последовательность команд.

Допустим, что сначала выполняется следующая команда:

```
trap "" 2
```

а затем запускается подоболочка, где, в свою очередь, другие программы оболочки выполняются как подоболочки. Если далее сгенерируется сигнал прерывания, он не окажет никакого влияния на выполняющиеся оболочки или подоболочки, поскольку он будет проигнорирован в них по умолчанию.

А если вместо предыдущей команды выполнить сначала следующую команду:

```
trap   2
```

а затем запустить на выполнение подоболочки, то в текущей оболочке не будет предпринято никаких действий при получении прерывания (т.е. фактически будет выполнена пустая команда), тогда как действие подоболочек будет (по умолчанию) прервано.

Сброс прерываний

Если действие, предпринимаемое по умолчанию при получении сигнала, было изменено, его можно восстановить по команде `trap`, опустив в ней первый аргумент. Таким образом, по следующей команде:

```
trap HUP INT
```

сбрасывается действие, предпринимаемое при получении сигнала `SIGHUP` или `SIGINT`, и восстанавливается режим работы оболочки по умолчанию.

Во многих программах оболочки применяется также конструкция, аналогичная следующей:

```
trap "/bin/rm -f $tempfile; exit" INT QUIT EXIT
```

чтобы команда `rm` не выводила сообщение об ошибке, если временный файл еще не был создан после выхода. Обработчик прерываний удаляет временный файл, если он существует, но ничего не делает, если он отсутствует.

Дополнительные сведения об организации ввода-вывода

Как известно, стандартные конструкции **<**, **>** и **>>** служат для переадресации ввода, вывода и вывода с присоединением соответственно. Известно также, что стандартный вывод ошибок можно переадресовать из любой команды, просто указав **2>** вместо **>** в следующей конструкции:

```
команда 2> файл
```

Иногда в программе требуется явным образом направить сообщения в стандартный вывод ошибок. Внеся незначительное изменение в предыдущую конструкцию, стандартный вывод можно переадресовать в стандартный вывод ошибок следующим образом:

```
команда >&2
```

Конструкция **>&** обозначает переадресацию вывода в файл, связанный с указанным далее *дескриптором файла*. В частности, дескриптор файла **0** обозначает

стандартный ввод, дескриптор файла **1** — стандартный вывод, а дескриптор файла **2** — стандартный вывод ошибок. Очень важно запомнить, что между знаками **>** и **&** не должно быть пробела. Например, направить сообщение в стандартный вывод ошибок можно следующим образом:

```
echo "Invalid number of arguments" >&2
```

Иногда требуется переадресовать как стандартный вывод (зачастую обозначаемый сокращенно как `stdout`), так и стандартный вывод ошибок (нередко обозначаемый сокращенно как `stderr`) из программы в один и тот же файл. Если известно имя файла, это нетрудно сделать, используя приведенную ниже конструкцию, где стандартный вывод `stdout` и `stderr` переадресовывается в указанный файл *имя_файла*.

```
команда > имя_файла 2>> имя_файла
```

Чтобы добиться того же самого результата, можно воспользоваться следующей конструкцией:

```
команда > имя_файла 2>&1
```

где стандартный вывод переадресовывается в указанный файл *имя_файла*, а стандартный вывод ошибок — в стандартный вывод, который уже переадресован в указанный файл *имя_файла*. Но поскольку оболочка определяет порядок переадресации в командной строке слева направо, то последовательность команд в последнем примере будет выполнена неверно, если переадресация в стандартный вывод `stderr` указана в командной строке первой, как показано ниже. Ведь в этом случае сначала будет выполнена переадресация результатов выполнения заданной *команды* в стандартный вывод ошибок, а затем переадресация стандартного вывода в указанный файл *имя_файла*.

```
команда 2>&1 > имя_файла
```

Стандартный ввод или вывод можно также динамически переадресовать в своей программе, используя команду `exec` следующим образом:

```
exec < datafile
```

В данном примере стандартный ввод переадресовывается в указанный файл `datafile`. И тогда последующие команды, читающие данные из стандартного ввода, будут читать их из указанного файла `datafile`. То же самое делается в следующей команде:

```
exec > /tmp/output
```

где все последующие команды, направляющие данные в стандартный вывод, направляют их в указанный файл `/tmp/output`, если только не указано явно какое-нибудь другое место их переадресации.

Естественно, что переадресовать можно и стандартный вывод ошибок, как показано ниже. В таком случае все сообщения об ошибках направляются не в стандартный вывод ошибок, а в файл /tmp/errors.

```
exec 2> /tmp/errors
```

Конструкции <&- и >&-

Конструкция **>&-** оказывает действие закрытия стандартного вывода. Если эта конструкция предваряется дескриптором файла, то закрывается соответствующий файл, а не стандартный вывод. Таким образом, команда

```
ls >&-
```

направляет результат своего выполнения в "никуда", поскольку стандартный вывод закрывается оболочкой перед выполнением команды ls. Следует, однако, признать, что пользы от этого практически никакой!

Встраиваемая переадресация ввода

Если последовательность символов **<<** употребляется в следующей форме:

команда <<слово

оболочка использует последующие строки в качестве входных данных указанной *команды* до тех пор, пока входная строка содержит заданное *слово*. Ниже приведен простой пример применения данной конструкции, где каждая введенная строка предоставляется оболочкой команде wc в качестве стандартного потока ввода до тех пор, пока не встретится строка, состоящая только из слова ENDOFDATA.

```
$ wc -l <<ENDOFDATA                  Воспользоваться строками
                                     в качестве стандартного ввода
                                     вплоть до слова ENDOFDATA

> here's a line
> and another
> and yet another
> ENDOFDATA
       3
$
```

Встраиваемая переадресация ввода называется также *встраиваемыми документами* и служит эффективным средством при написании программ оболочки. Это средство позволяет указывать стандартный ввод команды непосредственно в программе, исключая необходимость записывать его в отдельный файл или получать его с помощью команды echo.

В следующем примере демонстрируется применение этого средства в программе оболочки:

```
$ cat mailmsg
mail $* <<END-OF-DATA

Attention:

Our monthly computer users group meeting
will take place on Friday, March 4, 2016 at
8pm in Room 1A-308. Please try to attend.

END-OF-DATA
$
```

Чтобы отправить приведенное выше сообщение, сохраненное в файле `users_list`, всем членам группы можно вызвать следующую команду:

```
mailmsg $(cat users_list)
```

Оболочка подставляет вместо параметра переадресованные входные данные, выполняет команды в обратных кавычках, а также распознает знак обратной косой черты. Специальные символы обычно игнорируются во *встраиваемом документе*. Но если ввести в его строках знаки денежной единицы, обратных кавычек или обратной косой черты, то они могут быть правильно интерпретированы. Чтобы пренебречь этими знаками, их следует предварить знаком обратной косой черты. А если требуется, чтобы оболочка оставила все введенные строки без изменения, то слово, обозначающее конец документа и указываемое после символов **<<**, следует также предварить знаком обратной косой черты.

В следующем примере показан порядок интерпретации специальных символов во встраиваемом документе:

```
$ cat <<FOOBAR
> $HOME
> *****
> \$foobar
> `date`
> FOOBAR                Прервать ввод
/users/steve
*****`
     $foobar
Wed Oct 2 15:23:15 EDT 2002
$
```

В данном примере оболочка предоставляет команде `cat` все строки, введенные вплоть до слова FOOBAR, в качестве входных данных, подставляя значение переменной HOME, но не переменной `foobar`, поскольку знак `$` ссылки на нее предваряется знаком обратной косой черты. Кроме того, выполняется команда `date`, поскольку обратные кавычки правильно интерпретируются оболочкой.

Чтобы уклониться от интерпретации оболочкой специальных символов во введенных строках, достаточно предварить знаком обратной косой черты слово, обозначающее конец встраиваемого документа:

```
$ cat <<\FOOBAR
> \\\\
> `date`
> $HOME
> FOOBAR
\\\\
`date`
$HOME
$
```

Следует весьма тщательно подбирать слово, указываемое после символов **<<**. Как правило, это слово должно быть достаточно необычным, чтобы свести к минимуму вероятность его случайного появления в последующих строках вводимых данных.

Теперь вы знаете, что обозначает последовательность символов **<<**. Но имеется еще одна последовательность символов, распознаваемая в большинстве современных оболочек. Так, если указать после символов **<<** знак тире (**-**), любые начальные символы табуляции будут исключены оболочкой из входных данных. Это удобно, если требуется обозначить переадресовываемый текст с отступом ради повышения его удобочитаемости, в то же время сохранив выводимый текст нормально выровненным по левому краю, как демонстрируется в следующем примере:

```
$ cat <<-END
>            Indented lines
>            because tabs are cool
> END
Indented lines
because tabs are cool
$
```

Архивные файлы оболочек

Встраиваемой переадресацией ввода лучше всего воспользоваться для создания *архивных файлов оболочек*. Подобным способом можно разместить в одном архивном файле несколько взаимосвязанных программ оболочки, а затем отправить этот файл кому-нибудь другому, воспользовавшись стандартной для систем Unix командой `mail`. Как только отправленный архивный файл будет получен, его можно "распаковать", вызвав его как программу оболочки.

В качестве примера ниже приведен архивный файл, содержащий программы `lu`, `add` и `rem`, применяемые в программе `rolo`.

```
$ cat rolosubs
#
# Архивный файл программ, применяемых в программе rolo
#

echo Extracting lu
cat >lu <<\THE-END-OF-DATA
#
# Найти абонента в телефонном справочнике
#

if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments"
      echo "Usage: lu name"
      exit 1
fi

name=$1
grep "$name" $PHONEBOOK

if [ $? -ne 0 ]
then
      echo "I couldn't find $name in the phone book"
fi

THE-END-OF-DATA
echo Extracting add
cat >add <<\THE-END-OF-DATA
#
# Программа для ввода абонента в телефонный справочник
#

if [ "$#" -ne 2 ]
then
      echo "Incorrect number of arguments"
      echo "Usage: add name number"
      exit 1
fi

echo "$1 $2" >> $PHONEBOOK
sort -o $PHONEBOOK $PHONEBOOK
THE-END-OF-DATA

echo Extracting rem
cat >rem <<\THE-END-OF-DATA
#
# Удалить абонента из телефонного справочника
#
```

```
if [ "$#" -ne 1 ]
then
      echo "Incorrect number of arguments"
      echo "Usage: rem name"
      exit 1
fi

name=$1

#
# Найти количество совпавших записей
#

matches=$(grep "$name" $PHONEBOOK | wc -1)

#
# Если обнаружена не одна совпавшая запись,
# выдать сообщение, а иначе удалить найденную запись
#

if [ "$matches" -gt 1 ]
then
      echo "More than one match; please qualify further"
      elif [ "$matches" -eq 1 ]
then
      grep -v "$name" $PHONEBOOK > /tmp/phonebook
      mv /tmp/phonebook $PHONEBOOK
else
      echo "I couldn't find $name in the phone book"
fi
THE-END-OF-DATA
$
```

Для большей полноты этот архивный файл должен также включать в себя программу `rolo`, но в данном примере она в него не включена ради экономии места. В данном случае составлен единый переносимый архивный файл оболочки `rolosubs`, который содержит исходный текст всех трех программ `lu`, `add`, `rem` и может быть отправлен кому-нибудь другому по команде `mail`, как демонстрируется в следующем примере:

```
$ mail tony@aisystems.com < rolosubs        Отправить архивный файл
                                            по электронной почте
$ mail tony@aisystems.com                   Отправить по электронной
                                            почте сообщение
                                            пользователю tony
Tony,
      I mailed you a shell archive containing the programs
      lu, add, and rem. rolo itself will be sent along shortly.
Pat
<Ctrl+d>
$
```

Когда пользователь tony получит отправленный ему архивный файл по электронной почте, он может извлечь из него три программы, сохранив сообщение, а затем выполнив данный файл в оболочке, как показано ниже. Но предварительно он должен удалить любые строки заголовка сообщения электронной почты, которые могут оказаться в начале этого файла.

```
$ sh rolosubs
Extracting lu
Extracting add
Extracting rem
$ ls lu add rem
add
lu
rem
$
```

В качестве обобщения рассматриваемого здесь процесса ниже приведен пример программы shar, способной производить архивный файл оболочки, содержащий все указанные сценарии в формате, готовом для отправки электронной почтой.

```
$ cat shar
#
# Программа для создания архивного файла оболочки
# из ряда исходных файлов
#

echo "#"
echo "# To restore, type sh archive"
echo "#"

for file
do
      echo
      echo "echo Extracting $file"
      echo "cat >$file <<\THE-END-OF-DATA"
      cat $file
      echo "THE-END-OF-DATA"
done
```

Сравните исходный текст программы shar с содержимым рассмотренного выше архивного файла rolosubs. По существу, shar является программой оболочки, а не обычным выходным файлом. Более сложные программы архивирования допускают включать целые каталоги и пользоваться различными методами, исключающими потерю данных при передаче.

Функции

Во всех современных оболочках поддерживаются функции — длинные или короткие последовательности команд, на которые можно ссылаться или повторно использовать как угодно часто в программе оболочки. Для определения функции служит следующая общая форма:

```
имя () { команда;      команда; }
```

где *имя* обозначает имя функции, круглые скобки — определяемую функцию, а команды в фигурных скобках — тело функции. Эти команды будут выполнены всякий раз, когда выполняется сама функция.

Если команды указаны в одной строке, один символ пробела должен отделять открывающую фигурную скобку (**{**) от первой команды, а точка с запятой — закрывающую фигурную скобку (**}**) от последней команды. В следующем примере определяется функция `nu()`, отображающая количество пользователей, зарегистрированных в системе:

```
nu () { who | wc -l; }
```

Данная функция выполняется таким же образом, как и обычная команда. Для этого достаточно ввести ее имя в текущей оболочке:

```
$ nu
      22
$
```

Функции действительно полезны для программирующих на языке оболочки и способны облегчить разработку прикладных программ. Следует, однако, иметь в виду, что аргументы, указанные после имени функции в командной строке, присваиваются в функции позиционным параметрам `$1`, `$2`, и т.д. таким же образом, как и в любой другой команде.

Ниже приведена функция `nrrun()`, выполняющая команды `tbl`, `nroff` и `lp` над указанным файлом.

```
$ nrrun () { tbl $1 | nroff -mm -Tlp | lp; }
$ nrrun memo1              Выполнить функцию над файлом memo1
request id is laser1-33 (standard input)
$
```

Функции существуют только в оболочке, где они определяются таким образом, чтобы их нельзя было передавать подоболочкам. А поскольку функция выполняется в текущей оболочке, то изменения, внесенные в текущем каталоге или в переменных, останутся по завершении функции, как будто бы она была вызвана по команде `.`, рассмотренной ранее в данной книге:

```
$ db () {
>       PATH=$PATH:/uxn2/data
>       PS1=DB:
>       cd /uxn2/data
> }
$ db                              Выполнить эту функцию
DB:
```

При необходимости определение функции можно продлить на сколько угодно строк. Оболочка продолжит выводить вспомогательное приглашение на ввод команд в теле функции до тех пор, пока ее определение не завершится закрывающей фигурной скобкой (**}**).

С одной стороны, определения наиболее употребительных функций можно разместить в своем файле `.profile`, чтобы сделать их доступными при входе в систему. А с другой стороны, определения функций можно сгруппировать сначала в файле (например, под именем `myfuncs`), а затем выполнить этот файл в текущей оболочке, введя следующую команду:

```
. myfuncs
```

которая, как известно, сделает доступными в текущей оболочке любые функции, определенные в файле `myfuncs`.

В приведенном ниже примере определяется и демонстрируется функция `mycd`, где выгодно используется возможность выполнять функции в текущей среде. Она имитирует действие команды `cd` из оболочки Korn, способной подставлять отдельные части пути к текущему каталогу (более подробно эта команда поясняется в главе 14).

```
$ cat myfuncs                 Отобразить содержимое файла myfuncs
#
# новая функция cd:
#       mycd dir - сменяет каталог
#       mycd old new - подставляет новый каталог вместо
#                      старого в путь к текущему каталогу
#

mycd ()
{
      if [ $# -le 1 ] ; then
            # обычный случай -- 0 или 1 аргумент
            cd $1
      elif [ $# -eq 2 ] ; then
            # особый случай - замена аргумента $2 на $1
            cd $(echo $PWD | sed "s|$1|$2|")
      else
            # у команды cd может быть не больше двух аргументов
            echo mycd: bad argument count
            exit 1
      fi
}
```

```
$ . myfuncs                    Прочитать определение функции
$ pwd
/users/steve
$ mycd /users/pat              Сменить каталог
$ pwd                          Проверить, получилось ли?
/users/pat
$ mycd pat tony                Подставить tony вместо pat
$ pwd
/users/tony
$
```

Функции выполняются быстрее, чем эквивалентные программы оболочки. Ведь в этом случае оболочке не приходится искать файл программы на диске, открывать и читать его содержимое в оперативную память. Она может сразу же перейти к выполнению отдельных команд.

Еще одно преимущество функций заключается в их способности группировать связанные вместе программы оболочки в одном файле. Например, программы `add`, `lu` и `rem`, упоминавшиеся в главе 10, могут быть теперь определены как отдельные функции в файле программы `rolo`. Ниже приведен шаблон для написания программ оболочки в форме функций.

```
$ cat rolo
#
# Программа rolo, написанная в форме функции
#

#
# Функция для ввода абонента в телефонный справочник
#

add () {
      # ввести здесь команды из программы add
}

#
# Функция для поиска абонента в телефонном справочнике
#

lu () {
      # ввести здесь команды из программы lu
}

#
# Функция для удаления абонента из телефонного справочника
#

rem () {
      # ввести здесь команды из программы rem
}
```

```
#
# rolo - программа для поиска, ввода и удаления
#        сведений об абонентах из телефонного справочника
#

# ввести здесь команды из программы rolo
$
```

Ни одна из команд в первоначальных версиях программ `add`, `lu`, `rem` или `rolo` не потребовала никаких изменений. Первые три программы были преобразованы в функции, для чего их команды были размещены между заголовком соответствующей функции и закрывающей фигурной скобкой в исходном тексте программы `rolo`. Но теперь эти программы стали недоступны как самостоятельные команды, поскольку они определены как функции.

Удаление определения функции

Чтобы удалить определение функции из оболочки, достаточно воспользоваться командой `unset` с параметром **`-f`**, как показано ниже. Это очень похоже на удаление определения переменной из оболочки с помощью той же самой команды.

```
$ unset -f nu
$ nu
sh: nu: not found
$
```

Команда `return`

Если употребить команду `exit` в теле функции, то в конечном итоге она прервет выполнение не только функции, но и той программы оболочки, из которой она вызывается. По этой команде происходит выход и возврат непосредственно к командной строке. Но если требуется выйти только из функции, то лучше воспользоваться командой `return`, общая форма которой приведена ниже.

```
return n
```

Значение параметра *n* служит в качестве состояния, возвращаемого из функции. Если же этот параметр опущен, то возвращается состояние завершения последней команды. Именно оно возвращается, если в теле функции отсутствует команда `return`. А во всем остальном возвращаемое состояние равнозначно коду завершения. Его значение доступно через переменную оболочки `$?` и может быть проверено в операторах `if`, `while` и `until`.

Команда `type`

Вводя имя исполняемой команды, полезно знать, является ли эта команда отдельной функцией, встроенной в оболочку функцией, стандартной командой

Unix или даже псевдонимом оболочки. Именно здесь и приходит на помощь команда `type`, которая принимает в качестве своего аргумента имена одной или нескольких команд и сообщает, что именно ей известно о них. Ниже приведены некоторые примеры применения команды `type`.

```
$ nu () { who | wc -l; }
$ type pwd
pwd is a shell builtin
$ type ls
ls is aliased to `/bin/ls -F`
$ type cat
cat is /bin/cat
$ type nu
nu is a function
$
```

13

Возращение к программе rolo

В этой главе представлена окончательная версия программы rolo, существенно усовершенствованная дополнительными параметрами, а также допускающая применение более общих типов записей, а не только имен абонентов и номеров телефонов. В главе обсуждаются отдельные составляющие программы rolo, начиная с главной составляющей самой программы rolo, а в конце демонстрируется образец результата, выводимого из данной программы.

Вопросы форматирования данных

Несмотря на все удобства, которые предоставляет программа rolo для ведения телефонного справочника, рассматривавшаяся ранее в данной книге, она была бы, несомненно, более удобной, если бы позволяла хранить в телефонном справочнике не только имена абонентов и номера их телефонов. В частности, было бы неплохо, если бы данная программа позволяла создавать в телефонном справочнике почтовые и электронные адреса абонентов. Новая версия программы rolo, рассматриваемая в этой главе, допускает сохранять в телефонном справочнике записи, состоящие из нескольких строк. Ниже приведен пример такой записи.

```
Steve's Ice Cream
444 6th Avenue
New York City 10003
212-555-3021
```

Чтобы сделать данную программу более удобной для применения, отдельная запись должна содержать сколько угодно строк. Другая запись в телефонном справочнике может выглядеть следующим образом:

```
YMCA
(201) 555-2344
```

Чтобы логически разделить записи в файле телефонного справочника, каждую запись следует "упаковать" в отдельной строке, заменив в записи завершающие символы новой строки другим символом. В качестве такого символа совершенно произвольно выбран знак вставки (^), поскольку он редко встречается в тех данных, которые вводятся пользователями, в адресах и т.д. Из этого следует, что

данный символ не может быть использован как составная часть самой записи. Таким образом, первая запись, которую можно было бы сохранить подобным способом в файле телефонного справочника, выглядела бы следующим образом:

```
Steve's Ice Cream^444 6th Avenue^New York City 10003^212-555-3021^
```

а вторая запись таким образом:

```
YMCA^(201) 555-2344^
```

Теперь, когда записи хранятся в таком формате, оперировать ими намного проще. Это лишний раз свидетельствует о том, как, тщательно продумав решение поставленной задачи, можно получить немалые выгоды в ходе последующей разработки программы.

Программа `rolo`

Ниже приведен исходный текст программы `rolo`.

```
#
# rolo - программа для поиска, ввода и удаления
# сведений об абонентах из телефонного справочника
#

#
# Установить в переменной PHONEBOOK путь к файлу
# телефонного справочника и экспортировать ее, чтобы
# сделать ее доступной для других программ. Если она
# установлена изначально, оставить ее без изменения
#

: ${PHONEBOOK:=$HOME/phonebook}
export PHONEBOOK
if [ ! -e "$PHONEBOOK" ] ; then
      echo "$PHONEBOOK does not exist!"
      echo "Should I create it for you (y/n)? \c"
      read answer

      if [ "$answer" != y ] ; then
            exit 1
      fi

      > $PHONEBOOK || exit 1 # выйти из программы, если
                             # файл не удалось создать
fi

#
# Если предоставлены аргументы, то произвести поиск
#
```

```
if [ "$#" -ne 0 ] ; then
      lu "$@"
      exit
fi

#
# Установить сигнал прерывания (при нажатии клавиши <DELETE>),
# чтобы продолжить выполнение цикла

trap "continue" SIGINT

#
# Выполнять цикл до тех пор, пока пользователь не
# выберет вариант 'exit', т.е. выход из программы
#

while true
do
      #
      # Отобразить меню
      #

      echo '
      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book
            4. Change an entry in the phone book
            5. List all names and numbers in the phone book
            6. Exit this program

      Please select one of the above (1-6): \c'

      #
      # Прочитать и обработать результат выбора
      #

      read choice
      echo
      case "$choice"
      in
            1) echo "Enter name to look up: \c"
               read name

               if [ -z "$name" ] ; then
                     echo "Lookup ignored"
               else
                     lu "$name"
               fi;;
            2) add;;
            3) echo "Enter name to remove: \c"
```

```
                read name
                if [ -z "$name" ] ; then
                      echo "Removal ignored"
                else
                      rem "$name"
                fi;;

            4) echo "Enter name to change: \c"
                read name
                if [ -z "$name" ] ; then
                      echo "Change ignored"
                else
                      change "$name"
                fi;;
            5) listall;;
            6) exit 0;;
            *) echo "Bad choice\a";;
        esac
done
```

Одно из улучшений введено в самом начале новой версии рассматриваемой здесь программы. Вместо того чтобы требовать наличия файла телефонного справочника в начальном каталоге пользователя, в данной программе проверяется, установлена ли переменная PHONEBOOK. Если она установлена, то предполагается, что она содержит имя файла телефонного справочника. В противном случае в ней по умолчанию устанавливается значение $HOME/phonebook.

Затем в данной программе проверяется, существует ли конкретный файл телефонного справочника, и если он отсутствует, то у пользователя спрашивается, желает ли он создать такой файл. Благодаря этому улучшается первый опыт применения данной программы.

В рассматриваемой здесь версии программы rolo меню было также дополнено новыми вариантами выбора. Отдельные записи могут оказаться довольно длинными, и поэтому у пользователя теперь имеется возможность отредактировать конкретную запись с целью обновить ее. А до этого единственный способ изменить запись состоял в том, чтобы сначала удалить ее из телефонного справочника, а затем ввести в него совершенно новую запись.

Еще одна возможность состоит в том, чтобы отобразить весь телефонный справочник. В этом случае отображаются только первая и последняя строки (или поля) из каждой записи. При этом предполагается, что пользователь придерживается вполне определенного правила вводить имя абонента первым, а номер телефона — последним в строке. Кроме того, весь блок кода для выбора вариантов из меню теперь размещается в цикле while, чтобы программа rolo отображала меню до тех пор, пока пользователь не выберет вариант выхода из нее.

Обратите также внимание на то, что команда trap, пояснявшаяся в предыдущей главе, выполняется перед началом цикла. В ней указывается, что команда

continue будет выполнена, если действия пользователя приведут к появлению сигнала прерывания (SIGINT). Так, если пользователь нажмет комбинацию клавиш <Ctrl+c> в ходе выполнения операции (например, отображения всего телефонного справочника), данная программа прервет текущую операцию и снова выведет главное меню.

Теперь записи могут состоять из сколь угодно большого числа строк, и поэтому были внесены изменения в действие, выполняемое при выборе из меню варианта для ввода записи в телефонный справочник. Вместо того чтобы запрашивать у пользователя имя абонента и номер его телефона, программа rolo теперь вызывает программу add, чтобы получить новую запись, предоставляя ей самой позаботиться о выводе соответствующего приглашения и определении момента завершения ввода данных.

В варианты выбора поиска, изменения и удаления записей из телефонного справочника была введена проверка, чтобы пользователь не ввел пустое значение, нажав клавишу <Enter>, когда ему будет предложено ввести имя абонента. Благодаря этому исключается ошибка в регулярном выражении, которую выдаст команда grep, если ее первый аргумент окажется пустым.

А теперь рассмотрим отдельные составляющие программы rolo, уделив особое внимание соблюдению выбранного формата данных во всей структуре этой программы. В каждую первоначальную программу были внесены соответствующие изменения с учетом нового формата данных, а также с целью сделать ее более удобной для применения.

Программа add

Ниже приведен исходный текст программы add.

```
#
# Программа для ввода абонента в телефонный справочник
#

echo "Type in your new entry"
echo "When you're done, type just a single Enter on the line."

first=
entry=

while true
do
      echo ">> \c"
      read line

      if [ -n "$line" ] ; then
            entry="$entry$line^"
```

```
            if [ -z "$first" ] ; then
                  first=$line
            fi
      else
            break
      fi
done

echo "$entry" >> $PHONEBOOK
sort -o $PHONEBOOK $PHONEBOOK
echo
echo "$first has been added to the phone book"
```

Эта программа вводит запись в телефонный справочник. Она постоянно выводит для пользователя приглашение на ввод строк до тех пор, пока он не завершится нажатием клавиши <Enter>, т.е. совершенно пустой строкой. Каждая вводимая строка присоединяется к значению переменной `entry`, где специальный символ **^** служит для логического разделения полей. При выходе из цикла `while` в телефонный справочник вводится новая запись и сортируется файл этого справочника.

Программа `lu`

Ниже приведен исходный текст программы `lu`.

```
#
# Найти абонента в телефонном справочнике
#

name="$1"
grep -i "$name" $PHONEBOOK > /tmp/matches$$

if [ ! -s /tmp/matches$$ ] ; then
      echo "I can't find $name in the phone book"
else
      #
      # Отобразить каждую совпавшую запись
      #

      while read line
      do
            display "$line"
      done < /tmp/matches$$
fi

rm /tmp/matches$$
```

Данная программа предназначена для поиска записей в телефонном справочнике. Но теперь совпавшие записи записываются в указанный файл `/tmp/`

matches$$, и благодаря этому улучшается опыт применения данной программы в том случае, если совпавших записей не обнаружено.

Если размер данного выходного файла окажется нулевым, что можно проверить с помощью команды test -s, это означает, что совпадение не обнаружено. В противном случае данная программа начнет выполнять цикл для чтения каждой совпавшей записи из файла и ее отображения для пользователя. Новая программа, которая называется display и рассматривается в следующем разделе, служит для извлечения полей, разделяемых знаками ^, преобразуя их в многострочные записи. Эта новая программа применяется также в программах rem и change для отображения записей в телефонном справочнике.

Программа display

Ниже приведен исходный текст программы display.

```
#
# Отображение записи из телефонного справочника
#

echo
echo "---------------------------------------"

entry=$1
IFS="^"
set $entry

for line in "$1" "$2" "$3" "$4" "$5" "$6"
do
      printf "| %-34.34s |\n" $line
done
echo "|        o                   o        |"
echo "---------------------------------------"
echo
```

В программе display отображается разделяемая знаком ^ запись, передаваемая данной программе в качестве аргумента. Чтобы сделать вывод из нее эстетически более привлекательным, данная программа фактически "воспроизводит" карточку из вращающегося каталога типа rolodex. Типичный результат, выводимый из программы display, выглядит следующим образом:

```
---------------------------------------
| Steve's Ice Cream                   |
| 444 6th Avenue                      |
| New York City 10003                 |
| 212-555-3021                        |
|                                     |
|                                     |
|        o                   o        |
---------------------------------------
```

Анализируя исходный текст программы `display`, следует обратить внимание на то, что после пропуска строки и отображения верхнего края карточки данная программа сначала изменяет на `^` значение переменной `IFS`, а затем выполняет команду `set` с целью присвоить каждую "строку" другому позиционному параметру. Так, если значение переменной `entry` равно:

```
Steve's Ice Cream^444 6th Avenue^New York City 10003^212-555-3021^
```

то в результате выполнения команды `set` переменной `$3` присваивается следующее значение:

```
Steve's Ice Cream to $1, 444 6th Avenue to $2, New York City 10003
```

а переменной `$4` — значение `212-555-3021`.

После разделения полей с помощью команды `set` рассматриваемая здесь программа начинает цикл `for`, где выводятся лишь шесть строк данных независимо от их количества в записи. Этим гарантируется однородность внешнего вида карточек, а если потребуется воспроизводить более крупные карточки, то в саму программу нетрудно внести соответствующие изменения.

Если команда `set` была выполнена для значения `Steve's Ice Cream`, как показано выше, переменные `$5` и `$6` примут пустое значение, а следовательно, нижняя часть карточки будет заполнена пустыми строками. Чтобы обеспечить надлежащее выравнивание выводимого результата по левому и правому краю, в данном случае вызывается команда `printf`, отображающая ровно 38 символов по ширине, т.е. начальный знак `|`, пробел, первые 34 символа из переменной `$line`, пробел и завершающий знак `|`.

Программа `rem`

Ниже приведен исходный текст программы `rem`.

```
#
# Удалить абонента из телефонного справочника
#

name=$1

#
# Получить совпавшие записи и сохранить их во временном файле
#

grep -i "$name" $PHONEBOOK > /tmp/matches$$
if [ ! -s /tmp/matches$$ ] ; then
      echo "I can't find $name in the phone book"
      exit 1
fi
```

```
#
# Отобразить совпавшие записи по очереди и
# подтвердить их удаление
#

while read line
do
      display "$line"
      echo "Remove this entry (y/n)? \c"
      read answer < /dev/tty

if [ "$answer" = y ] ; then
      break
fi
done < /tmp/matches$$

rm /tmp/matches$$

if [ "$answer" = y ] ; then
      if grep -i -v "^$line$" $PHONEBOOK > /tmp/phonebook$$
      then
            mv /tmp/phonebook$$ $PHONEBOOK
            echo "Selected entry has been removed"
      elif [ ! -s $PHONEBOOK ] ; then
            echo "Note: You now have an empty phonebook."
      else
            echo "Entry not removed"
      fi
fi
```

Программа `rem` сначала накапливает все совпавшие записи во временном файле, а затем проверяет полученный результат. Так, если размер этого файла окажется нулевым, это означает, что совпадение не было обнаружено, и поэтому выводится соответствующее сообщение об ошибке. В противном случае данная программа отображает каждую совпавшую запись и запрашивает у пользователя разрешение на ее удаление.

С точки зрения пользователя такая норма практики программирования дает пользователю возможность лишний раз убедиться, что удалены будут именно те записи, от которых он собирался избавиться, даже если с заданным шаблоном совпала лишь одна запись.

Как только пользователь даст положительный ответ (`y`) на запрос, цикл будет прерван по команде `break`. По завершении цикла данная программа проверяет значение переменной `answer` с целью определить порядок выхода из цикла. Если значение этой переменной не равно `y`, это означает, что пользователь вообще не желает удалять запись независимо от конкретной причины. В противном случае программа продолжит запрашивать удаление записей, выявляя по команде `grep` все строки, не совпадающие с заданным шаблоном. Следует, однако, иметь в виду,

что команда grep обнаруживает совпадение только с целыми строками, привязывая регулярное выражение в заданном шаблоне к началу и к концу строки.

Следует также иметь в виду, что в крайнем случае, когда пользователь удалит последнюю запись в своем телефонном справочнике, рассматриваемая здесь программа сначала проверяет, существует ли файл телефонного справочника и не равен ли его размер нулю, а затем выводит соответствующее информационное сообщение. И хотя этот случай нельзя считать ошибочным, такая проверка гарантирует от появления неприятного сообщения об ошибке, которое в противном случае будет выдано в результате вызова команды grep с параметром **-v**.

Программа change

Ниже приведен исходный текст программы change.

```
#
# Изменить запись в телефонном справочнике
#

name=$1

#
# Получить совпавшие записи и сохранить их во временном файле
#

grep -i "$name" $PHONEBOOK > /tmp/matches$$
if [ ! -s /tmp/matches$$ ] ; then
      echo "I can't find $name in the phone book"
      exit 1
fi

#
# Отобразить совпавшие записи по очереди и
# подтвердить их изменение
#

while read line
do
      display "$line"
      echo "Change this entry (y/n)? \c"
      read answer < /dev/tty

      if [ "$answer" = y ] ; then
            break
      fi
done < /tmp/matches$$

rm /tmp/matches$$
```

```
if [ "$answer" != y ] ; then
      exit
fi

#
# Запустить редактор с подтвержденной записью
#

echo "$line\c" | tr '^' '\012' > /tmp/ed$$

echo "Enter changes with ${EDITOR:=/bin/vi}"
trap "" 2           # не прерывать редактирование
                    # при нажатии клавиши <DELETE>
$EDITOR /tmp/ed$$

#
# Удалить теперь прежнюю запись и ввести новую
#

grep -i -v "^$line$" $PHONEBOOK > /tmp/phonebook$$
{ tr '\012' '^' < /tmp/ed$$; echo; } >> /tmp/phonebook$$
# последняя команда echo была выполнена с целью восстановить
# завершающий знак новой строки, преобразованный по команде tr

sort /tmp/phonebook$$ -o $PHONEBOOK
rm /tmp/ed$$ /tmp/phonebook$$
```

Программа change дает пользователю возможность отредактировать запись, выбранную в телефонном справочнике. Первая часть данной программы завершается практически таким же образом, как и программа rem, обнаруживая сначала совпавшие записи, а затем предлагая пользователю выбрать среди них ту запись, которую требуется изменить.

Выбранная запись записывается далее во временный файл /tmp/ed$$, причем знаки ^ преобразуются в знаки новой строки. Тем самым запись "развертывается" в отдельные строки в соответствии с порядком отображения записей в программе rolo, а также в целью упростить редактирование. После этого рассматриваемая здесь программа отображает следующее сообщение:

```
echo "Enter changes with ${EDITOR:=/bin/vi}"
```

Это сообщение служит двум целям, извещая пользователя, с помощью какого именно редактора предстоит внести изменения в запись, а также устанавливая значение /bin/ed vi в переменной EDITOR, если она еще не установлена. Благодаря этому пользователь может выбрать предпочтительный редактор, просто присвоив его имя переменной EDITOR, прежде чем запустить программу rolo на выполнение:

```
$ EDITOR=emacs rolo
```

Сигнал номер **2**, генерируемый при нажатии клавиши <DELETE>, игнорируется, поэтому программа `change` не прервет свое выполнение, если пользователь нажмет клавишу <DELETE>, правя запись в редакторе. Далее запускается редактор, чтобы пользователь смог внести любые требующиеся изменения. Как только это будет сделано, программа `change` перейдет к удалению прежней записи из файла телефонного справочника по команде `grep`, а затем измененная запись преобразуется в исходный формат, где поля разделяются знаками **^**, и далее присоединяется в конце данного файла. И здесь необходимо ввести лишний знак новой строки, чтобы он сохранился в файле после новой записи, что нетрудно сделать, выполнив команду `echo` без аргументов. И наконец, файл телефонного справочника сортируется, а временный файл удаляется.

Программа `listall`

Ниже приведен исходный текст программы `listall`.

```
#
# Перечислить все записи в телефонном справочнике
#

IFS='^'     # служит для применения в указанной ниже команде set
echo "-----------------------------------------------------"
while read line
do
      #
      # Получить первое и последнее поля, вероятно,
      # содержащие имена абонентов и номера телефонов
      #

      set $line

      #
      # Отобразить первое и последнее поля (в обратном порядке!)
      #

      eval printf "\"%-40.40s %s\\n\"" "\"$1\"" "\"\${$#}\""
done < $PHONEBOOK
echo "-----------------------------------------------------"
```

Программа `listall` перечисляет все записи в файле телефонного справочника, выводя лишь первую и последнюю строки из каждой записи. В переменной `IFS` устанавливается знак **^**, который употребляется в качестве внутреннего разделителя полей в цикле. Затем из данного файла читается каждая строка, которая далее присваивается переменной `line`. После этого команда `set` присваивает каждое поле соответствующему позиционному параметру, как пояснялось ранее.

Трудность состоит в том, чтобы получить значение первого и последнего позиционных параметров. Сделать первое нетрудно по ссылке на первый позиционный параметр непосредственно в переменной `$1`. А для того чтобы сделать второе, в рассматриваемой здесь программе используются потенциальные возможности команды `eval`, пояснявшейся в главе 12. В частности, команда

```
eval echo \${$#}
```

отображает значение последнего параметра. А в данной конкретной программе вызывается такая команда:

```
eval printf "\"%-40.40s %-s\\n\"" "\"$1\"" "\"\${$#}\""
```

по которой, например, вычисляется следующий результат:

```
printf "%-40.40s %-s\n" "Steve's Ice Cream" "${4}"
```

где упомянутая выше запись просматривается снова и затем подставляется значение переменной `${4}`, прежде чем выполнять команду `printf`.

Образец выводимого результата

А теперь самое время выяснить, каким образом действует рассматриваемая здесь усовершенствованная программа `rolo`. И начнем мы с пустого телефонного справочника, чтобы ввести в него несколько записей. Затем мы перечислим все записи, попытаемся найти в телефонном справочнике знакомого абонента и далее изменим запись о нем. Ради экономии места мы полностью отобразим главное меню программы `rolo` лишь первый раз, как показано ниже.

```
$ PHONEBOOK=/users/steve/misc/book
$ export PHONEBOOK
$ rolo                          Запустить программу на выполнение
/users/steve/misc/book does not exist!
Should I create it for you (y/n)? Y

      Would you like to:

            1. Look someone up
            2. Add someone to the phone book
            3. Remove someone from the phone book
            4. Change an entry in the phone book
            5. List all names and numbers in the phone
            6. Exit this program

      Please select one of the above (1-6): 2

      Type in your new entry
      When you're done, type just a single Enter on the line.
      >> Steve's Ice Cream
      >> 444 6th Avenue
```

```
>> New York City 10003
>> 212-555-3021
>>

Steve's Ice Cream has been added to the phone book

      Would you like to:
            ...

      Please select one of the above (1-6): 2

Type in your new entry
When you're done, type just a single Enter on the line.
>> YMCA
>> 973-555-2344
>>

YMCA has been added to the phone book
      Would you like to:
            ...
      Please select one of the above (1-6): 2

Type in your new entry
When you're done, type just a single Enter on the line.
>> Maureen Connelly
>> Hayden Book Companu
>> 10 Mulholland Drive
>> Hasbrouck Heights, N.J. 07604
>> 201-555-6000
>>

Maureen Connelly has been added to the phone book

      Would you like to:
            ...
      Please select one of the above (1-6): 2

Type in your new entry
When you're done, type just a single Enter on the line.
>> Teri Zak
>> Hayden Book Company
>> (see Maureen Connelly for address)
   (см. адрес в записи абонента Maureen Connelly)
>> 201-555-6060
>>

Teri Zak has been added to the phone book

      Would you like to:
            ...
      Please select one of the above (1-6): 5
```

```
------------------------------------------------------------
Maureen Connelly                          201-555-6000
Steve's Ice Cream                         212-555-3021
Teri Zak                                  201-555-6060
YMCA                                      973-555-2344
------------------------------------------------------------

      Would you like to:
            ...
      Please select one of the above (1-6): 1
Enter name to look up: Maureen
-------------------------------------
| Maureen Connelly                  |
| Hayden Book Companu               |
| 10 Mulholland Drive               |
| Hasbrouck Heights, NJ 07604       |
| 201-555-6000                      |
|      o                 o          |
-------------------------------------

-------------------------------------
| Teri Zak                          |
| Hayden Book Company               |
| (see Maureen Connelly for address)|
| 201-555-6060                      |
|                                   |
|      o                 o          |
-------------------------------------

      Would you like to:
            ...
      Please select one of the above (1-6): 4
      Enter name to change: Maureen

-------------------------------------
| Maureen Connelly                  |
| Hayden Book Companu               |
| 10 Mulholland Drive               |
| Hasbrouck Heights, NJ 07604       |
| 201-555-6000                      |
|      o                 o          |
-------------------------------------

Change this person (y/n)? y
Enter changes with /bin/ed
101
1,$p
Maureen Connelly
Hayden Book Companu
10 Mulholland Drive
Hasbrouck Heights, NJ 07604
201-555-6000
```

```
      2s/anu/any                    Исправить опечатку
      Hayden Book Company
      w
      101
      q

            Would you like to:
                  ...
            Please select one of the above (1-6): 6
$
```

Надеемся, что этот сложный пример программирования на языке оболочки даст ясное представление о том, как разрабатывать крупные программы оболочки и как совместно действуют различные инструментальные средства программирования, предоставляемые системой. Помимо команд, встроенных в оболочку, для выполнения конкретного задания в программе `rolo` применяются команды `tr`, `grep`, `sort`, редактор и стандартные команды файловой системы вроде `mv` и `rm`.

Простота и изящество, с которыми можно связывать вместе все эти инструментальные средства, являются главной причиной вполне заслуженной популярности системы Unix. Подробнее о загрузке программ, входящих в состав программы `rolo`, см. в приложении Б. А в главе 14 представлены интерактивные средства оболочки, а также две оболочки, обладающие рядом замечательных средств, отсутствующих в стандартной оболочке POSIX.

14

Интерактивные и нестандартные средства оболочки

В этой главе речь пойдет о средствах оболочки, удобных для интерактивного взаимодействия с пользователями и не являющихся частью стандарта POSIX на оболочку. Эти средства доступны в двух оболочках, Bash и Korn, которые являются наиболее распространенными среди несовместимых со стандартом POSIX оболочек в системах Unix, Linux и Mac OS.

Оболочка Korn была разработана Дэвидом Корном (David Korn) из компании AT&T Bell Laboratories и предназначена для "восходящей совместимости" с оболочкой System V Bourne и оболочкой по стандарту POSIX. Она широко доступна на всех главных платформах Unix, и если у вас имеется доступ к командной строке, то вам, вероятнее всего, доступна и оболочка Korn в форме `ksh`.

Оболочка Bash (сокращение от Bourne-Again Shell — "Возрожденная" оболочка Борна) была разработана Брайаном Фоксом (Brian Fox) для Фонда свободного программного обеспечения (Free Software Foundation). Она была предназначена для совместимости с оболочкой System V Bourne и стандартной оболочкой POSIX и дополнительно содержит многие расширения из оболочек Korn и C. Оболочка Bash является стандартной в системах Linux и большинстве современных систем Unix и Mac OS, где она была заменена оболочкой Bourne. (На самом деле, если вы пользуетесь оболочкой `sh`, то, вероятнее всего, пользуетесь и оболочкой `bash`, даже не подозревая об этом.)

За исключением нескольких малозначительных отличий, оболочки Bash и the Korn предоставляют все средства, доступные в оболочке по стандарту POSIX, а также немало новых средств. Чтобы дать ясное представление о совместимости этих оболочек со стандартом POSIX, все программы, представленные в данной книге, вполне работоспособны и в оболочке Bash, и в Korn. В этой главе особое внимание уделяется любым нестандартным средствам, а в ее конце и в табл. 14.4 перечислены те средства, которые поддерживаются в разных оболочках.

Выбор подходящей оболочки

В рассмотренных до сих пор примерах программ все команды размещались в файле, который выполнялся как программа оболочки. При этом вообще не

обсуждалось, какая именно оболочка читала строки из файла программы и выполняла ее. По умолчанию программы оболочки выполняются в исходной оболочке после входа в систему, и поэтому особых затруднений в этой связи до сих пор не возникало.

Оказывается, что все основные интерактивные оболочки позволяют указывать, какая именно оболочка, входящая в распространяемую версию системы Unix или Linux, должна использоваться для выполнения файла. Так, если в первой строке файла первыми указаны символы **#!**, то в остальной части этой строки — интерпретатор данного файла. Таким образом, следующая строка:

```
#!/bin/ksh
```

обозначает оболочку Korn, а приведенная ниже строка — оболочку Bash.

```
#!/bin/bash
```

Если используются конструкции или условные обозначения, характерные для одной оболочки, это дает возможность выполнять программы в оболочке, избегая трудностей совместимости. Благодаря тому что в оболочке можно указать какую угодно программу, программа на Perl, начинающаяся со следующей строки:

```
#!/usr/bin/perl
```

вынуждает оболочку вызвать интерпретатор `/usr/bin/perl` для обработки строк в файле данной программы.

Но такой возможностью следует пользоваться аккуратно, поскольку многие программы (например, на Perl) не находятся в стандартном для каждой системы Unix месте. Кроме того, подобная возможность не указана в стандарте POSIX, несмотря на то, что она имеется во всякой современной оболочке и даже реализована на уровне операционной системы во многих версиях Unix.

Чаще всего приведенное ниже условное обозначение можно обнаружить в программах системной оболочки, где оно присутствует для того, чтобы пользоваться оболочкой Bourne независимо от того, с чего именно начинается исходная оболочка при входе пользователя в систему.

```
#!/bin/sh
```

Файл ENV

Когда оболочка запускается, она, прежде всего, обнаруживает среду, указанную в переменной `ENV`. Если значение этой переменной оказывается непустым, то выполняется файл, указанный в переменной `ENV`, подобно файлу `.profile` при входе в систему. Файл `ENV` содержит команды для установки среды оболочки. В этой главе упоминаются различные элементы, которые должны быть включены в данный файл.

Если решить, что файл ENV необходим, то переменную ENV следует установить и экспортировать в файле .profile следующим образом:

```
$ cat .profile
...
export ENV=$HOME/.alias
...
$
```

Обратите внимание на сокращение, использованное в приведенном выше примере. Вместо того чтобы присваивать сначала значение переменной ENV, а затем вызывать команду export, в данном случае это сделано в одной строке ради большей эффективности.

Для пользователей оболочки Bash файл ENV читается только в том случае, если оболочка Bash вызывается под именем sh с параметром командной строки **--posix** или после выполнения команды set с параметром **-o posix** — все это обеспечивает совместимость со стандартом POSIX. Когда запускается неинтерактивная оболочка Bash (например, при выполнении программы оболочки), по умолчанию она читает команды из файла, указанного в переменной окружения BASH_ENV. Но этого не происходит, когда запускается интерактивная оболочка Bash (например, в том случае, если ввести **bash** в командной строке).

Если вы работаете в прежней версии системы, то должны также экспортировать переменную SHELL из файла .profile следующим образом:

```
$ grep SHELL .profile
SHELL=/bin/ksh ; export SHELL
$
```

Эта переменная применяется в определенных приложениях (например, в редакторе vi) с целью определить, какую именно оболочку следует запустить при переключении на другой командный процессор. В подобных случаях требуется убедиться, чтобы всякий раз, когда запускается новая оболочка, в конечном итоге получалась требующаяся, а не прежняя оболочка Bourne. Хотя переменная SHELL, вероятнее всего, уже установлена в исходной оболочке при входе в систему. Это можно проверить следующим образом:

```
$ echo $SHELL
/bin/bash
$
```

В предыдущем примере продемонстрирован еще один способ установки и экспорта переменной с помощью двух отдельных команд, разделяемых в одной строке точкой с запятой. В чем же причина несовместимости? Она заключается в такой гибкости системы Unix, которая позволяет разрабатываемым и применяемым программам оболочки решать одну и ту же задачу разными способами, а для этого приходится мириться с некоторыми отличиями в обозначениях!

Редактирование в режиме командной строки

Режим редактирования строк — это средство оболочки, позволяющее редактировать командную строку таким образом, чтобы имитировать средства, доступные в двух распространенных экранных редакторах. В оболочке по стандарту POSIX предоставляется возможность сымитировать возможности редактора `vi`, а в оболочках Bash и Korn — еще и режим редактирования строк в редакторе `emacs`. Полный перечень команд редактора `vi` приведен в табл. А.4 приложения А.

Если вы привыкли пользоваться любым из этих экранных текстовых редакторов, то обнаружите, что встроенные в оболочку редакторы являются точно воспроизведенными их копиями с точки зрения выполняемых функций. И это одна из самых полезных возможностей оболочки.

Чтобы включить режим редактирования строк, достаточно ввести команду `set` с параметром **`-o`** *режим*, где *режим* обозначает редактор `vi` и `emacs`. Ниже показано, как это делается на практике. Строку с командой `set` можно ввести в файл `.profile` или `ENV`, чтобы запустить оболочку с автоматически включенным избранным режимом редактирования строк.

```
$ set -o vi              Включить режим редактирования vi
```

Предыстория команд

Независимо от того, какая именно оболочка применяется, в ней ведется предыстория всех вводившихся ранее команд. Всякий раз, когда вы нажимаете клавишу <Enter>, чтобы выполнить введенную команду, эта команда добавляется в конце списка предыстории. В зависимости от сделанных вами установок ваша предыстория команд может быть даже сохранена в файле и восстановлена в промежутке между сеансами регистрации в системе, чтобы у вас была возможность быстро получить доступ к командам из предыдущих сеансов работы в системе.

По умолчанию список предыстории команд хранится в файле `.sh_history` (или `.bash_history` для оболочки Bash), который находится в начальном каталоге текущего пользователя. Имя этого файла можно изменить как угодно, установив его в переменной `HISTFILE`. Эту переменную можно устанавливать и экспортировать в своем файле `.profile`.

В предыстории оболочки может быть сохранено ограниченное количество команд. Минимальное количество сохраняемых команд составляет **128**, хотя в большинстве оболочек допускается сохранять **500** и больше команд в списке предыстории. Всякий раз, когда вы входите в систему, оболочка автоматически усекает ваш файл предыстории до этой величины.

Размер файла предыстории можно регулировать посредством переменной `HISTSIZE`. Если же устанавливаемый по умолчанию размер файла предыстории

не отвечает вашим потребностям, установите в переменной HISTSIZE большее значение, например, **500** или **1000**. Значение, присваиваемое переменной HISTSIZE, может быть установлено и экспортировано в файле .profile следующим образом:

```
$ grep HISTSIZE .profile
HISTSIZE=500
export HISTSIZE
$
```

Только не переусердствуйте, устанавливая размер файла предыстории. Чем он больше, тем больше пространства на диске потребуется для хранения файла предыстории и тем дольше оболочке придется искать предыдущие команды в предыстории.

Режим правки строк в редакторе vi

Включив режим правки строк в редакторе vi, вы можете ввести все последующие команды в привычном для пользователей редактора vi режиме *ввода*. Вероятно, вы даже не заметите никаких отличий, поскольку сможете вводить и выполнять команды точно так же, как и по стандартному приглашению в оболочке:

```
$ set -o vi
$ echo hello
hello
$ pwd
/users/pat
$
```

Чтобы воспользоваться построчным редактором, вам придется перейти в *командный* режим, нажав клавишу <ESCAPE> (или <ESC>), которая обычно располагается слева вверху на клавиатуре. Как только вы перейдете в командный режим, курсор сместится на одну позицию влево к символу, введенному последним.

Текущим оказывается тот символ, на котором находится курсор. Подробнее о текущем символе речь пойдет несколько позже. Команды редактора vi можно вводить только в командном режиме, где они интерпретируются немедленно после их ввода, не требуя нажатия клавиши <Enter>.

Характерное затруднение, которое может возникнуть при вводе длинных команд, заключается в том, что ошибка обнаруживается только после ввода всей команды. И зачастую такую ошибку совершают в начале строки!

В командном режиме можно перемещать курсор и исправлять ошибки, не обращаясь непосредственно к командной строке. Переместив курсор на место обнаруженной ошибки, можно заменить одну или несколько букв на все, что угодно. И как только будет нажата клавиша <Enter> (независимо от местоположения курсора в строке), введенная команда будет передана оболочке для последующей интерпретации.

В приведенных ниже примерах знак подчеркивания (_) обозначает местоположение курсора. Сначала отображается прежнее состояние командной строки, затем одно или несколько нажатий клавиш и последующее состояние командной строки в такой форме:

прежде нажатия клавиш после

Рассмотрим сначала порядок перемещения курсора. Многие системы позволяют перемещать курсор влево или право, нажимая клавишу <←> или <→> соответственно.

Но более общий порядок перемещения курсора, принятый под влиянием редактора `vi`, состоит в том, чтобы нажать клавишу <h> для перемещения курсора влево или клавишу <l> для перемещения курсора вправо. Попробуйте переместить курсор подобным способом, нажав сначала клавишу <Esc>, чтобы перейти в командный режим, а затем несколько раз клавишу <h> или <l>. В итоге курсор должен переместиться в строке. Если же вы попытаетесь переместить курсор за левый или правый край строки, оболочка выдаст предупреждающий сигнал.

```
$ mary had a little larb_       Esc    $ mary had a little larb
$ mary had a little larb        h      $ mary had a little larb
$ mary had a little larb        h      $ mary had a little larb
$ mary had a little larb        l      $ mary had a little larb
```

Установив курсор на том символе, который требуется заменить, воспользуйтесь командой `x`, чтобы удалить текущий символ, как показано ниже. Обратите внимание на то, что после удаления символа `r` курсор сместился влево к символу `b`, который стал текущим.

```
$ mary had a little larb        x      $ mary had a little lab
```

Чтобы добавить символы в командной строке, следует воспользоваться командой `i` или `a`. В частности, команда `i` вставляет символы *до* текущего символа, а команда `a` — *после* него. Обе эти команды возвращают вас в режим ввода, поэтому не забудьте нажать клавишу <Esc>, чтобы вернуться обратно в командный режим. Ниже приведен наглядный пример редактирования командной строки.

```
$ mary had a little lab         im     $ mary had a little lamb
$ mary had a little lamb        m      $ mary had a little lammb
$ mary had a little lammb       Esc    $ mary had a little lammb
$ mary had a little lammb       x      $ mary had a little lamb
$ mary had a little lamb        a      $ mary had a little lamb_
$ mary had a little lamb_       da     $ mary had a little lambda_
```

Если вы считаете, что курсор перемещается медленно в результате поочередного нажатия клавиши <h> или <l>, то вы совершенно правы. Чтобы ускорить перемещение курсора в нужную сторону, достаточно предварить команду `h` или `l` числом, обозначающим количество позиции, на которое следует переместить курсор, как демонстрируется в следующем примере:

```
$ mary had a little lambda_        Esc    $ mary had a little lambda
$ mary had a little lambda         10h    $ mary had a little lambda
$ mary had a little lambda         13h    $ mary had a little lambda
$ mary had a little lambda         5x     $ had a little lambda
```

Как видите, команду x можно также предварить числом, чтобы указать количество удаляемых символов. Чтобы переместить курсор в конец строки, достаточно ввести команду $, а в начало строки — команду 0 (т.е. нуль):

```
$ had a little lambda        $     $ had a little lambda
$ had a little lambda        0     $ had a little lambda
```

Для перемещения курсора удобны также команды w и b. В частности, команда w перемещает курсор вперед к началу следующего слова, которое является строкой букв, чисел и знаков подчеркивания, ограниченных пробелами или знаками пунктуации, а команда b — к началу предыдущего слова. Обе эти команды можно предварить числом, чтобы обозначить перемещение курсора на заданное количество слов вперед или назад, как показано в следующем примере:

```
$ had a little lambda          w     $ had a little lambda
$ had a little lambda          2w    $ had a little lambda
$ had a little lambda          3b    $ had a little lambda
```

Клавишу <Enter> можно нажать в любой момент, после чего текущая строка будет выполнена как команда. А после выполнения команды произойдет возврат в режим ввода, как показано ниже.

```
$ had a little lambda              Нажать клавишу <Enter>
ksh: had: not found
$ _
```

Доступ к командам из предыстории

До сих пор было показано, как редактировать текущую командную строку. Но в режиме редактирования строк в редакторе vi можно также воспользоваться командами k и j для извлечения команд из предыстории. В частности, команда k заменяет текущую строку на терминале введенными ранее строками, размещая курсор в начале строки.

Допустим, были введены следующие команды:

```
$ pwd
/users/pat
$ cd /tmp
$ echo this is a test
this is a test
$ _
```

Перейдите в командный режим, нажав клавишу <Esc>, и воспользуйтесь командой k, чтобы получить доступ к предыдущим командам из предыстории команд:

```
$ _                    Esc k       $ echo this is a test
```

Всякий раз, когда выполняется команда k, текущая строка заменяется предыдущей строкой из предыстории команд настолько, насколько удалось углубиться в предысторию в пределах, заданных в переменной HISTSIZE, как пояснялось ранее.

```
$ echo this is a test          k       $ cd /tmp
$ cd /tmp                      k       $ pwd
```

Чтобы выполнить отображаемую команду, достаточно нажать клавишу <Enter>.

```
$ pwd              Нажать клавишу <Enter>
/tmp
$ _
```

Команда j выполняет действие, противоположное команде k, и служит для отображения следующей команды, сохраненной в предыстории самой последней. Иными словами, команда k перемещает вас во времени назад, тогда как команда j — вперед. Попробуйте выполнить их несколько раз, чтобы лучше понять их назначение.

Команда / производит поиск в предыстории той команды, которая содержит указанную символьную строку. Так, если после команды / введена символьная строка, оболочка произведет в обратном порядке поиск в предыстории, чтобы найти самую последнюю из выполнявшихся команду, содержащую данную строку. В итоге найденная команда отображается.

Если указанная символьная строка отсутствует в предыстории, оболочка выдаст сигнал, предупреждающий об ошибке. Когда вводится команда /, она заменяет текущую строку:

```
/tmp
$ _                          Esc /test     /test_
```

Чтобы выполнить команду, полученную в результате поиска, следует нажать клавишу <Enter> еще раз:

```
$ echo this is a test          Нажать клавишу <Enter> снова
this is a test
$ _
```

Если отображаемая команда не является искомой, необходимо продолжить поиск, введя команду / *без* шаблона и нажав клавишу <Enter>. Оболочка в достаточной степени развита логически, чтобы воспользоваться символьной строкой, введенной при выполнении команды поиска в последний раз, продолжив поиск в предыстории команд дальше.

Обнаружив команду в предыстории, будь то команда k, j или /, ее можно отредактировать, используя рассмотренные ранее команды редактора vi. Однако

данная операция совсем не означает изменение команды в самой предыстории. По существу, редактируется копия данной команды, которая затем вводится в предысторию как самая последняя команда нажатием клавиши <Enter>. Основные команды редактирования строк в редакторе `vi` сведены в табл. 14.1.

Таблица 14.1. Основные команды редактирования строк в редакторе `vi`

Команда	Назначение
`h`	Переместить курсор на один символ влево
`l`	Переместить курсор на один символ вправо
`b`	Переместить курсор на одно слово влево
`w`	Переместить курсор на одно слово вправо
`0`	Переместить курсор в начало строки
`$`	Переместить курсор в конец строки
`x`	Удалить символ на позиции курсора
`dw`	Удалить слово на позиции курсора
`rc`	Заменить на *`c`* символ на позиции курсора
`a`	Перейти в режим ввода и ввести текст после текущего символа
`i`	Перейти в режим ввода и ввести текст до текущего символа
`k`	Получить предыдущую команду из предыстории
`j`	Получить следующую команду из предыстории
`/`*`строка`*	Найти в предыстории самую последнюю команду, содержащую указанную *`строку`*; если указанная *`строка`* оказывается пустой, будет использована предыдущая строка

На первый взгляд кажется, что команд для редактирования строк в редакторе `vi` слишком много. Но на самом деле для работы в режиме редактирования командной строки в редакторе `vi` можно вполне обойтись командами `j` и `k`, чтобы перемещаться по списку команд в предыстории; командами `h` и `l`, чтобы перемещать курсор в командной строке; а также командой `i`, чтобы ввести текст и нажать клавишу <Enter> для вызова оболочки.

Режим правки строк в редакторе `emacs`

Если вам больше по душе построчный редактор `emacs`, а не визуальный редактор `vi`, как, впрочем, и для большинства членов сообщества разработчиков открытого программного обеспечения, то в оболочке имеется также режим для редактирования строк в построчном редакторе `emacs`. Перейдя в этот режим, вы и в этом случае не заметите особых отличий, поскольку вводить и выполнять команды вам придется как и прежде:

```
$ set -o emacs
$ echo hello
```

```
hello
$ pwd
/users/pat
$
```

Но на этот раз вводятся команды построчного редактора `emacs`, чтобы воспользоваться им для редактирования строк. Команды редактора `emacs` представляют собой *управляющие* символы, вводимые одновременным нажатием клавиши <Ctrl> и клавиши соответствующего символа, или же символы, вводимые после нажатия клавиши <Esc>. Команды редактора `emacs` могут быть введены в любой удобный момент, поскольку в данном случае отсутствуют отдельные "режимы", как в редакторе `vi`. Следует, однако, иметь в виду, что ввод команд редактора `emacs` *не* завершается нажатием клавиши <Enter>. (Полный перечень команд редактора `emacs` можно найти в документации для оболочки Bash или Korn.)

Выясним сначала, каким образом курсор перемещается в командной строке. Так, при нажатии комбинации клавиш <Ctrl+b> курсор перемещается влево (или назад), а при нажатии комбинации клавиш <Ctrl+f> — вправо (или вперед). Попробуйте нажать комбинации клавиш <Ctrl+b> и <Ctrl+f> несколько раз, вводя команду в командной строке. В итоге курсор должен переместиться в строке. Если же вы попытаетесь переместить курсор за левый или правый край строки, оболочка просто проигнорирует введенную вами команду. Ниже приведен наглядный пример редактирования строк в режиме работы редактора `emacs`.

```
$ mary had a little larb_      Ctrl+b     $ mary had a little larb
$ mary had a little larb       Ctrl+b     $ mary had a little larb
$ mary had a little larb       Ctrl+b     $ mary had a little larb
$ mary had a little larb       Ctrl+f     $ mary had a little larb
```

Как только курсор окажется на символе, который требуется заменить, нажмите комбинацию клавиш <Ctrl+d>, чтобы удалить текущий символ, как показано ниже. Обратите внимание на то, что после удаления символа `r` курсор сместился влево к символу `b`, который стал текущим.

```
$ mary had a little larb       Ctrl+d      $ mary had a little lab
```

Чтобы добавить символы в командной строке, достаточно ввести их. Такие символы вводятся *до* текущего символа, как показано ниже. Но текущий символ всегда удаляется *слева* от курсора, для чего достаточно нажать клавишу <Backspace> или комбинацию клавиш <Ctrl+h>.

```
$ mary had a little lab        m          $ mary had a little lamb
$ mary had a little lamb       m          $ mary had a little lammb
$ mary had a little lammb      Ctrl+h     $ mary had a little lamb
```

Команды, вызываемые нажатием комбинаций клавиш <Ctrl+a> и <Ctrl+e>, могут быть использованы для перемещения курсора в начало и в конец командной строки соответственно.

```
$ mary had a little lamb      Ctrl+a      $ mary had a little lamb
$ mary had a little lamb      Ctrl+e      $ mary had a little lamb_
```

Следует также иметь в виду, что команда, вызываемая нажатием комбинации клавиш <Ctrl+e>, перемещает курсор на одну позицию вправо от последнего символа в строке. (Если вы не находитесь в режиме редактирования строк в редакторе `emacs`, курсор всегда устанавливается в конце строки, т.е. на одну позицию вправо от последнего введенного символа.)

Это удобно потому, что когда курсор находится в конце строки, все, что будет введено дальше, будет присоединено к тому, что уже находится в строке, как показано ниже.

```
$ mary had a little lamb_        da     $ mary had a little lambda_
```

Для перемещения курсора удобны также команды, вызываемые нажатием комбинаций клавиш <Esc+f> и <Esc+b>. Так, при нажатии комбинации клавиш <Esc+f> курсор перемещается вперед к концу текущего слова, а при нажатии комбинации клавиш <Esc+b> — назад в начало предыдущего слова, как демонстрируется в приведенном ниже примере. Следует, однако, иметь в виду, что для выполнения подобных команд следует *сначала* нажать и отпустить клавишу <Esc>, а *затем* нажать соответствующую клавишу (<f>, <b> и т.д.).

```
$ mary had a little lambda_     Esc b    $ mary had a little lambda
$ mary had a little lambda      Esc b    $ mary had a little lambda
$ mary had a little lambda      Esc b    $ mary had a little lambda
$ mary had a little lambda      Esc f    $ mary had a little lambda
$ mary had a little lambda      Esc f    $ mary had a little lambda
```

Клавишу <Enter> можно нажать в любой момент. После этого текущая строка будет выполнена как команда.

```
$ mary had a little_lambda          Нажать клавишу <Enter>;
                                    ввести команду
ksh: mary: not found
$ _
```

Доступ к командам из предыстории

До сих пор было показано, как редактировать текущую командную строку, но в оболочке ведется предыстория недавно введенных команд. Для доступа к этим командам из предыстории можно воспользоваться специальными командами редактора `emacs`. Так, если нажать комбинацию клавиш <Ctrl+p>, текущая строка на терминале будет заменена *предыдущей* командой из списка в предыстории, а

курсор расположится в конце строки. Аналогичное действие произойдет, если нажать комбинацию клавиш <Ctrl+n>, но на этот раз произойдет замена на *следующую* команду из списка в предыстории.

Допустим, были введены следующие команды:

```
$ pwd
/users/pat
$ cd /tmp
$ echo this is a test
this is a test
$ _
```

А теперь нажмите комбинацию клавиш <Ctrl+p>, чтобы получить предыдущую команду из списка в предыстории:

```
$ _                    Ctrl+p          $ echo this is a test_
```

Всякий раз, когда нажимается комбинация клавиш <Ctrl+p>, текущая строка заменяется предыдущей строкой из списка команд в предыстории:

```
$ echo this is a test_      Ctrl+p      $ cd /tmp_
$ cd /tmp_                  Ctrl+p      $ pwd_
```

Чтобы выполнить команду, отображаемую в командной строке, достаточно нажать клавишу <Enter>.

```
$ pwd_              Нажать клавишу <Enter>
/tmp
$ _
```

Команда, вызываемая нажатием комбинации клавиш <Ctrl+r>, служит для поиска в предыстории тех команд, которые содержат указанную символьную строку. Так, если нажать комбинацию клавиш <Ctrl+r>, ввести символьную строку в качестве шаблона для поиска и затем нажать клавишу <Enter>, оболочка произведет поиск в предыстории, чтобы найти самую последнюю из выполнявшихся команду, содержащую указанную строку. Если искомая команда найдена, она отображается, а иначе оболочка выдаст предупреждающий сигнал.

Как только будет нажата комбинация клавиш <Ctrl+r>, оболочка заменит текущую строку приглашением ^R на ввод символьной строки в качестве шаблона для поиска:

```
$ _                    Ctrl+r  test      $ ^Rtest_
```

Поиск начнется, как только будет нажата клавиша <Enter>:

```
$ ^Rtest_                  Enter       $ echo this is a test_
```

Чтобы выполнить команду, отображаемую в командной строке в результате поиска, необходимо еще раз нажать клавишу <Enter>, как показано ниже. А для того чтобы продолжить поиск команд в предыстории, следует нажать комбинацию клавиш <Ctrl+r> и затем клавишу <Enter>.

```
$ echo this is a test_              Нажать клавишу <Enter> снова
this is a test
$ _
```

В оболочке Bash команда, вызываемая нажатием комбинации клавиш <Ctrl+r>, обрабатывается несколько иначе. После нажатия комбинации клавиш <Ctrl+r> оболочка Bash заменит текущую строку на `(reverse-i-search)`':`

```
$ _                          Ctrl+r       (reverse-i-search)`': _
```

По мере ввода текста строка, ограниченная знаками ` и ', обновляется вводимым текстом, а остальная часть строки обновляется совпадающей командой:

```
(reverse-i-search)`': _              c    (reverse-i-search)`c': echo this is a
test
(reverse-i-search)`c': echo this is a test  d  (reverse-i-search)`cd': cd /tmp
```

Обратите внимание, как оболочка Bash выделяет совпадающую часть команды, устанавливая на ней курсор. Как и в оболочке Korn, найденная команда выполняется после нажатия клавиши <Enter>.

Найдя команду в предыстории, будь то с помощью комбинации клавиш <Ctrl+p>, <Ctrl+n> или <Ctrl+r>, ее можно далее отредактировать, воспользовавшись рассмотренными ранее командами редактора `emacs`. Как и в режиме редактирования строк в редакторе `vi`, в данном случае изменяется не оригинал, а копия команды из предыстории, которая вводится после правки в предысторию нажатием клавиши <Enter>. Основные команды редактирования строк в редакторе `emacs` сведены в табл. 14.2.

Таблица 14.2. Основные команды редактирования строк в редакторе `emacs`

Команда	Назначение
<Ctrl+b>	Переместить курсор на один символ влево
<Ctrl+f>	Переместить курсор на один символ вправо
<Esc+f>	Переместить курсор на одно слово вперед
<Esc+b>	Переместить курсор на одно слово назад
<Ctrl+a>	Переместить курсор в начало строки
<Ctrl+e>	Переместить курсор в конец строки
<Ctrl+d>	Удалить текущий символ
<Esc+d>	Удалить текущее слово

Окончание табл. 14.2

Команда	Назначение
Символ стирания	Удалить предыдущий символ (***символ стирания*** определяется пользователем; как правило, это знак # или комбинация клавиш <Ctrl+h>)
<Ctrl+p>	Получить предыдущую команду из предыстории
<Ctrl+n>	Получить следующую команду из предыстории
<Ctrl+r> ***строка***	Найти в предыстории самую последнюю команду, содержащую указанную ***строку***

Другие способы доступа к предыстории команд

Имеются и другие способы доступа к предыстории команд, о которых стоит знать на тот случай, если вам не подойдет ни один из рассмотренных выше режимов редактирования строк в редакторе `vi` или `emacs`.

Команда `history`

Чтобы получить доступ к предыстории команд, проще всего ввести команду `history`:

```
$ history
507   cd shell
508   cd ch15
509   vi int
510   ps
511   echo $HISTSIZE
512   cat $ENV
513   cp int int.sv
514   history
515   exit
516   cd shell
517   cd ch16
518   vi all
519   run -n5 all
520   ps
521   lpr all.out
522   history
```

Слева от команд указаны их относительные номера. Так, команда номер 1 считается первой или самой давней в предыстории.

Но следует иметь в виду, что действие команды `history` несколько отличается в оболочках Korn и Bash. Так, в оболочке Korn команда `history` направляет в стандартный вывод 16 последних команд, тогда как в оболочке Bash — *всю* предысторию команд, даже если она состоит из 500 или 1000 строк.

Если вы работаете в оболочке Bash и не желаете выводить сразу всю предысторию, укажите в качестве аргумента количество одновременно отображаемых команд, как показано в следующем примере:

```
$ history 10
  513   cp int int.sv
  514   history
  515   exit
  516   cd shell
  517   cd ch16
  518   vi all
  519   run -n5 all
  520   ps
  521   lpr all.out
  522   history 10
$
```

Команда `fc`

Эта команда позволяет запустить редактор по одной команде или больше из предыстории или направляет список команд из предыстории на терминал. Она действует аналогично команде `history`, но только в более удобной форме, где с помощью параметра **`-l`** можно указать количество выводимых из предыстории команд. Например, команда

```
fc -l 510 515
```

направляет в стандартный вывод команды под номерами **510–515**, а команда

```
fc -n -l -20
```

последние 20 команд, опуская их номера (с помощью параметра **`-n`**). Допустим, что вы только что выполнили длинную команду и решили, что было бы неплохо преобразовать введенную командную строку в программу оболочки под названием `runx`. Используя команду `fc`, вы можете получить команду из своей предыстории и переадресовать ввод-вывод, чтобы записать данную команду в файл, как показано ниже.

```
fc -n -l -1 > runx
```

В данном примере число **`-1`**, указанное после параметра **`l`**, обозначает извлечение самой последней команды из предыстории, т.е. с отступом на одну команду от текущей команды вглубь предыстории. Более подробно команда `fc` рассматривается в приложении А.

Команда `r`

Эта простая команда оболочки Korn позволяет повторно выполнять предыдущие команды нажатием нескольких клавиш. Достаточно ввести команду `r`, чтобы оболочка Korn повторно выполнила последнюю команду, как демонстрируется в следующем примере:

```
$ date
Thu Oct 24 14:24:48 EST 2002
$ r                    Повторно выполнить предыдущую команду
date
Thu Oct 24 14:25:13 EST 2002
$
```

Как только вы введете команду `r`, оболочка Korn повторно отобразит предыдущую команду и сразу же выполнит ее. Если вы укажете в команде `r` имя команды в качестве аргумента, оболочка Korn повторно выполнит самую последнюю из вашей предыстории команду, *начинающуюся* с заданного шаблона. И в этом случае оболочка Korn сначала выведет командную строку из предыстории, а затем выполнит ее автоматически, как показано ниже.

```
$ cat docs/planA
...
$ pwd
/users/steve
$ r cat                Возвратить последнюю команду cat
cat docs/planA
$
```

И наконец, команда `r` позволяет заменить первое вхождение одной символьной строки следующим ее вхождением. Чтобы повторно выполнить последнюю команду `cat` для вывода содержимого файла `planB` вместо файла `planA`, достаточно ввести следующее:

```
$ r cat planA=planB
cat docs/planB
...
$
```

а еще проще:

```
$ r cat A=B
cat docs/planB
...
$
```

Аналогичная сокращенная команда имеется и в оболочке Bash. Так, команда `!`*строка* позволяет произвести поиск в предыстории, а команда `!!` — повторно выполнить предыдущую команду, как показано ниже. Следует, однако, иметь в виду, что знак `!` и указанная *строка* вводятся подряд без пробела.

```
$ !!
cat docs/planB
...
$ !d
date
Thu Oct 24 14:39:40 EST 2002
$
```

Команду `fc` можно ввести с параметром **`-s`**, чтобы выполнить ту же самую операцию в любой оболочке, совместимой со стандартом POSIX, как показано ниже. А команда `r`, по существу, служит *псевдонимом* команды `fc` в оболочке Korn (подробнее об этом — далее в главе).

```
$ fc -s cat
cat docs/planB
...
$ fc -s B=C
cat docs/planC
...
$
```

Функции

В оболочках Bash и Korn поддерживаются функции, не доступные в оболочке, совместимой со стандартом POSIX. Рассмотрим их более подробно в последующих разделах.

Локальные переменные

В оболочках Bash и Korn функции могут иметь локальные переменные, благодаря которым функции могут быть сделаны рекурсивными. Эти переменные определяются с помощью команды `typeset`, как демонстрируется ниже. Если переменная с таким же самым именем уже существует, она сохраняется при выполнении команды `typeset` и восстанавливается при выходе из функции.

```
typeset i j
```

Приобретя некоторый опыт работы в оболочке, вы можете разработать ряд функций, чтобы постоянно пользоваться ими в интерактивных сеансах работы в оболочке. Такие функции удобнее всего разместить в файле `ENV`, чтобы они определялись всякий раз, когда вы запускаете новую оболочку.

Автоматически загружаемые функции

В оболочке Korn можно установить специальную переменную `FPATH` аналогично переменной окружения `PATH`. Если затем попытаться выполнить функцию, которая еще не определена, оболочка Korn произведет поиск файла по имени данной функции в разделяемом двоеточиями списке каталогов, указанных в переменной `FPATH`. Если оболочка обнаружит такой файл, она выполнит его, предполагая, что в нем находится определение искомой функции.

Целочисленные арифметические операции

В оболочках Bash и Korn поддерживается вычисление арифметических выражений без специального развертывания арифметического аппарата. С этой целью используется синтаксис, аналогичный конструкции `$((...))`, только без

знака $. А поскольку развертывание арифметического аппарата не выполняется, то подобную конструкцию можно употреблять непосредственно как команду. Ниже приведен характерный тому пример.

```
$ x=10
$ ((x = x * 12))
$ echo $x
120
$
```

Подлинная ценность рассматриваемой здесь конструкции состоит в том, что она позволяет применять арифметические операции в операторах `if`, `while` и `until`. В операциях сравнения устанавливается ненулевое значение кода завершения в том случае, если результат сравнения окажется ложным, и нулевое значение, если результат сравнения окажется истинным. Так, если написать следующую конструкцию:

```
(( i == 100 ))
```

то по своему действию она будет равнозначна проверке на равенство значения переменной `i` величине **100** и установке соответствующего кода завершения. Благодаря этому операции целочисленной арифметики идеально подходят для условных выражений в операторе `if`, как показано в следующем примере:

```
if (( i == 100 ))
then

fi
```

В данном примере конструкция `(( i == 100 ))` возвращает нулевой код завершения (истинное значение), если значение переменной `i` равно величине **100**, а иначе — единичный код завершения (ложное значение), что равнозначно написанию следующего блока кода:

```
if [ "$i" -eq 100 ]
then

fi
```

В данном случае возвращается логическое значение `TRUE`, если результат деления на **10** значения переменной `i` не равен нулю. Операциями целочисленной арифметики можно выгодно пользоваться и в цикле `while`. Так, в следующем примере:

```
x=0
while ((x++ < 100))
do
      команды
done
```

указанные *команды* выполняются **100** раз.

Целочисленные типы данных

В оболочках Korn и Bash поддерживаются целочисленные типы данных. В частности, переменные можно объявить целочисленными, введя команду `typeset` с параметром **`-i`** в следующей форме:

```
typeset -i переменные
```

где *переменные* — любые имена переменных, достоверные в оболочке. Первоначальные значения могут быть присвоены переменным в тот момент, когда они объявляются:

```
typeset -i signal=1
```

Главное преимущество такого подхода заключается в том, что арифметические операции выполняются над целочисленными переменными с помощью конструкции `((...))` быстрее, чем над нецелочисленными переменными. Тем не менее целочисленной переменной нельзя присвоить ничего, кроме целочисленного значения или выражения. Если же попытаться присвоить ей нецелочисленное значение, оболочка выдаст сообщение `"bad number"` (неверное число):

```
$ typeset -i i
$ i=hello
ksh: i: bad number
```

В оболочке Bash игнорируются любые символьные строки, которые не содержат числовые значения, и генерируется ошибка, если они содержат не только числа, но и другие символы:

```
$ typeset -i i
$ i=hello
$ echo $i
0
$ i=1hello
bash: 1hello: value too great for base (error token is "1hello")
$ i=10+15
$ echo $i
25
$
```

В приведенном выше примере показано также, что целочисленные выражения могут быть присвоены целочисленным переменным даже без применения конструкции `((...))`. Это справедливо для обеих рассматриваемых здесь оболочек.

Числа с разным основанием системы счисления

В оболочках Korn и Bash допускается также выполнять арифметические операции над числами с разным основанием системы счисления. Для представления в этих оболочках чисел в другой системе счисления служит следующее обозначение:

```
основание#число
```

Например, чтобы выразить десятичное значение **100** в восьмеричной системе счисления (с основанием **8**), можно написать следующую строку:

```
8#100
```

Значения констант могут быть записаны в разных системах счисления везде, где допускается указывать целочисленное значение. Например, чтобы присвоить восьмеричное значение **100** целочисленной переменной `i`, можно написать следующую команду:

```
typeset -i i=8#100
```

Однако основание системы счисления первого значения, присваиваемого целочисленной переменной в оболочке Korn, по умолчанию устанавливается для всех последующих значений этой же переменной. Иными словами, если первое значение, присваиваемое целочисленной переменной `i`, является восьмеричным, то при любой последующей ссылке на эту переменную в командной строке оболочка Korn отобразит ее значение как восьмеричное число, используя обозначение `8#`*значение*. Соответствующий пример приведен ниже.

```
$ typeset -i i=8#100
$ echo $i
8#100
$ i=50
$ echo $i
8#62
$ (( i = 16#a5 + 16#120 ))
$ echo $i
8#705
$
```

В данном примере первое значение, присваиваемое переменной `i`, оказывается восьмеричным (**8#100**), и поэтому при всех последующих ссылках на переменную `i` будет также получено восьмеричное значение. А когда этой переменной присваивается значение **50** по основанию **10**, оно все равно отображается как равнозначное восьмеричное значение **8#62**.

В приведенном выше примере проявляется следующая любопытная особенность: несмотря на то, что значение переменной `i` отображается как восьмеричное, основание системы счисления значений, присваиваемых этой переменной, по-прежнему остается десятичным, если не указано иное. Другими словами, выражение `i=50` не равнозначно выражению `i=8#50`, даже если оболочке известно, что обращаться к значению переменной `i` следует по основанию 8.

В предыдущем примере конструкция `((...))` применяется для ввода двух шестнадцатеричных значений — **a5** и **120**. А полученный результат все равно отображается в восьмеричной форме. Следует, однако, признать, что данная особенность едва заметна и редко проявляется в повседневном программировании на языке оболочки или ее применении в интерактивном режиме!

В оболочке Bash применяется как синтаксис *основание*#*число* для указания произвольных оснований систем счисления, так и синтаксис языка C для представления восьмеричных и шестнадцатеричных чисел. При этом восьмеричные числа предваряются префиксом **0** (нуль), а шестнадцатеричные — префиксом **0x**, как демонстрируется в следующем примере:

```
$ typeset -i i=0100
$ echo $i
64
$ i=0x80
$ echo $i
128
$ i=2#1101001
$ echo $i
105
$ (( i = 16#a5 + 16#120 ))
$ echo $i
453
$
```

В отличие от оболочки Korn, в оболочке Bash не отслеживается основание системы счисления числовых значений переменных, и поэтому значения целочисленных переменных отображаются в десятичной форме. Впрочем, для вывода целочисленных значений в восьмеричной или шестнадцатеричной форме можно всегда воспользоваться командой `printf`.

Как видите, оболочки Bash и Korn свободно обращаются с разными основаниями систем счисления, что дает возможность создавать функции, выполняющие преобразование основания системы счисления и арифметические операции в недесятичной форме.

Команда `alias`

Псевдоним служит кратким обозначением, предоставляемым оболочкой с целью разрешить специализацию команд. В оболочке поддерживается список псевдонимов, которые обнаруживаются при вводе команд прежде любых других подстановок. Если первым словом в команде оказывается псевдоним, оно заменяется текстом данного псевдонима.

Псевдоним определяется с помощью команды `alias`, общая форма которой выглядит так:

```
alias имя=строка
```

где *имя* — это имя псевдонима, а *строка* — любая символьная строка. Например, в следующей команде:

```
alias ll='ls -l'
```

псевдониму ll присваивается команда ls -l. Если теперь пользователь введет команду ll, оболочка негласно заменит ее командой ls -l. А еще лучше ввести аргументы после имени псевдонима в командной строке, как демонстрируется в следующем примере:

```
ll *.c
```

где после подстановки псевдонима получается приведенная ниже команда.

```
ls -l *.c
```

Оболочка выполняет обычную обработку командной строки как при установке псевдонима, так и при его использовании, поэтому заключение в кавычки может быть затруднено. В качестве примера напомним, что оболочка отслеживает текущий рабочий каталог в переменной PWD:

```
$ cd /users/steve/letters
$ echo $PWD
/users/steve/letters
$
```

В таком случае можно создать псевдоним dir, предоставляющий базовое имя текущего рабочего каталога, используя переменную PWD и одну из конструкций для подстановки параметров следующим образом:

```
alias dir="echo ${PWD##*/}"
```

Такой прием кажется вполне обоснованным, но давайте посмотрим, каким образом этот псевдоним действует на практике:

```
$ alias dir="echo ${PWD##*/}"          Определить псевдоним
$ pwd                                  Выяснить местоположение
/users/steve
$ dir                                  Выполнить псевдоним
steve
$ cd letters                           Сменить каталог
$ dir                                  Выполнить псевдоним снова
steve
$ cd /usr/spool                        Сделать еще одну попытку
$ dir
steve
$
```

Независимо от текущего каталога, под псевдонимом dir выводится каталог steve. Дело в том, что при определении псевдонима dir кавычки были указаны неаккуратно. Если вспомнить, что оболочка выполняет подстановку параметра, указанного в двойных кавычках, то оказывается, что в момент определения псевдонима оболочка вычислила следующее выражение:

```
${PWD##*/}
```

Это означает, что псевдоним dir был, по существу, определен так, как будто бы мы ввели его следующим образом:

```
$ alias dir="echo steve"
```

Поэтому и не удивительно, что он действует неверно! В качестве выхода из данного положения, определяя псевдоним dir, можно указать одиночные кавычки вместо двойных, чтобы отложить подстановку параметра до тех пор, пока не будет вычислен псевдоним:

```
$ alias dir='echo ${PWD##*/}'          Определить псевдоним
$ pwd                                  Выяснить местоположение
/users/steve
$ dir                                  Выполнить псевдоним
steve
$ cd letters                           Сменить каталог
$ dir                                  Выполнить псевдоним снова
letters
$ cd /usr/spool                        Сделать еще одну попытку
$ dir
spool
$
```

Если псевдоним оканчивается пробелом, то слово, следующее после псевдонима, также проверяется на подстановку псевдонима. Так, в следующем примере:

```
alias nohup="/bin/nohup "
nohup ll
```

оболочка проверяет псевдоним в символьной строке ll после подстановки значения /bin/nohup вместо псевдонима nohup.

Заключение команды в кавычки или предварение ее знаком обратной косой черты предотвращает подстановку псевдонима. Например:

```
$ 'll'
ksh: ll: command not found
$
```

В форме

```
alias имя
```

перечисляется значение псевдонима *имя*, а команда alias выполняется без аргументов, перечисляя все псевдонимы. Приведенные ниже псевдонимы определяются автоматически при запуске оболочки Korn.

```
autoload='typeset -fu'
functions='typeset -f'
history='fc -l'
integer='typeset -i'
```

```
local=typeset
nohup='nohup '
r='fc -e -'
suspend='kill -STOP $$'
```

В приведенном выше примере r фактически служит псевдонимом команды fc с параметром **-e**, а history — псевдонимом команды fc с параметром **-l**. Для сравнения в оболочке Bash по умолчанию ни один из псевдонимов не определяется автоматически.

Удаление псевдонимов

Команда unalias служит для удаления псевдонимов из списка. Так, следующая форма команды unalias:

```
unalias имя
```

служит для удаления псевдонима *имя*, а приведенная ниже команда — всех псевдонимов.

```
unalias -a
```

Если разработать ряд определений псевдонимов, чтобы пользоваться ими в сеансах работы в оболочке, то их можно определить в файле ENV таким образом, чтобы они были всегда доступны для применения. Ведь иначе они переместятся в подоболочки.

Массивы

В оболочках Korn и Bash предоставляются ограниченные возможности манипулирования массивами. Массивы в оболочке Bash могут содержать неограниченное количество элементов, которое фактически ограничивается доступным объемом оперативной памяти. А в оболочке Korn размеры массивов ограничиваются 4096 элементами. Индексация массивов в обеих оболочках начинается с нуля.

Элемент доступен в массиве по *индексу*, которым может быть целочисленное выражение, заключаемое в квадратные скобки. Вместо того чтобы объявлять максимальный размер массива в оболочке, можно просто присвоить значения его элементам по мере надобности. Присваиваемые значения ничем не отличаются от обычных значений переменных, как демонстрируется в следующем примере:

```
$ arr[0]=hello
$ arr[1]="some text"
$ arr[2]=/users/steve/memos
$
```

Чтобы извлечь элемент из массива, необходимо указать имя массива, открывающую скобку, номер элемента и закрывающую скобку. Вся эта конструкция должна быть заключена в пару фигурных скобок, а перед этим следует указать знак денежной единицы. Кажется сложно? На самом деле несложно, как показано в следующем примере:

```
$ echo ${array[0]}
hello
$ echo ${array[1]}
some text
$ echo ${array[2]}
/users/steve/memos
$ echo $array
hello
$
```

Как следует из приведенного выше примера, если индекс массива не указан, то используется нулевой элемент. А если не указать фигурные скобки при выполнении подстановки, то будет получен не тот результат, который предполагался:

```
$ echo $array[1]
hello[1]
$
```

Сначала в данном примере подставляется значение переменной `array` (т.е. значение `hello` элемента массива `array[0]`), а затем оно отображается вместе со значением `[1]` по команде `echo`. Но поскольку оболочка подставляет имя файла после подстановки переменной, она попытается сопоставить шаблон `hello[1]` с файлами в текущем каталоге.

Конструкция `[*]` может быть использована в качестве индекса массива для получения всех его элементов в командной строке, где каждый элемент разделяется пробелом:

```
$ echo ${array[*]}
hello some text /users/steve/memos
$
```

Конструкция `${#`*`array`*`[*]}` может служить для поиска количества элементов в указанном массиве *`array`* следующим образом:

```
$ echo ${#array[*]}
3
$
```

Выводимое подобным способом число фактически обозначает количество значений, хранящихся в элементах массива, а не наибольший индекс, применяемый для хранения элемента в массиве.

```
$ array[10]=foo
$ echo ${array[*]}            Отобразить все элементы
hello some text /users/steve/memos foo
$ echo ${#array[*]}           Количество элементов в массиве
4
$
```

Массив, в котором содержатся несмежные значения, называется *разреженным*. Например, массив целых чисел можно определить, указав его имя в команде `typeset` с параметром **`-i`** следующим образом:

```
typeset -i data
```

Целочисленные вычисления могут быть выполнены над элементами массива с помощью конструкции `((...))`:

```
$ typeset -i array
$ array[0]=100
$ array[1]=50
$ (( array[2] = array[0] + array[1] ))
$ echo ${array[2]}
150
$ i=1
$ echo ${array[i]}
50
$ array[3]=array[0]+array[2]
$ echo ${array[3]}
250
$
```

Однако знак **$** и фигурные скобки можно опустить не только при ссылке на элементы массива в двойных круглых скобках, но и за их пределами при объявлении массива как относящегося к целочисленному типу. А кроме того, знаки **$** не обязательно указывать перед переменными, употребляемыми в индексных выражениях.

Ниже в качестве примера приведена программа `reverse`, которая сначала читает строки из стандартного ввода, а затем направляет их в стандартный вывод, но в обратном порядке. Первый цикл `while` в данной программе выполняется до конца файла или до тех пор, пока не будет прочитано 4096 строк (т.е. тот предел, который накладывается на размеры массивов в оболочке Korn).

```
$ cat reverse
# прочитать строки в массив buf

typeset -i line=0

while (( line < 4096 )) && read buf[line]
do
      (( line = line + 1 ))
done
```

```
# а теперь вывести строки в обратном порядке

while (( line > 0 )) do
      (( line = line - 1 ))
      echo "${buf[line]}"
done

$ reverse
line one
line two
line three
<Ctrl+d>
line three
line two
line one
$
```

В еще одном, приведенном ниже примере определяется функция `cdh()`, не только изменяющая текущий каталог, но и применяющая массив для ведения предыстории предыдущих каталогов. Это дает пользователю возможность вывести из предыстории список каталогов и перейти в любой каталог из этого списка.

```
$ cat cdh
CDHIST[0]=$PWD       # инициализировать элемент массива CDHIST[0]

cdh ()
{
      typeset -i cdlen i
      if [ $# -eq 0 ] ; then     # если аргументы отсутствуют, то
                                 # по умолчанию установить начальный
                                 # каталог в переменной HOME
            set -- $HOME
      fi

      cdlen=${#CDHIST[*]}        # количество элементов
                                 # в массиве CDHIST
      case "$@" in
      -l)                        # вывести список каталогов
            i=0
            while ((i < cdlen))
            do
                  printf "%3d %s\n" $i ${CDHIST[i]}
                  ((i = i + 1))
            done
            return ;;
      -[0-9]|-[0-9][0-9])        # перейти в каталог из списка
            i=${1#-}             # удалить начальный дефис '-'
            cd ${CDHIST[i]} ;;
      *)                         # перейти в новый каталог
            cd $@ ;;
      esac

      CDHIST[cdlen]=$PWD
}
$
```

В массиве CDHIST сохраняется каждый каталог, посещаемый с помощью функции cdh(). А первым в этом массиве является элемент CDHIST[0], который инициализируется текущим каталогом, когда данная функция начинает выполняться из файла cdh:

```
$ pwd
/users/pat
$ . cdh              Определить функцию cdh()
$ cdh /tmp
$ cdh -l
  0 /users/pat
  1 /tmp
$
```

При первом запуске файла cdh на выполнение первому элементу массива CDHIST[0] был присвоен путь к каталогу /users/pat, а затем определена функция cdh(). В результате выполнения команды cdh /tmp переменной cdlen было присвоено количество элементов в массиве CDHIST (в данном случае один элемент), тогда как второму элементу массива CDHIST[1] — указанный путь к каталогу /tmp. А в результате выполнения команды cdh -l содержимое каждого элемента массива CDHIST было выведено по команде printf (в данном случае в переменой cdlen было установлено значение **2**, поскольку два первых элемента массива CDHIST по индексу **0** и **1** содержали данные).

Обратите внимание на блок условного оператора if в начале рассматриваемой здесь функции, где в переменной $1 устанавливается значение из переменной $HOME (т.е. путь к начальному каталогу пользователя), если данная функция вызвана без аргументов. Попробуем выполнить ее еще раз.

```
$ cdh
$ pwd
/users/pat
$ cdh -l
  0 /users/pat
  1 /tmp
  2 /users/pat
$
```

Функция вроде бы сработала нормально, но дважды вывела каталог /users/pat. (Исправьте этот недостаток самостоятельно.)

Самым полезным в команде cdh является параметр ***-n***, позволяющий сменить текущий каталог на указанный в списке, как показано ниже.

```
$ cdh /usr/spool/uucppublic
$ cdh -l
  0 /users/pat
  1 /tmp
  2 /users/pat
  3 /usr/spool/uucppublic
```

```
$ cdh -1
$ pwd
/tmp
$ cdh -3
$ pwd
/usr/spool/uucppublic
$
```

Команда cdh заменяет стандартную команду cd, поскольку в ней производится поиск псевдонима перед выполнением встроенных команд. Если же создать псевдоним cd команды cdh, то в конечном итоге получится усовершенствованная команда cd. Но для этого каждый вызов команды cd придется заключить в одиночные кавычки в теле функции cdh(), предотвратив тем самым нежелательную рекурсию, как показано ниже.

```
$ cat cdh
CDHIST[0]=$PWD       # инициализировать элемент массива CDHIST[0]

cdh ()
{
      typeset -i cdlen i
      if [ $# -eq 0 ] ; then     # если аргументы отсутствуют, то
                                 # по умолчанию установить начальный
                                 # каталог в переменной HOME
            set -- $HOME
      fi

      cdlen=${#CDHIST[*]}        # количество элементов
                                 # в массиве CDHIST
      case "$@" in
      -l)                        # вывести список каталогов
            i=0
            while ((i < cdlen))
            do
                  printf "%3d %s\n" $i ${CDHIST[i]}
                  ((i = i + 1))
            done
            return ;;
      -[0-9]|-[0-9][0-9])        # перейти в каталог из списка
            i=${1#-}             # удалить начальный дефис '-'
            'cd' ${CDHIST[i]} ;;
      *)                         # перейти в новый каталог
            'cd' $@ ;;
      esac

      CDHIST[cdlen]=$PWD
}
$

$ . cdh              Определить функцию cdh() и псевдоним cd
$ cd /tmp
```

```
$ cd -l
  0 /users/pat
  1 /tmp
$ cd /usr/spool
$ cd -l
  0 /users/pat
  1 /tmp
  2 /usr/spool
$
```

В табл. 14.3 сведены различные конструкции массивов, которые поддерживаются в оболочках Korn и Bash.

Таблица 14.3. Конструкции массивов

Конструкция	Назначение
${*array*[i]}	Подставить значение **i**-го элемента указанного массива
$*array*	Подставить значение первого элемента указанного массива (***array*[0]**)
${*array*[*]}	Подставить значения всех элементов указанного массива
${#*array*[*]}	Подставить количество элементов указанного массива
array*[i]=*значение	Сохранить заданное ***значение*** в элементе ***array*[i]** указанного массива

Управление заданиями

В оболочке предоставляются средства для непосредственного управления заданиями из командной строки. *Задание* — это любая команда или последовательность команд, выполняемых в оболочке, как демонстрируется в следующем примере:

```
who | wc
```

Когда команда запускается в фоновом режиме (т.е. по ссылке **&**), оболочка выводит номер задания в квадратных скобках (**[]**), а также идентификатора процесса, как показано ниже.

```
$ who | wc &
[1] 832
$
```

По завершении задания оболочка выводит сообщение в следующей форме:

```
[n] + Done     последовательность
```

где *n* — номер завершенного задания; *последовательность* — текст последовательности команд, использованных при создании задании.

Чтобы вывести состояние незавершенных заданий, проще всего воспользоваться командой `jobs` следующим образом:

```
$ jobs
[3] +  Running      make ksh &
[2] -  Running      monitor &
[1]    Running      pic chapt2 | troff > aps.out &
```

Знаками **+** и **-** после номера задания обозначаются текущие и предыдущие задания соответственно. Текущим считается задание, которое было самым последним отправлено на выполнение в фоновый режим, а предыдущим — задание, следующее после него. В целом ряде встроенных команд ради краткости номер текущего или предыдущего задания можно указывать в качестве аргумента.

Например, встроенная в оболочку команда `kill` может быть использована для прерывания выполняющегося задания в фоновом режиме, как демонстрируется в приведенном ниже примере. В качестве ее аргумента можно указать идентификатор процесса или знак процента (**%**), а затем номер задания, знак **+**, обозначающий текущее задание, знак **-**, обозначающий предыдущее задание, или же еще один знак **%**, также обозначающий текущее задание.

```
$ pic chapt1 | troff > aps.out &
[1]      886
$ jobs
[1] +   Running          pic chapt1 | troff > aps.out &
$ kill %1
[1]     Done             pic chapt1 | troff > aps.out &
$
```

В приведенном выше примере команду `kill` можно было бы использовать с параметрами **%+** или **%%** для выполнения того же самого задания. Первые несколько символов из последовательности команд могут быть также использованы для ссылки на задание. Так, для прерывания конкретного задания в приведенном выше примере вполне подошла бы команда `kill %pic`.

Остановленные задания и команды `fg` и `bg`

Если задание выполняется в приоритетном режиме (т.е. без ссылки **&**) и при этом требуется приостановить его, то достаточно нажать комбинацию клавиш <Ctrl+z>. Выполнение задания приостановится, и оболочка выведет сообщение в следующей форме:

```
[n] + Stopped (SIGTSTP)      последовательность
```

Остановленное задание становится текущим. Чтобы продолжить его выполнение, следует воспользоваться командой `fg` или `bg`. В частности, команда `fg` позволяет возобновить выполнение текущего задания в приоритетном режиме, а команда `bg` — в фоновом.

Для обозначения конкретного задания в команде `fg` или `bg` можно также указать номер задания, несколько первых символов из конвейера команд, а также знаки **+**, **-** и **%**. Эти команды выводят также последовательность команд, чтобы напомнить о тех заданиях, которые были перенесены в приоритетный или фоновый режим:

```
$ troff memo | photo
<Ctrl+z>
[1] + Stopped (SIGTSTP)        troff memo | photo
$ bg
[1]       troff memo | photo &
$
```

В приведенном выше примере демонстрируется последовательность команд, чаще всего употребляемая для управления заданиями. Она состоит в приостановке задания, выполняющегося в приоритетном режиме, и его отправке на дальнейшее выполнение в фоновый режим. При попытке прочитать данные с терминала в задании, выполняющемся в фоновом режиме, оно будет приостановлено с выдачей сообщения в следующей форме:

```
[n] + Stopped (SIGTTIN)     последовательность
```

В таком случае задание может быть перенесено в приоритетный режим по команде `fg`, чтобы разрешить ввод данных с терминала. По завершении ввода с терминала задание может быть приостановлено снова (нажатием комбинации клавиш <Ctrl+z>) и перенесено обратно в фоновый режим, чтобы продолжить его выполнение.

Данные, выводимые из задания, выполняющегося в фоновом режиме, как правило, направляются непосредственно на терминал, что может быть неудобно, если в это время пользователь выполняет какое-нибудь другое задание. Этот недостаток можно исправить с помощью команды

```
stty tostop
```

выполнение которой приведет к тому, что любое задание, выполняющееся в фоновом режиме и пытающееся вывести данные на терминал, будет приостановлено с выдачей сообщения в приведенной ниже форме. (В оболочке Bash генерируются несколько иные, чем представленные здесь сообщения, хотя управление заданиями функционирует аналогичным образом.)

```
[n] - Stopped (SIGTTOU)     последовательность
```

В следующем примере демонстрируется, каким образом можно воспользоваться управлением заданиями на практике:

```
$ stty tostop
$ rundb                    Запустить программу базы данных
??? find green red         Найти зеленые и красные объекты
<Ctrl+z>                     Выполнение задания может отнять время
[1] + Stopped rundb
$ bg                       Поэтому перенести задание в фоновый режим
[1]         rundb &
...                        Выполнить какое-нибудь другое задание
$ jobs
[1] + Stopped(tty output)           rundb &
$ fg                Перенести задание обратно в приоритетный режим
rundb
1973    Ford     Mustang          red
1975    Chevy    Monte Carlo      green
1976    Ford     Granada          green
1980    Buick    Century          green
1983    Chevy    Cavalier         red
??? find blue              Найти синие объекты
<Ctrl+z>                   Приостановить задание снова
[1] + Stopped          rundb
$ bg                  Перенести задание обратно в фоновый режим
[1]          rundb &
...                        Продолжить работу до завершения
```

Разные средства

И в завершение этой главы будет рассмотрен ряд других полезных средств, доступных в обеих рассматриваемых оболочках.

Другие возможности команды cd

На первый взгляд команда cd кажется довольно простой, тем не менее, в ее арсенале имеется ряд специальных возможностей. В частности, аргумент - означает "предыдущий каталог" и служит в качестве удобного сокращения. Как показано в следующем примере, с помощью команды cd - можно без особого труда переходить из одного каталога к другому:

```
$ pwd
/usr/src/cmd
$ cd /usr/spool/uucp
$ pwd
/usr/spool/uucp
$ cd -                     Перейти в предыдущий каталог
/usr/src/cmd               Команда cd выводит имя нового каталога
$ cd -
/usr/spool/uucp
$
```

В оболочке Korn команда cd обладает возможностью подставлять отдельные части из пути к текущему каталогу. (Такая возможность не поддерживается ни в

оболочке Bash, ни в оболочке по стандарту POSIX.) Такая форма команды cd выглядит следующим образом:

```
cd прежний новый
```

где предпринимается попытка заменить первое вхождение заданной символьной строки *прежний* (каталог) в пути к текущему каталогу на указанную строку *новый* (каталог). В приведенном ниже примере демонстрируется применение данной формы команды cd на практике.

```
$ pwd
/usr/spool/uucppublic/pat
$ cd pat steve                    Сменить каталог pat на каталог steve
/usr/spool/uucppublic/steve       Команда cd выводит имя нового каталога
$ pwd                             Подтвердить новое местоположение
/usr/spool/uucppublic/steve
$
```

Замена знака тильды

Если слово в командной строке начинается со знака тильды (**~**), оболочка выполнит следующее: если знак тильды является единственным символом в слове или же если после знака тильды следует знак **/**, то в данном месте командной строки подставляется значение переменной HOME. В следующем примере показано, каким образом замена знака тильды осуществляется на практике:

```
$ echo ~
/users/pat
$ qrep Korn ~/shell/chapter9/ksh
The Korn shell is a new shell developed
by David Korn at AT&T
for the Bourne shell would also run under the Korn
the one on System V, the Korn shell provides you with
idea of the compatibility of the Korn shell with Bourne's,
the Bourne and Korn shells.
The main features added to the Korn shell are:
$
```

Если остальная часть слова вплоть до знака косой черты представляет собой регистрационное имя пользователя в файле /etc/passwd, знак тильды в регистрационном имени пользователя заменяется его начальным каталогом из переменной HOME, как демонстрируется в следующем примере кода:

```
$ echo ~steve
/users/steve
$ echo ~pat
/users/pat
$ qrep Korn -pat/shell/chapter9/ksh
The Korn shell is a new shell developed
by David Korn at AT&T
```

```
for the Bourne shell would also run under the Korn
the one on System V, the Korn shell provides you with
idea of the compatibility of the Korn shell with Bourne's,
the Bourne and Korn shells.
The main features added to the Korn shell are:
$
```

Если после знака тильды в командной строке оболочек Korn и Bash следует знак **+** или **–**, то вместо него подставляется значение переменной `PWD` или `OLDPWD` соответственно, как в приведенном ниже примере. Переменные `PWD` и `OLDPWD` устанавливаются по команде `cd` и содержат полные пути к текущему и предыдущему каталогам соответственно. Но в оболочке по стандарту POSIX сочетания символов **~+** и **~–** не поддерживаются.

```
$ pwd
/usr/spool/uucppublic/steve
$ cd
$ pwd
/users/pat
$ echo ~+
/users/pat
$ echo ~-
/usr/spool/uucppublic/steve
$
```

Помимо упомянутых выше подстановок, оболочка проверяет также наличие знака тильды после знака двоеточия (`:`), заменяя знак тильды и в этом случае. Именно поэтому путь вроде `~/bin` правильно интерпретируется в переменной `PATH`.

Порядок поиска

Вводя имя команды, следует иметь в виду следующий порядок поиска, применяемый оболочкой при анализе содержимого командной строки.

1. Сначала оболочка проверяет, является ли введенная команда зарезервированным словом (например, `for` или `do`).
2. Если это не зарезервированное слово и не заключенное в кавычки, оболочка далее проверяет список своих псевдонимов. И если она обнаружит совпадение, то выполнит подстановку. А если определение псевдонима оканчивается пробелом, то оболочка и в этом случае попытается подставить псевдоним в следующем слове. Окончательный результат затем проверяется по списку зарезервированных слов, и если проверяемое слово не зарезервировано, то оболочка перейдет к следующему этапу.
3. Оболочка проверяет команду по списку ее функций и выполняет одноименную функцию, если она обнаружена.

4. Оболочка проверяет, является ли проверяемая команда встроенной, например, `cd` и `pwd`.
5. И наконец, оболочка производит поиск введенной команды в переменной `PATH`.
6. Если введенная команда все еще не найдена, выдается сообщение об ошибке `"command not found"` (команда не найдена).

Краткий итог совместимости оболочек

В табл. 14.4 кратко подытожена совместимость стандартной оболочки POSIX с оболочками Korn и Bash в отношении средств, рассмотренных в этой главе. В этой таблице буквой "X" обозначено поддерживаемое средство, буквами "UP" — дополнительное средство в оболочке по стандарту POSIX, называемое также средством переносимости пользователя (User Portability) в спецификации данной оболочки, а буквами "POS" — средство, которое поддерживается в оболочке Bash только в том случае, если она вызывается по имени `sh`, с параметром командной строки **`--posix`** или после выполнения команды `set -o posix`.

Таблица 14.4. Совместимость оболочек POSIX, Korn и Bash

	Оболочка POSIX	Оболочка Korn	Оболочка Bash
Файл **`ENV`**	X	X	POS
Режим редактирования строк в редакторе **`vi`**	X	X	X
Режим редактирования строк в редакторе **`emacs`**		X	X
Команда **`fc`**	X	X	X
Команда **`r`**			X
`!!`			
`!`*`строка`*			X
Функции	X	X	X
Локальные переменные		X	X
Автоматическая загрузка через переменную **`FPATH`**		X	
Целочисленные выражения, реализуемые с помощью конструкции **`((...))`**		X	X
Целочисленные типы данных		X	X
Представление чисел в разных системах счисления		X	X
Форматы **`0`*`хшестнадцатеричное_число`***, **`0`*`восьмеричное_число`***			X
Псевдонимы	UP	X	X

Окончание табл. 14.4

	Оболочка POSIX	Оболочка Korn	Оболочка Bash
Массивы		X	X
Управление заданиями	UP	X	X
Команда `cd -`	X	X	X
Команда `cd прежний новый`		X	
Комбинации `~имя_пользователя`, `~/`	X	X	X
Комбинации `~+`, `~-`		X	X

A

Краткое изложение оболочки

В этом приложении вкратце излагаются основные средства оболочки POSIX по стандарту IEEE 1003.1-2001.

Запуск

При запуске оболочки в командной строке могут быть заданы такие же параметры, как при вводе команды `set`. Кроме того, могут быть указаны следующие параметры.

`-c` ***команды***	Здесь ***команды*** — это исполняемые команды
`-i`	Интерактивная оболочка, где сигналы **2**, **3** и **15** игнорируются
`-s`	В этом случае команды читаются из стандартного ввода

Команды

Общая форма для ввода команд выглядит следующим образом:

команды аргументы

где *команды* — имя выполняемой программы или команды, тогда как *аргументы* — ее конкретные аргументы. Имя программы или команды и ее аргументы разделяются пробельными символами (как правило, пробелами, знаками табуляции и новой строки), но это положение можно изменить, внеся соответствующие коррективы в переменную `IFS`.

В одной строке можно ввести несколько команд, разделив их точкой с запятой (**`;`**). Каждая выполняемая команда возвращает число, называемое *кодом завершения*, где нулевое значение обозначает удачный исход, а ненулевое — неудачный исход выполнения команды.

Знак **`|`** может служить для связывания стандартного вывода из одной команды со стандартным вводом другой, как демонстрируется в следующем примере:

```
who | wc -l
```

В этом случае код завершения обозначает исход выполнения последней команды в конвейере. Если указать знак **`!`** в начале конвейера, код завершения будет обозначать состояние, противоположное исходу выполнения последней команды в конвейере.

Если последовательность команд оканчивается знаком амперсанда (**&**), она выполняется асинхронно в фоновом режиме. Оболочка выводит на терминал идентификатор процесса и номер задания, присваиваемый команде, а также приглашение на ввод следующей команды в интерактивном режиме. Ввод команды можно продолжить с новой строки, указав знак **** последним в текущей строке.

Знаки **&&** указывают на то, что следующая команда будет выполнена только в том случае, если предыдущая возвратит нулевой код завершения, а знаки **||** — на то, что следующая команда будет выполнена лишь при условии, что предыдущая возвратит ненулевой код завершения. Так, в следующем примере:

```
who | grep "fred" > /dev/null && echo "fred's logged on"
```

команда `echo` будет выполнена лишь в том случае, если команда `grep` возвратит нулевой код завершения.

Комментарии

Если в строке обнаруживается знак **#**, оболочка воспринимает остальную часть строки как комментарий, игнорируя ее в целях интерпретации, подстановки и выполнения.

Переменные оболочки

Имя переменной оболочки должно начинаться с буквенного символа или знака подчеркивания (_), а за ним может следовать любое количество буквенно-цифровых символов или знаков подчеркивания. Переменным оболочки можно присваивать значения в приведенной ниже форме, где указанное *значение не* заменяется на имя файла.

```
переменная=значение переменная=значение ...
```

Позиционные параметры

Всякий раз, когда выполняется программа оболочки, ее имя присваивается переменной `$0`, а ее аргументы, введенные в командной строке, — переменным `$1`, `$2` и т.д., обозначающим соответствующие позиционные параметры. Значения могут быть присвоены позиционным параметрам и с помощью команды `set`. Ссылаться на позиционные параметры 1–9 можно непосредственно, как показано выше, а на позиционные параметры, следующие после 9-го, — в фигурных скобках, как, например: `${10}`.

Специальные параметры

В табл. А.1 перечислены специальные параметры оболочки, обозначаемые соответствующими переменными.

Таблица А.1. Переменные, обозначающие специальные параметры

Параметр	Назначение
`$#`	Количество аргументов, передаваемых программе, или же количество параметров, устанавливаемых с помощью команды **set**
`$*`	Общая ссылка на все позиционные параметры в виде `$1`, `$2` и т.д.
`$@`	То же что, и `$*`, но если заключается в двойные кавычки (`"$@"`), то обозначает общую ссылку на все позиционные параметры в виде `"$1"`, `"$2"` и т.д.
`$0`	Имя выполняемой программы
`$$`	Идентификатор процесса, присваиваемый выполняемой программе
`$!`	Идентификатор процесса, присваиваемый последней программе, отправленной на выполнение в фоновый режим
`$?`	Код завершения последней команды, *не* выполнявшейся в фоновом режиме
`$-`	Действующие в настоящий момент установленные параметры (см. далее описание команды **set**)

Помимо перечисленных выше специальных параметров, в оболочке применяются и другие параметры, обозначаемые соответствующими переменными. Наиболее употребительные из них перечислены в табл. А.2.

Таблица А.2. Другие переменные, используемые в оболочке

Переменная	Назначение
CDPATH	Каталоги для поиска при выполнении команды **cd** без указания полного пути в качестве аргумента
ENV	Имя файла, выполняемого оболочкой в текущей среде, при условии запуска в интерактивном режиме
FCEDIT	Редактор, применяемый в команде **fc**. Если он не задан, применяется редактор **ed**
HISTFILE	Если эта переменная установлена, она обозначает файл, применяемый для хранения предыстории команд. А если эта переменная *не* установлена или же если файл не доступен для записи, то применяется файл **$HOME/.sh_history**
HISTSIZE	Если эта переменная установлена, она обозначает количество введенных ранее команд, доступных для редактирования. По умолчанию это количество оказывается не меньше **128**
HOME	Начальный каталог пользователя, в который происходит переход при выполнении команды **cd** без аргументов
IFS	Внутренний разделитель полей, употребляемый в оболочке для разделения слов при синтаксическом анализе командной строки для команд **read** и **set**; при подстановке результатов, выводимых из команды, заключаемой в кавычки; а также при подстановке параметров. Как правило, эта переменная содержит три символа: пробела, горизонтальной табуляции и новой строки

Окончание табл. А.2

Переменная	Назначение
`LINENO`	В этой переменной оболочка устанавливает номер строки выполняемого сценария или программы. Значение этой переменной устанавливается перед выполнением строки и начинается с `1`
`MAIL`	Имя файла, в котором оболочка периодически проверяет поступающую почту. Если поступит новая почта, оболочка выводит сообщение `"You have mail"` (У вас есть почта). См. также переменные `MAILCHECK` и `MAILPATH`
`MAILCHECK`	Количество секунд, обозначающих периодичность, с которой оболочка проверяет поступление почты в файл, задаваемый в переменной `MAIL`, или же в файлах, перечисленных в переменной `MAILPATH`. По умолчанию устанавливается значение `600`. А если установлено нулевое значение, то оболочка проверяет почту всякий раз, когда собирается выводить приглашение на ввод команд
`MAILPATH`	Список файлов, в которых проверяется поступление почты. Имя каждого файла отделяется двоеточием, после чего может быть указан знак процента (`%`) и сообщение, выводимое при поступлении почты в указанный файл (по умолчанию выводится сообщение `"You have mail"`)
`PATH`	Разделяемый двоеточием список каталогов, где оболочка ищет команду, которую требуется выполнить. Текущий каталог обозначается как `::`, `:.:` или просто `:`, если им начинается и оканчивается список каталогов
`PPID`	Идентификатор процесса, присвоенный той программе, из которой вызвана данная оболочка (т.е. идентификатор родительского процесса)
`PS1`	Основное приглашение на ввод команд (как правило, `"$ "`)
`PS2`	Вспомогательное приглашение на ввод команд (как правило, `"> "`)
`PS4`	Приглашение, употребляемое при отслеживании выполнения отлаживаемых программ, когда задается параметр `-x` оболочки или вводится команда `set -x`. По умолчанию используется приглашение `"+ "`
`PWD`	Путь к текущему рабочему каталогу

Подстановка параметров

В простейшем случае значение параметра может быть доступно по ссылке, где номер параметра предваряется знаком денежной единицы (`$`). В табл. А.3 перечислены различные виды конструкций для подстановки значений параметров. Оболочка подставляет параметры прежде подстановки имени файла и разделения командной строки на аргументы.

Наличие двоеточия после *параметра*, указанного в табл. А.3, обозначает, что этот *параметр* проверяется на наличие в нем непустого значения. Если же двоеточие отсутствует, то *параметр* проверяется только на установку в нем значения.

Таблица А.3. Конструкции для подстановки значений параметров

Конструкция	Назначение
`$`*`параметр`* или `${`*`параметр`*`}`	Подставить значение указанного *параметра*
`${`*`параметр`*`:-`*`значение`*`}`	Подставить значение указанного *параметра*, если задано непустое его значение, а иначе — заданное *значение*
`${`*`параметр`*`-`*`значение`*`}`	Подставить значение указанного *параметра*, если задано его значение, а иначе — заданное *значение*
`${`*`параметр`*`:=`*`значение`*`}`	Подставить значение указанного *параметра*, если задано непустое его значение, а иначе — подставить заданное *значение* и присвоить его указанному *параметру*
`${`*`параметр`*`=`*`значение`*`}`	Подставить значение указанного *параметра*, если это значение установлено, а иначе — подставить заданное *значение* и присвоить его указанному *параметру*
`${`*`параметр`*`:?`*`значение`*`}`	Подставить значение указанного *параметра*, если это значение установлено и не является пустым, а иначе — направить заданное *значение* в стандартный вывод ошибок и на этом завершить операцию. Если же *значение* пропущено, вывести сообщение об ошибке "*`parameter`*`: parameter null or not set `" (параметр: пустой или не установленный)
`${`*`параметр`*`:+`*`значение`*`}`	Подставить заданное *значение*, если в указанном *параметре* установлено непустое значение, а иначе — подставить пустое значение
`${`*`параметр`*`+`*`значение`*`}`	Подставить заданное *значение*, если оно установлено в указанном *параметре*, а иначе — пустое значение
`${#`*`параметр`*`}`	Подставить длину указанного *параметра*. Если в качестве *параметра* указан знак `*` или `@`, результат данной операции не определен
`${`*`параметр`*`#`*`шаблон`*`}`	Подставить значение указанного *параметра*, удалив слева наименьшую его часть, совпадающую с заданным *шаблоном*, где могут использоваться символы подстановки имен файлов (`*`, `?`, `[ . . . ]`, `!` и `@`)
`${`*`параметр`*`##`*`шаблон`*`}`	То же, что и выше, только слева удаляется наибольшая часть значения указанного *параметра*, совпадающая с заданным *шаблоном*
`${`*`параметр`*`%`*`шаблон`*`}`	То же, что и подстановка `${`*`параметр`*`#`*`шаблон`*`}`, только наименьшая часть значения указанного *параметра*, совпадающая с заданным *шаблоном*, удаляется справа
`${`*`параметр`*`%%`*`шаблон`*`}`	То же, что и выше, только справа удаляется наибольшая часть значения указанного *параметра*, совпадающая с заданным *шаблоном*

Повторный ввод команд

В оболочке ведется список предыстории недавно введенных команд. Количество команд, доступных в предыстории, определяется значением переменной `HISTSIZE` (по умолчанию оно, как правило, равно **128**), а файл, в котором хранится предыстория, — значением переменной `HISTFILE` (по умолчанию она содержит путь к файлу `$HOME/.sh_history`). А поскольку предыстория команд хранится в файле, то они становятся доступными после выхода и повторного входа в систему. Доступ к командам из предыстории можно получить одним из трех способов.

Команда `fc`

Встроенная в оболочку команда `fc` позволяет запустить редактор для правки одной или нескольких команд из предыстории. Как только отредактированная команда будет записана и завершится работа в редакторе, поправленная версия команды будет благополучно выполнена. Конкретный редактор определяется значением переменной `FCEDIT` (по умолчанию — `ed`). Для обозначения конкретного редактора вместо переменной `FCEDIT` можно также ввести команду `fc` с параметром **`-e`**.

Параметр **`-s`** позволяет выполнять команды, не вызывая сначала редактор. Простая возможность отредактировать команду встроена в команду `fc -s`. Аргумент в следующей форме:

прежняя=новая

может быть использован для замены первого вхождения символьной строки *прежняя* в символьной строке *новая* для повторного выполнения редактируемых команд.

Режим редактирования строк в редакторе `vi`

В оболочке поддерживается режим редактирования строк, совместимый с редактором `vi`. Когда этот режим включен, пользователь переносится в состояние, дублирующее режим *ввода* в редакторе `vi`. Для перехода в режим *редактирования* достаточно нажать клавишу <ESC>. В этом режиме большинство команд редактора `vi` надлежащим образом интерпретируются оболочкой. Текущая командная строка может быть отредактирована таким же образом, как и любые строки из предыстории команд. Находясь в режиме ввода или командном режиме и нажав клавишу <Enter>, можно в любой момент выполнить поправленную команду.

Все команды редактирования строк в режиме редактора `vi` перечислены в табл. А.4. При этом следует иметь в виду, что `[`*подсчет*`]` означает целочисленное значение и может быть опущено.

Таблица А.4. Команды редактирования строк в режиме редактора `vi`

Команды режима ввода

Команда	Назначение
`erase`	Удалить предыдущий символ (символ стирания; обычно знак `#` или комбинация клавиш <Ctrl+h>)
<Ctrl+w>	Удалить предыдущее слово, отделяемое пробелом
`kill`	Удалить полностью текущую строку (символ удаления строки; обычно знак `@` или комбинация клавиш <Ctrl+u>)
`eof`	Прервать выполнение оболочки, если текущая строка оказывается пустой (символ конца файла; обычно комбинация клавиш <Ctrl+d>)
<Ctrl+v>	Если сначала нажать комбинацию клавиш <Ctrl+v>, то затем можно ввести знак следующей кавычки, знаки редактирования, стирания и удаления
<Enter>	Выполнить текущую строку
<ESC>	Перейти в режим редактирования

Команды режима редактирования

Команда	Назначение
`[`***подсчет***`]k`	Получить предыдущую команду из предыстории
`[`***подсчет***`]-`	Получить предыдущую команду из предыстории
`[`***подсчет***`]j`	Получить следующую команду из предыстории
`[`***подсчет***`]+`	Получить следующую команду из предыстории
`[`***подсчет***`]G`	Получить из предыстории команду по указанному номеру ***подсчет***
`/`***строка***	Найти в предыстории самую последнюю команду, содержащую заданную ***строку***. Если указанная ***строка*** окажется пустой (т.е. ее ввод будет прерван нажатием клавиши <Enter> или комбинации клавиш <Ctrl+j>), будет использована предыдущая строка. А если указанная ***строка*** начинается со знака `^`, то поиск в предыстории начнется с указанной ***строки***
`?`***строка***	То же, что и команда `/`***строка***, только на этот раз будет произведен поиск самой первой команды
`n`	Повторить последнюю (`/`) или прежнюю (`?`) команду
`N`	Повторить последнюю (`/`) или прежнюю (`?`) команду, но изменить направление поиска на обратное
`[`***подсчет***`]l` или `[`***подсчет***`]`***пробел***	Переместить курсор на один символ вправо
`[`***подсчет***`]w`	Переместить курсор на одно буквенно-цифровое слово вправо

Продолжение табл. А.4

Команда	Назначение
[*подсчет*]W	Переместить курсор вправо к следующему слову, отделяемому пробелом
[*подсчет*]e	Переместить курсор в конец слова
[*подсчет*]E	Переместить курсор в конец текущего слова, отделяемого пробелом
[*подсчет*]h	Переместить курсор на один символ влево
[*подсчет*]b	Переместить курсор на одно слово влево
0	Переместить курсор в начало строки
^	Переместить курсор к первому непробельному символу
$	Переместить курсор в конец строки
[*подсчет*]|	Переместить курсор к столбцу под указанным номером ***подсчет*** (по умолчанию — **1**)
[*подсчет*]f*c*	Переместить курсор вправо к указанному символу ***c***
[*подсчет*]F*c*	Переместить курсор влево к указанному символу ***c***
[*подсчет*]t*c*	То же, что и команда **f*c***, только затем выполняется команда **h**
[*подсчет*]T*c*	То же, что и команда **F*c***, только затем выполняется команда **l**
;	Повторить последнюю команду **f**, **F**, **t** или **T**
,	Выполнить команду, противоположную команде **;**
a	Перейти в режим ввода и ввести текст после текущего символа
A	Присоединить текст в конце строке; то же, что и ссылка **$a**
[*подсчет*]c *движение*	Удалить символы от текущего до указанного в параметре ***движение*** и перейти в режим ввода; если параметр ***движение*** принимает значение **c**, удаляется вся строка
C	Удалить символы от текущего до конца строки и перейти в режим ввода
S	То же, что и команда **cc**
[*подсчет*]d *движение*	Удалить символы от текущего до указанного в параметре ***движение***; если параметр ***движение*** принимает значение **d**, удаляется вся строка
D	Удалить символы от текущего до конца строки; то же, что и команда **d$**
i	Перейти в режим ввода и ввести текст прежде текущего символа
I	Перейти в режим ввода и ввести текст прежде первого слова в строке
[*подсчет*]P	Разместить предыдущее изменение текста прежде курсора
[*подсчет*]p	Разместить предыдущее изменение текста после курсора
[*подсчет*]y *движение*	Скопировать символы от текущего до указанного в параметре ***движение*** непосредственно в буфер, применяемый в командах **p** и **P**; если параметр ***движение*** принимает значение **y**, копируется вся строка

Окончание табл. А.4

Команда	Назначение
Y	Скопировать символы от текущего до конца строки; то же, что и команда **y$**
R	Перейти в режим ввода и перезаписать символы в строке
[*подсчет*]r*c*	Заменим текущий символ указанным символом ***c***
[*подсчет*]x	Удалить текущий символ
[*подсчет*]X	Удалить предыдущий символ
[*подсчет*].	Повторить предыдущую команду изменения текста
~	Сменить на противоположный регистр букв текущего символа и переместить курсор вперед
[*подсчет*]_	Присоединить слово под указанным номером ***подсчет*** из предыдущей команды и перейти в режим ввода; по умолчанию это последнее слово
=	Вывести список файлов, начинающийся с текущего слова
****	Дополнить путь текущим словом; если текущее слово является каталогом, добавить знак **/**; а если текущее слово является файлом, то добавить пробел
u	Отменить последнюю команду изменения текста
U	Восстановить первоначальное состояние текущей строки
@*буква*	Программируемая функциональная клавиша. Если определен псевдоним имени ***_буква***, будет выполнена команда, определяемая его значением
[*подсчет*]v	Запустить редактор **vi** для правки строки под указанным номером ***подсчет***; если же ***подсчет*** опущен, используется текущая строка
<Ctrl+l>	Перевести строку и вывести текущую строку
L	Вывести текущую строку снова
<Ctrl+j>	Выполнить текущую строку
<Ctrl+m>	Выполнить текущую строку
<Enter>	Выполнить текущую строку
#	Вставить знак **#** в начале строки и ввести строку в предысторию команд; то же, что и последовательность **I#**<Enter>

Заключение в кавычки

В оболочке приняты четыре механизма заключения в кавычки (табл. А.5).

Таблица А.5. Сводка механизмов заключения в кавычки

Механизм	Описание
'...'	Отменяет специальное назначение всех символов, заключенных в кавычки

Окончание табл. А.5

Механизм	Описание
"..."	Отменяет специальное назначение всех символов, заключенных в кавычки, кроме знаков $, ' и \
\c	Отменяет специальное назначение указанного символа *c*; в двойных кавычках отменяется назначение указанных знаков $, ', ", \ и новой строки, которые иначе не интерпретируются; применяется для продолжения строки, если оказывается последним символом в строке (знак новой строки удаляется)
`` `команда` `` или $(команда)	Выполняет указанную *команду* и вставляет в данном месте стандартный вывод

Замена знака тильды

Каждое слово и переменная оболочки в командной строке проверяется на наличие в ее начале знака тильды (~) без кавычек. Если есть этот знак, остальная часть слова или значения переменной вплоть до знака / считается регистрационным именем пользователя, которое ищется в системном файле (как правило, в файле /etc/passwd). И если такой пользователь существует, то его начальный каталог заменяет знак ~ и регистрационное имя пользователя. А если пользователь отсутствует, то текст не изменяется. Сам же знак ~ или следующий за ним знак / заменяется значением переменной HOME.

Арифметические выражения

Общая форма конструкции для выполнения арифметических операций выглядит следующим образом:

```
$((выражение))
```

Оболочка вычисляет целочисленное арифметическое *выражение*. Такое выражение может содержать операции, константы, переменные оболочки, которые совсем не обязательно предваряются знаками денежной единицы ($). Наиболее употребительные операции перечислены ниже в порядке приоритетности.

-	Унарный минус
~	Поразрядное НЕ
!	Логическое отрицание
*, /, %	Умножение, деление, получение остатка от деления
+, -	Сложение, вычитание
<<, >>	Сдвиг влево, сдвиг вправо

<=, >=, <, >	Сравнение
==, !=	Равенство, неравенство
&	Поразрядное И
^	Поразрядное исключающее ИЛИ
|	Поразрядное ИЛИ
&&	Логическое И
||	Логическое ИЛИ
***выражение*$_1$? *выражение*$_2$: *выражение*$_3$**	Условная операция
=, *=, /=, %=, +=, <<=, >>=, &=, ^=, |=	Присваивание

Для переопределения приоритетности операций можно воспользоваться круглыми скобками. Нулевой код завершения (т.е. истинное значение) возвращается в том случае, если последнее выражение дает ненулевой результат, и единичный код завершения (т.е. ложное значение), если последнее выражение дает нулевой результат. Операции `sizeof`, ++ и – языка C могут быть доступны в конкретной реализации оболочки, хотя рассматриваемый здесь стандарт этого не требует. Чтобы выяснить их доступность в конкретной реализации, достаточно ввести операцию `sizeof` и посмотреть, что же произойдет.

Ниже приведены характерные примеры применения упомянутых выше операций в арифметических выражениях.

```
y=$((22 * 33))
z=$((y * y / (y - 1)))
```

Подстановка имен файлов

После подстановки параметров и команд в командной строке оболочка производит поиск специальных символов *****, **?** и **[**. Если они не заключены в кавычки, оболочка производит поиск текущего или другого каталога при условии, что он предваряется знаком **/**, а затем подставляет имена всех совпадающих файлов. А если ни одного совпадения не найдено, то специальные символы остаются без изменения.

Следует, однако, иметь в виду, что имена файлов, начинающиеся с точки (**.**), должны совпадать непосредственно. Иными словами, скрытые файлы отобразятся не по команде `echo *`, а по команде `echo .*`. Все символы подстановки имен файлов сведены в табл. А.6.

Таблица А.6. Символы подстановки имен файлов

Символ(ы)	Назначение
?	Совпадение с любым одиночным символом
*****	Совпадение с нулевым или большим количеством символов

Окончание табл. А.6

Символ(ы)	Назначение
[*символы*]	Совпадение с любым одиночным символом, присутствующим среди указанных *символов*; для совпадения с любым символом в пределах от C_1 до C_2 включительно можно воспользоваться формой C_1-C_2. Например с шаблоном `[A-Z]` совпадает любая прописная буква
[! *символы*]	Совпадение с любым одиночным символом, отсутствующим среди указанных *символов*; диапазон сопоставляемых с шаблоном символов может быть указан, как описано выше

Переадресация ввода-вывода

Просматривая командную строку, оболочка ищет специальные символы переадресации < и >. Если такие символы обнаружены, они обрабатываются и удаляются (вместе с любыми связанными аргументами) из командной строки. Различные виды переадресации ввода-вывода, которые поддерживаются в оболочке, перечислены в табл. А.7.

Таблица А.7. Переадресация ввода-вывода

Конструкция	Назначение
< *файл*	Переадресовать стандартный ввод из указанного *файла*
> *файл*	Переадресовать стандартный вывод в указанный *файл*; если этот *файл* не существует, он создается, а если он существует, то обнуляется
>| *файл*	Переадресовать стандартный вывод в указанный *файл*; если этот *файл* не существует, он создается, а если он существует, то обнуляется; при этом параметр `noclobber (-C)` команды `set` игнорируется
>> *файл*	То же, что и >, только результат вывода присоединяется к указанному *файлу*, если он существует
<< *слово*	Переадресация стандартного ввода из последующих строк до тех пор, пока не встретится строка, содержащая указанное *слово*. В строках выполняется подстановка параметров, выполняются команды, заключенные в кавычки, а также интерпретируется знак обратной косой черты. Если любой символ в указанном *слове* заключен в кавычки, то ни одна из перечисленных выше операций не выполняется; а если указанное *слово* предваряется знаком –, то из строк удаляются начальные знаки табуляции
<& *цифра*	Переадресация стандартного ввода из файла, связанного с указанным дескриптором *цифра*
>& *цифра*	Переадресация стандартного вывода в файл, связанный с указанным дескриптором *цифра*
<&-	Закрытие стандартного ввода
>&-	Закрытие стандартного вывода
<> *файл*	Открытие указанного *файла* как для чтения, так и для записи

Следует, однако, иметь в виду, что имя указанного *файла* не подставляется. Любой из конструкций, перечисленных в табл. А.7, может предшествовать номер дескриптора, оказывающий аналогичное воздействие на связанный с ним файл. В частности, дескриптор **0** связан со стандартным вводом, дескриптор **1** — со стандартным выводом, а дескриптор **2** — со стандартным выводом ошибок.

Экспортируемые переменные и выполнение подоболочек

Все команды, кроме встроенных в оболочку, как правило, выполняются в новом экземпляре оболочки, называемом *подоболочкой*. В подоболочках нельзя изменять значения переменных из родительской оболочки, и в них доступны только те переменные, которые (явно или неявно) *экспортированы* из родительской оболочки. Если в подоболочке изменяется значение одной из этих переменных и требуется известить об этом ее собственные подоболочки, такая переменная должна быть экспортирована перед выполнением подоболочки. А по завершении подоболочки любые установленные в ней переменные становятся недоступными для родительской оболочки.

Конструкция `( . . . )`

Если заключить одну или несколько команд в круглые скобки, эти команды будут выполнены в подоболочке.

Конструкция `{ . . . ; }`

Если заключить одну или несколько команд в фигурные скобки, эти команды будут выполнены в *текущей* оболочке. С помощью конструкций `{ . . . ; }` и `( . . . )` можно переадресовать и направить ввод-вывод по каналу в ряд команд, заключенных в скобки, а также отправить их на выполнение в фоновый режим, указав в конце знак **&**. Например, в следующей строке:

```
(prog1; prog2; prog3) 2>errors &
```

три перечисленные программы отправляются на выполнение в фоновый режим, а стандартный вывод ошибок из всех трех программ переадресовывается в файл `errors`.

Дополнительные сведения о переменных оболочки

Переменную оболочки можно поместить в среду команды, присвоив ей параметр в командной строке перед именем самой команды:

```
PHONEBOOK=$HOME/misc/phone rolo
```

Здесь переменной PHONEBOOK присваивается указанное значение, а затем она помещается в среду программы rolo. При этом среда текущей оболочки остается без изменения, как будто бы вместо предыдущей команды была выполнена следующая последовательность команд:

```
(PHONEBOOK=$HOME/misc/phone; export PHONE BOOK; rolo)
```

Функции

Функции принимают следующую форму:

```
имя () составная_команда
```

где *составная_команда* — это ряд команд, заключенных в скобки (...), {...} или блок операторов for, case, until, while. Чаще всего функции определяются в следующей форме:

```
имя () { команда; команда;     команда; }
```

где *имя* — это имя функции, определяемой в *текущей* оболочке, поскольку функции нельзя экспортировать. Определение функции можно распространить на столько строк, сколько потребуется. С помощью команды return можно прервать выполнение функции, не прерывая выполнение самой оболочки (подробнее об этом — в описании команды return). Так, в следующем примере:

```
nf () { ls | wc -l; }
```

определяется функция nf() для подсчета количества файлов в текущем каталоге.

Управление заданиями

В последующих разделах вкратце поясняется, каким образом организуется управление заданиями.

Задания в оболочке

Каждой последовательности команд, выполняемых в фоновом режиме, присваивается номер задания, начиная с первого. На задание можно ссылаться по его *идентификатору* (job_id), который состоит из знака **%** и номера задания, последовательности знаков **%+**, **%-**, **%%**, знака **%** и первых нескольких букв канала или последовательности **%?*строка***.

Идентификатор задания может быть задан в качестве аргумента следующим встроенным в оболочку командам: kill, fg, bg и wait. Специальные идентификаторы **%+** и **%-** обозначают текущие и предыдущие задания соответственно, а идентификатор **%%** — текущее задание. Текущим считается самое последнее задание, отправленное на выполнение в фоновый режим, или же задание,

выполняемое в приоритетном режиме. Идентификатор **%*строка*** обозначает задание, имя которого начинается с указанной ***строки***, а идентификатор **%?*строка*** — задание, имя которого содержит указанную ***строку***. Для вывода текущего состояния всех выполняемых заданий можно воспользоваться командой `jobs`.

Если активизирован параметр `monitor` команды `set`, оболочка выводит соответствующее сообщение по окончании каждого задания. Если попытаться выйти из оболочки при наличии в ней незавершенных заданий, будет выведено сообщение, предупреждающее об этом. Если при этом попытаться выйти из оболочки снова, выход из нее произойдет без всякого предупреждения. В интерактивных оболочках параметр `monitor` активизируется по умолчанию.

Остановка заданий

Если оболочка выполняется в системе с управлением заданиями и при этом активизирован параметр `monitor` команды `set`, задания, выполняемые в приоритетном режиме, могут быть перенесены в фоновый режим, и наоборот. Как правило, текущее задание останавливается нажатием комбинации клавиш <Ctrl+z>. По команде `bg` остановленное задание переносится в фоновый режим, а по команде `fg` — фоновое или остановленное задание в приоритетный режим

Всякий раз, когда в фоновом задании предпринимается попытка прочитать данные с терминала, оно приостанавливается до тех пор, пока не будет перенесено в приоритетный режим. Результаты, выводимые из фоновых заданий, как правило, направляются на терминал. Если выполняется команда `stty tostop`, вывод из фоновых заданий запрещается, а задание, выводящее результаты на терминал, приостанавливается до тех пор, пока не будет перенесено в приоритетный режим. При выходе из оболочки все остановленные задания уничтожаются.

Сводка команд

В этом разделе вкратце поясняются команды, встроенные в стандартную оболочку. В действительности некоторые из этих команд (например, `echo` и `test`) могут не быть встроены в оболочку или иметь упрощенную версию для встраивания в оболочку, а более сложную версию — в виде отдельной программы. Но в любом случае подобные функции должны быть предоставлены в виде утилиты в системе, совместимой со стандартом POSIX. Они встроены в оболочки Bash и Korn и применяются практически во всех сценариях оболочки.

Команда :

Общая форма: :

По существу, это *пустая* команда. Она нередко применяется с целью удовлетворить требованию наличия команды.

Пример:

```
if who | grep jack > /dev/null ; then
      :
else
      echo "jack's not logged in"
fi
```

В данном примере команда : возвращает нулевой код завершения.

Команда .

Общая форма: . *файл*

Команда-точка (.) вынуждает оболочку прочитать и выполнить указанный файл, как будто строки из этого файла были введены в данный момент. Однако указанный *файл* совсем не обязательно должен быть исполняемым, а только доступным для чтения. Для поиска указанного *файла* оболочка обращается также к переменной PATH.

Пример

```
progdefs              Выполнить команды из файла progdefs
```

В данном примере команда . вынуждает оболочку искать указанный файл progdefs в текущей переменной PATH. Если она обнаружит этот файл, то прочитает и выполнит команды из него. Тем не менее оболочки, устанавливаемые и/или изменяемые в указанном *файле*, продолжают действовать и по завершении команд из этого *файла*.

Команда alias

Общая форма: alias *имя=строка* [*имя=строка* ...]

Команда alias присваивает указанную *строку* заданному псевдониму *имя*. Всякий раз, когда используется псевдоним *имя*, оболочка подставляет сначала указанную *строку*, а затем выполняет остальные подстановки после размещения этой *строки* в командной строке.

Примеры:

```
alias ll='ls -l'
alias dir='basename $(pwd)'
```

Если псевдоним оканчивается пробелом, следующее далее слово также проверяется, не является ли оно псевдонимом. А в следующей форме данной команды:

```
alias имя
```

выводится указанный псевдоним *имя*. Если же команда alias указана без аргументов, она выводит все псевдонимы. Команда alias возвращает нулевой код завершения, если только не задано *имя*, для которого определен псевдоним, как, например, в команде alias *имя*.

Команда bg

Общая форма: bg *job_id*

Если активизировано управление заданиями, то задание, обозначаемое идентификатором *job_id*, переносится в фоновый режим. Если же команда bg введена без аргументов, то в фоновый режим переносится самое последнее приостановленное задание.

Пример:

```
bg %2
```

Команда break

Общая форма: break

Выполнение данной команды приводит к немедленному завершению наиболее глубоко вложенного цикла for, while или until. Выполнение будет продолжено с команд, сразу же следующих после данного цикла. Если же команда break используется в следующей форме:

```
break n
```

то автоматически завершаются *n* наиболее глубоко вложенных циклов.

Команда case

Общая форма:

```
case значение in
      шаблон1) команда
               команда
               ...
               команда;;
      шаблон2) команда
               команда
               ...
               команда;;
               ...
      шаблонn) команда
               команда
               ...
               команда;;
esac
```

Указанное слово *значение* последовательно сравнивается с *шаблоном*$_1$, *шаблоном*$_2$ и *шаблоном*$_n$ до тех пор, пока не будет обнаружено совпадение. А далее команды, следующие сразу же после совпавшего шаблона, выполняются до тех пор, пока не встретятся две следующие подряд точки с запятой (;;). В этот момент выполнение команды, реализующий оператор case, завершается.

Если же указанное *значение* не совпадает ни с одним из заданных шаблонов, то ни одна из команд, расположенных в ветвях выбора из оператора `case`, не выполняется. Шаблон *****, обозначающий *любое* совпадение, нередко употребляется в последней ветви оператора `case` в качестве "универсального" варианта выбора по умолчанию.

В шаблонах могут быть использованы следующие метасимволы:

- ***** (совпадение с нулевым или бо́льшим количеством символов);
- **?** (совпадение с любым одиночным символом);
- **[...]** (совпадение с любым одиночным символом, заключенным в квадратные скобки).

А знак **|** может служить для обозначения логического объединения двух шаблонов по ИЛИ. Так, следующее выражение:

шаблон$_1$**|** *шаблон*$_2$

означает совпадение с заданным *шаблоном*$_1$ или *шаблоном*$_2$.

Примеры:

```
case $1 in
      -l) lopt=TRUE;;
      -w) wopt=TRUE;;
      -c) copt=TRUE;;
      *) echo "Unknown option";;
esac
case $choice in
      [1-9]) valid=TRUE;;
          *) echo "Please choose a number from 1-9";;
esac
```

Команда cd

Общая форма: `cd` *каталог*

Выполнение этой команды приводит к тому, что оболочка делает указанный *каталог* текущим. Если же *каталог* опущен, оболочка сделает текущим каталог, указанный в переменной `HOME`.

Если значение переменной оболочки `CDPATH` окажется пустым, в качестве *каталога* следует указать полный путь к каталогу (например, `/users/steve/documents`) или путь относительно текущего каталога (например, `documents` или `../pat`). Если же значение переменной оболочки `CDPATH` окажется непустым, а в качестве *каталога* указан полный путь, оболочка произведет поиск каталога, содержащего заданный *каталог*, в списке разделяемых двоеточием каталогов, находящемся в переменной `CDPATH`.

Примеры:

```
$ cd documents/memos      Перейти в каталог documents/memos
$ cd                      Перейти в начальный каталог
```

Если в команде cd указан аргумент -, оболочка переместит пользователя обратно в предыдущий каталог. В итоге выводится путь к новому текущему каталогу.

Примеры:

```
$ pwd
/usr/lib/uucp
$ cd /
$ cd -
/usr/lib/uucp
$
```

Команда cd устанавливает в переменной PWD новый текущий каталог, а в переменной OLDPWD — предыдущий.

Команда continue

Общая форма: continue

Выполнение команды continue в цикле for, while или until приводит к пропуску любых команд, следующих после этой команды. А выполнение цикла продолжается далее, как обычно. Если же данная команда применяется в следующей форме:

```
continue n
```

пропускаются команды, находящиеся в *n* наиболее глубоко вложенных циклах. А выполнение цикла продолжается далее, как обычно.

Команда echo

Общая форма: echo *аргументы*

Выполнение этой команды приводит к тому, что указанные *аргументы* направляются в стандартный вывод. Каждое слово, которое содержат указанные *аргументы*, разделяется пробелом. Знак новой строки выводится последним. Если же *аргументы* опущены, то происходит обычный пропуск строки. Некоторые символы, экранированные знаками обратной косой черты, имеют специальное назначение в команде echo (табл. А.8).

Таблица А.8. Управляющие символы, применяемые в команде echo

Символ	Назначение
\a	Предупреждение
\b	Возврат на одну позицию
\c	Строка без завершающего знака новой строки
\f	Перевод страницы
\n	Новая строка
\r	Перевод каретки
\t	Горизонтальная табуляция

Окончание табл. А.8

Символ	Назначение
\v	Вертикальная табуляция
****	Обратная косая черта
\0*ппп*	Символ со значением ***ппп*** в коде ASCII, где ***ппп*** — восьмеричное число, состоящее из одной–трех цифр и начинающееся с нуля

Перечисленные выше символы следует заключить в кавычки, чтобы они были правильно интерпретированы командой `echo`, а не оболочкой.

Примеры:

```
$ echo *                          Перечислить все файлы
                                  в текущем каталоге
bin docs mail mise src
$ echo                            Пропустить строку

$ echo 'X\tY'                     Вывести буквы X и Y,
                                  разделенные знаком табуляции
X Y
$ echo "\n\nSales Report"          Пропустить две строки, прежде чем
                                  отображать слова Sales Report


Sales Report
$ echo "Wake up!!\a"              Вывести сообщение и
                                  предупреждающий сигнал на терминал
Wake up!!
$
```

Команда `eval`

Общая форма: `eval` *аргументы*

Выполнение этой команды приводит к тому, что оболочка сначала вычисляет указанные *аргументы*, а затем выполняет полученные результаты. Это удобно в том случае, если требуется, чтобы оболочка дважды просматривала командную строку.

Пример:

```
$ x='abc def'
$ y='$x'              Присвоить переменной y значение переменной $x
$ echo $y
$x
$ eval echo $y
abc def
$
```

Команда exec

Общая форма: exec *команда аргументы*

Когда оболочка выполняет команду exec, она инициирует выполнение указанной *команды* с заданными *аргументами*. В отличие от других команд, указанная *команда* заменяет текущий процесс, а это означает, что новый процесс не создается. Как только указанная *команда* начнет выполняться, вернуться в программу, инициировавшую команду exec, уже нельзя. Если же указана переадресация ввода-вывода, то соответственно переадресовывается и ввод-вывод для оболочки.

Примеры:

```
exec /bin/sh        Заменить текущий процесс оболочкой sh
exec < datafile     Переназначить стандартный ввод из файла datafile
```

Команда exit

Общая форма: exit *n*

Выполнение команды exit приводит к тому, что текущая программа оболочки немедленно прерывается. Код завершения программы принимает целочисленное значение аргумента *n*, если он предоставлен. Если же аргумент *n не* предоставлен, то возвращается код завершения последней команды, выполнявшейся прежде команды exit.

Нулевой код завершения служит для обозначения удачного исхода, а ненулевой код завершения — неудачного исхода (например, ошибочного условия). Этот код используется оболочкой для вычисления условий в операторах if, while и until, а также в операциях **&&** и **||**.

Примеры:

```
who | grep $user > /dev/null
exit              Выйти с кодом завершения последней команды grep
exit 1            Выйти с кодом завершения 1
if finduser       Если команда finduser возвращает нулевой
                  код завершения, то...
then

fi
```

Однако следует иметь в виду, что выполнение команды exit в исходной оболочке приведет к выходу из системы.

Команда export

Общая форма: export *переменные*

Команда export сообщает оболочке, что указанные в ней *переменные* должны быть помечены как экспортируемые. Это означает, что их значения должны быть переданы далее подоболочкам.

Примеры:

```
export PATH PS1
export dbhome x1 y1 date
```

При экспорте переменные могут быть установлены в следующей форме:

```
export переменная=значение...
```

Таким образом, следующие строки:

```
PATH=$PATH:$HOME/bin; export PATH
CDPATH=.:$HOME:/usr/spool/uucppublic; export CDPATH
```

могут быть переписаны таким образом:

```
export PATH=$PATH:$HOME/bin CDPATH=.:$HOME:/usr/spool/uucppublic
```

В результате выполнения команды `export` с аргументом **`-p`** выводится список экспортируемых переменных и их значений в следующей форме:

```
export переменная=значение
```

или в такой форме:

```
export переменная
```

если указанная *переменная* экспортирована, но ее значение еще не установлено.

Команда `false`

Общая форма: `false`

Команда `false` возвращает ненулевой код завершения.

Команда `fc`

Общая форма: `fc -e` *редактор* `-lnr` *первая последняя*
`fc -s` *прежняя=новая первая*

Команда `fc` служит для редактирования команд, находящихся в предыстории. Команды указываются в пределах от *первая* до *последняя*, где *первая* и *последняя* команды могут быть обозначены номерами или символьными строками. Отрицательный номер интерпретируется как смещение от номера текущей команды, тогда как символьная строка обозначает самую последнюю введенную команду, начинающуюся с данной строки. Команды вводятся в указанный *редактор* и выполняются после выхода из него. Если же *редактор* не указан, то используется значение из переменной оболочки `FCEDIT`. А если переменная `FCEDIT` не установлена, используется редактор `ed`.

Параметр **-l** команды fc обозначает простое перечисление команд от *первой* до *последней*, когда редактор не вызывается. Если же указан и параметр **-n**, эти команды не предваряются номерами. А параметр **-r** позволяет вывести команды в обратном порядке.

Если аргумент *прежняя* не указан, то по умолчанию принимается значение аргумента *первая*. А если не указан аргумент *первая*, то по умолчанию для редактирования выбирается предыдущая команда, а также значение **-16** для перечисления списка команд. Параметр **-s** обусловливает выполнение выбранной команды без предварительного ее редактирования. Следующая форма команды fc:

```
fc -s прежняя=новая первая
```

вызывает повторное выполнение команды *первая* после замены в ней символьной строки *прежняя* на строку *новая*. Если же команда *первая* не указана, то используется предыдущая команда. А если не указано выражение *прежняя=новая*, то команда не изменяется.

Примеры:

```
fc -l                 Перечислить 16 последних команд
fc -e vi sed          Прочитать команду редактора sed в редактор vi
fc 100 110            Прочитать команды 100-110 в переменную $FCEDIT
fc -s                 Выполнить повторно предыдущую команду
fc -s abc=def 104     Выполнить повторно команду 104,
                      заменив abc на def
```

Команда fg

Общая форма: fg *job_id*

Если активизировано управление заданиями, то задание с указанным идентификатором *job_id* переносится в приоритетный режим. Если же аргументы в данной команде отсутствуют, то задание, приостановленное или отправленное в фоновый режим самым последним, переносится в приоритетный режим.

Пример:

```
fg %2
```

Команда for

Общая форма:

```
for переменная in слово1 слово2... словоn
do
      команда
      команда

done
```

Выполнение этой команды приводит к тому, что команды, указанные между операторами do и done, будут выполнены столько раз, сколько слов перечислено после элемента in. На первом шаге цикла for указанной *переменной* этого цикла присваивается первое слово (т.е. $слово_1$) и далее выполняются команды, указанные между операторами do и done. На втором шаге данного цикла указанной *переменной* присваивается второе слово ($слово_2$) и те же самые команды выполняются снова.

Этот процесс повторяется до тех пор, пока указанной *переменной* цикла не будет присвоено последнее слово (т.е. $слово_n$) и выполнены команды, указанные между операторами do и done. В этот момент цикл завершается и выполнение продолжается с команды, следующей сразу же после оператора done.

В следующей специальной форме команды for:

```
for переменная
do

done
```

предполагается употребление списка позиционных параметров $1, $2 и т.д., что равнозначно такой форме:

```
for переменная in "$@"
do

done
```

Пример:

```
# Обработать все файлы в текущем каталоге по команде nroff
for file in *
do
    nroff -Tlp $file | lp
done
```

Команда getopts

Общая форма: getopts *параметры переменная*

Эта команда обрабатывает аргументы командной строки. Здесь *параметры* — это список достоверных однобуквенных параметров. Если после любой буквы *параметры* содержат двоеточие (:), такому параметру требуется дополнительный аргумент, отделяемый от параметра хотя бы одним пробелом в командной строке.

Всякий раз, когда команда getopts вызывается, она обрабатывает следующий аргумент в командной строке. Если обнаруживается достоверный параметр, команда getopts сохраняет соответствующую букву параметра в указанной *переменной* и возвращает нулевой код завершения. Если же указан недостоверный

параметр (или, наоборот, параметр, отсутствующий в списке параметров), команда `getopts` сохраняет знак **?** в указанной *переменной* и возвращает нулевой код завершения. Она направляет также сообщение об ошибке в стандартный вывод ошибок.

Если параметр принимает аргумент, команда `getopts` сохраняет соответствующую букву параметра в указанной *переменной*, тогда как аргумент командной строки — в специальной переменной `OPTARG`. Если же аргументы оставляются в командной строке, команда `getopts` устанавливает знак **?** в указанной *переменной* и направляет сообщение об ошибке в стандартный вывод ошибок. А если в командной строке больше не остается параметров (т.е. следующий аргумент в командной строке не начинается со знака **-**), то команда `getopts` возвращает ненулевой код завершения.

В команде `getopts` применяется также специальная переменная `OPTIND`. Первоначально в ней устанавливается значение **1**, а затем оно корректируется всякий раз, когда происходит возврат из команды `getopts`, указывая номер следующего аргумента, который следует обработать в командной строке. Чтобы обозначить конец списка аргументов, в командной строке может быть указан аргумент **--**.

В команде `getopts` поддерживаются составные аргументы, как показано ниже.

```
repx -iau
```

что равнозначно следующему:

```
repx -i -a -u
```

Параметры, которым требуются аргументы, могут и не быть составными. Если используется следующая форма команды `getopts`:

```
getopts параметры переменная аргументы
```

данная команда выполняет синтаксический анализ указанных *аргументов*, а *не* аргументов командной строки.

Пример:

```
usage="Usage: foo [-r] [-O outfile] infile"
while getopts ro: opt

do
      case "$opt"
      in
            r) rflag=1;;
            O) oflag=1
               ofile=$OPTARG;;
          \?) echo "$usage"
```

```
                exit 1;;
      esac
done

if [ $OPTIND -gt $# ]
then
      echo "Needs input file!"
      echo "$usage"
      exit 2
fi

shift $((OPTIND - 1))
ifile=$1
```

Команда `hash`

Общая форма: `hash` *команды*

Эта команда предписывает оболочке найти указанные *команды* и запомнить каталоги, в которых они находятся. Если *команды* не указаны, то выводится список хешированных команд. Если же данная команда используется в следующей форме:

```
hash -r
```

то оболочка удаляет все команды из своего хеш-списка. А при последующем выполнении любой другой команды оболочка применяет обычные методы поиска команд.

Примеры:

```
hash rolo whoq          Добавить команды rolo и whoq в хеш-список
hash                    Вывести хеш-список
hash -r                 Удалить хеш-список
```

Команда `if`

Общая форма:

```
if команда_t
then
      команда
      команда

fi
```

Сначала выполняется указанная $команда_t$, а затем проверяется код ее завершения. Если этот код оказывается нулевым, выполняются команды, находящиеся между операторами `then` и `fi`. В противном случае эти команды пропускаются.

Пример:

```
if grep $sys sysnames > /dev/null
then
      echo "$sys is a valid system name"
fi
```

Если команда `grep` возвращает в данном примере нулевой код завершения, а это происходит в том случае, если она обнаружит переменную `$sys` в файле `sysnames`, то выполняется команда `echo`. В противном случае эта команда пропускается. Нередко после условного оператора `if` в описываемой здесь команде для непосредственного вызова указывается встроенная команда `test`, обозначаемая явно или неявно в сокращенной форме **`[]`**.

Пример

```
if [ $# -eq 0 ] ; then
      echo "Usage: $0 [-l] file ..."
      exit 1
fi
```

Условный оператор `if` может быть дополнен оператором `else`, если рассматриваемая здесь команда возвращает ненулевой код завершения. В этом случае ее общая форма выглядит следующим образом:

```
if команда_t
then
      команда
      команда

else
      команда
      команда

fi
```

Если указанная *команда*$_t$ возвращает нулевой код завершения, то выполняются команды, находящиеся между операторами `then` и `else`, а команды, следующие после оператора `else`, пропускаются. Если же указанная *команда*$_t$ возвращает ненулевой код завершения, то команды, находящиеся между операторами `then` и `else`, пропускаются, а выполняются команды, находящиеся между операторами `else` и `fi`.

Пример:

```
if [ -z "$line" ]
then
      echo "I couldn't find $name"
else
      echo "$line"
fi
```

Если переменная `line` в приведенном выше примере содержит нулевую длину, то выполняется команда `echo`, выводящая сообщение `"I couldn't find $name"` (Не удалось найти имя, указанное в переменной `$name`). А приведенная ниже завершающая форма команды `if` удобна в том случае, если требуется сделать двойной выбор.

Пример:

```
if команда1
then
      команда
      команда
      ...
elseif команда2
then
      команда
      команда
      ...
elseif командаn
then
      команда
      команда

else
      команда
      команда

fi
```

Указанная последовательность *команда*$_1$, *команда*$_2$, ... *команда*$_n$ вычисляется по порядку до тех пор, пока одна из этих команд не возвратит нулевой код завершения. И в этот момент выполняются команды, следующие сразу же после оператора `then` и вплоть до очередного оператора `elif`, `else` или `fi`. Если же ни одна из указанных команд не возвратит нулевой код завершения, то выполняются команды, следующие сразу же после оператора `else`, при условии, что они заданы.

Пример:

```
if [ "$choice" = a ] ; then
      add $*
elif [ "$choice" = d ] ; then
      delete $*
elif [ "$choice" = l ] ; then
      list
else
      echo "Bad choice!"
      error=TRUE
fi
```

Команда jobs

Общая форма: jobs

По этой команде выводится список активных заданий. Если в этой команде указан параметр **-l**, то каждое задание перечисляется вместе с присвоенным ему идентификатором процесса. Если же указан параметр **-p**, то перечисляются только идентификаторы процесса, присвоенные активным заданиям. А если в команде jobs указан дополнительный идентификатор задания *job_id*, то перечисляются только сведения об этом задании.

Пример:

```
$ sleep 100 &
[1] 1104
$ jobs
[1] +         Running sleep 100 &
$
```

Команда kill

Общая форма: kill -*сигнал задание*

Команда kill посылает сигнал указанному процессу на его прерывание, где *задание* — идентификатор процесса или идентификатор задания *job_id*, а *сигнал* — номер или одно из наименований сигналов, определенных в заголовочном файле <signal.h> (подробнее об этом см. далее описание команды trap). Имена сигналов перечисляются по команде kill -l. Если вместе с параметром **-l** указан номер сигнала, то выводится соответствующее имя сигнала. А если вместе с параметром **-l** указан идентификатор процесса, то выводится наименование сигнала, прервавшего данный процесс, если он действительно был прерван этим сигналом.

Вместе с наименованием сигнала может быть также указан параметр **-s**. В этом случае наименование сигнала не предваряется знаком дефиса, как демонстрируется в одном из приведенных ниже примеров. Если же конкретный *сигнал* не указан, то используется наименование сигнала SIGTERM (или просто TERM). Следует также иметь в виду, что в одной строке с командой kill может быть указано несколько идентификаторов процессов.

Примеры:

```
kill -9 1234
kill -HUP %2 3456
kill -s TERM %2
kill %1
```

Команда newgrp

Общая форма: newgrp *группа*

Эта команда заменяет настоящий идентификатор группы (GID) указанной *группой*. Если она указана без аргументов, то происходит возврат к группе, используемой по умолчанию.

Пример:

```
newgrp shbook                  Заменить группу на shbook
newgrp                         Вернуться обратно к используемой
                               по умолчанию группе
```

Если с новой группой связан пароль, а текущий пользователь не перечислен в качестве члена этой группы, ему будет предложено ввести свой пароль. По команде newgrp -l происходит возврат к исходной группе, выбираемой при регистрации в системе.

Команда pwd

Общая форма: pwd

Эта команда предписывает оболочке вывести рабочий каталог текущего пользователя, направив его в стандартный вывод.

Примеры:

```
$ pwd
/users/steve/documents/memos
$ cd
$ pwd
/users/steve
$
```

Команда read

Общая форма: read *переменные*

Выполнение этой команды приводит к тому, что оболочка читает строку из стандартного ввода и присваивает указанной *переменной* слова, разделяемые пробелами далее в строке. Если в строке указано меньше переменных, чем слов, дополнительные слова сохраняются в последней переменной.

Указание лишь одной переменной означает, что этой переменной присваивается прочитанная строка. Команда read возвращает нулевой код завершения, если только не обнаруживается условие, обозначающее конец файла.

Пример:

```
$ read hours mins
10 19
$ echo "$hours:$mins"
10:19
```

```
$ read num rest
39 East 12th Street, New York City 10003
$ echo "$num\n$rest"
39
East 12th Street, New York City 10003
$ read line
     Here    is an entire        line \r
$ echo "$line"
Here     is an entire        line r
$
```

В последнем примере обратите внимание на то, что любые начальные пробельные символы будут "поглощены" оболочкой при чтении строки. Если это вызывает трудности, можно соответственно изменить значение переменной IFS.

Следует также иметь в виду, что знаки обратной косой черты интерпретируются оболочкой при чтении строки и любой подобный знак (например, двойная косая черта как одиночная) интерпретируется командой echo, если отображается значение переменной. Если же в команде read указан параметр **-r**, то знак **** не должен интерпретироваться в конце строки как ее продолжение.

Команда readonly

Общая форма: readonly *переменные*

Эта команда сообщает оболочке, что перечисленным в ней *переменным* нельзя присваивать значения. Значения этим переменным могут быть дополнительно присвоены только в командной строке readonly. Если же впоследствии попытаться присвоить значение переменной, указанной в командной строке readonly, оболочка выдаст соответствующее сообщение об ошибке.

Переменные удобно устанавливать в командной строке readonly для того, чтобы исключить неумышленную перезапись их значений. Этим также гарантируется, что другие пользователи программы оболочки не смогут изменить значения отдельных переменных (например, переменных HOME или PATH). Атрибут readonly (только для чтения) не передается далее подоболочкам. Если команда readonly указана с параметром **-p**, она выводит список переменных с атрибутом readonly.

Примеры:

```
$ readonly DB=/users/steve/database          Присвоить значение
                                             переменной DB и сделать
                                             ее доступной только
                                             для чтения
$ DB=foo                                     Попытаться присвоить
                                             значение переменной DB
sh: DB: is read-only                         Сообщение об ошибке,
                                             выводимое оболочкой
$ echo $DB                                   Тем не менее ее значение
/users/steve/database                        по-прежнему доступно
$
```

Команда `return`

Общая форма: `return` *n*

Выполнение этой команды приводит к тому, что оболочка прерывает выполнение текущей функции и сразу же возвращает код завершения *n* вызывающей части программы. Если же аргумент *n* опущен, то возвращается код завершения команды, выполнявшейся непосредственно перед командой `return`.

Команда `set`

Общая форма: `set` *параметры аргументы*

Эта команда служит как для установки и сброса указанных *параметров*, так и для установки позиционных параметров, обозначаемых заданными *аргументами*. Каждый из указанных однобуквенных *параметров* устанавливается, если он предваряется знаком **-**, или сбрасывается, если он предваряется знаком **+**. Параметры могут быть сгруппированы, как демонстрируется в приведенном ниже примере, где устанавливаются параметры `f` и `x`.

```
set -fx
```

В табл. А.9 сведены параметры, которые могут быть выбраны для установки или сброса в команде `set`.

Таблица А.9. Параметры, доступные для установки или сброса в команде `set`

Параметр	Назначение
--	Не интерпретировать последующие ***аргументы***, которые предваряются знаком **-**, как параметры. В отсутствие аргументов позиционные параметры не устанавливаются
-a	Автоматически экспортировать все переменные, которые последовательно определяются или видоизменяются
-b	Если этот параметр поддерживается в конкретной реализации, оболочка уведомляет пользователя о завершении фоновых заданий
-C	Не разрешать переадресацию вывода для перезаписи существующих файлов. Чтобы обеспечить перезапись отдельных файлов, можно указать последовательность символов **>|**, даже если данный параметр установлен
-e	Завершить выполнение при неудачном исходе выполнения любой команды или возврате ненулевого кода завершения
-f	Запретить генерирование имени файла
-h	Ввести команды из функций в хеш-список по мере их определения, но не делать этого по мере их выполнения
-m	Включить монитор заданий
-n	Прочитать команды, не выполняя их (это удобно для проверки уравновешенности блоков операторов **do** ... **done** и **if** ... **fi**)
+o	Записать текущие настройки режима установки параметров в формате команды

Окончание табл. A.9

Параметр	Назначение
-o *m*	Включить режим установки параметров ***m*** (см. табл. A.10)
-u	Выдать ошибку, если происходит обращение к переменной, которой еще не было присвоено значение, или же обращение к неустановленному еще позиционному параметру
-v	Вывести каждую командную строку по мере ее чтения в оболочке
-x	Вывести каждую команду и ее аргументы по мере их выполнения, предварив ее знаком +

Режимы оболочки включаются или выключаются с помощью обозначения **-o** или **+o**, после которого следует имя параметра. Эти параметры сведены в табл. A.10.

Таблица A.10. Режимы оболочки

Режим	Назначение
allexport	То же, что и параметр **-a**
errexit	То же, что и параметр **-e**
ignoreeof	Для выхода из оболочки следует воспользоваться командой **exit**
monitor	То же, что и параметр **-m**
noclobber	То же, что и параметр **-C**
noexec	То же, что и параметр **-n**
noglob	То же, что и параметр **-f**
nolog	Не размещать определения функций в предыстории
nounset	То же, что и параметр **-u**
verbose	То же, что и параметр **-v**
vi	Установить **vi** в качестве строкового редактора
xtrace	То же, что и параметр **-x**

По команде `set -o` без имени какого-нибудь из перечисленных выше режимов оболочки выводится список всех подобных режимов и их текущие установки. Переменная оболочки `$-` содержит установки текущих параметров.

В каждом слове, перечисленном в указанных *аргументах*, задаются соответственно позиционные параметры `$1`, `$2` и т.д. Если первое слово может начинаться со знака "минус", то надежнее указать в команде `set` параметр **--**, чтобы исключить интерпретацию его значения. Если же *аргументы* указаны, то в переменной `$#` будет установлено количество параметров, присвоенных после выполнения данной команды.

Примеры:

`set -vx`	*Вывести все командные строки по мере их чтения, а также каждую команду и ее аргументы по мере их выполнения*
`set "$name" "$address" "$phone"`	*Установить параметр* ***$1*** *в переменной* ***$name****, параметр* ***$2*** *– в переменной* ***$address****, а параметр* ***$3*** *– в переменной* ***$phone***
`set -- -1`	*Установить в параметре* ***$1*** *значение* ***-1***
`set -o vi`	*Установить режим редактирования строк в редакторе* ***vi***
`set +o verbose -o noglob`	*Выключить многословный режим, включить режим* ***noglob***

Команда `shift`

Общая форма: `shift`

Выполнение этой команды приводит к смещению позиционных параметров `$1`, `$2`, ..., `$`*n* на одну позицию влево. Это означает, что на позиции параметра `$1` оказывается параметр `$2`, на позиции параметра `$2` — параметр `$3`, а на позиции параметра `$`*n-1* — параметр `$`*n*. Соответственно корректируется и значение переменной `$#`.

Если же используется следующая форма данной команды:

```
shift n
```

то смещение происходит на *n* позиций влево.

Примеры:

```
$ set a b c d
$ echo "$#\n$*"
4
a b c d
$ shift
$ echo "$#\n$*"
3
b c d
$ shift 2
$ echo "$#\n$*"
1
d
$
```

Команда `test`

Общая форма:

`test` *условие*

или

`[` *условие* `]`

Оболочка вычисляет заданное *условие* и возвращает нулевой код завершения, если в результате будет получено логическое значение TRUE. Если же в результате будет получено логическое значение FALSE, то возвращается ненулевой код завершения. А если применяется форма `[` *условие* `]`, то после открывающей скобки (**[**) и перед закрывающей скобкой (**]**) указывается пробел.

Заданное *условие* состоит из одной или нескольких операций, приведенных в табл. А.11. Операция **-a** имеет больший приоритет, чем операция **-o**. Но в любом случае для группирования подвыражений употребляются круглые скобки. Не следует только забывать, что круглые скобки принимаются во внимание оболочкой, поэтому они должны быть заключены в кавычки. Операции и операнды, включая и круглые скобки, должны разделяться одним или несколькими пробелами, поэтому команда `test` интерпретирует их как отдельные аргументы. Чаще всего команда `test` применяется для проверки условий в операторах, реализуемых командами `if`, `while` или `until`.

Примеры:

```
# выяснить, является ли файл perms исполняемым

if test -x /etc/perms
then

fi

# выяснить, является ли проверяемый элемент каталогом
# или обычным читаемым файлом

if [ -d $file -o \( -f $file -a -r $file \) ]
then

fi
```

Таблицы А.11. Операции, используемые в команде `test`

Операция	Возвращает логическое значение *TRUE*, т.е. нулевой код завершения при условии, что...
Файловые операции	
`-b` ***файл***	Указанный ***файл*** является специальным файлом блочного устройства
`-c` ***файл***	Указанный ***файл*** является специальным файлом символьного устройства
`-d` ***файл***	Указанный ***файл*** является каталогом
`-e` ***файл***	Указанный ***файл*** существует
`-f` ***файл***	Указанный ***файл*** является обычным файлом
`-g` ***файл***	В указанном ***файле*** установлен бит смены идентификатора группы (**`SGID`**)
`-h` ***файл***	Указанный ***файл*** является символической ссылкой
`-k` ***файл***	В указанном ***файле*** установлен бит закрепления
`-L` ***файл***	Указанный ***файл*** является символической ссылкой
`-p` ***файл***	Указанный ***файл*** является именованным каналом
`-r` ***файл***	Указанный ***файл*** доступен для чтения в текущем процессе
`-S` ***файл***	Указанный ***файл*** является сокетом
`-s` ***файл***	Указанный ***файл*** имеет ненулевую длину
`-t` ***fd***	Указанный параметр ***fd*** является дескриптором открытого файла, связанным с терминалом (по умолчанию он равен **`1`**)
`-u` ***файл***	В указанном ***файле*** установлен бит смены идентификатора пользователя (**`SUID`**)
`-w` ***файл***	Указанный ***файл*** доступен для записи в текущем процессе
`-x` ***файл***	Указанный ***файл*** является исполняемым
Строковые операции	
строка	Указанная ***строка*** не является пустой
`-n` ***строка***	Указанная ***строка*** *не* является пустой и должна быть проверена в команде **`test`**
`-z` ***строка***	Указанная ***строка*** является пустой и должна быть проверена в команде **`test`**
строка$_1$ **`=`** ***строка***$_2$	Указанная ***строка***$_1$ равна заданной ***строке***$_2$
строка$_1$ **`!=`** ***строка***$_2$	Указанная ***строка***$_1$ *не* равна заданной ***строке***$_2$
Целочисленные операции сравнения	
int$_1$ **`-eq`** ***int***$_2$	Указанные целочисленные значения ***int***$_1$ и ***int***$_2$ равны
int$_1$ **`-ge`** ***int***$_2$	Указанное целочисленное значение ***int***$_1$ больше или равно заданному целочисленному значению ***int***$_2$

Окончание табл. A.11

Операция	Возвращает логическое значение ***TRUE***, т.е. нулевой код завершения при условии, что...
int$_1$ **-gt** ***int***$_2$	Указанное целочисленное значение ***int***$_1$ больше заданного целочисленного значения ***int***$_2$
int$_1$ **-le** ***int***$_2$	Указанное целочисленное значение ***int***$_1$ меньше или равно заданному целочисленному значению ***int***$_2$
int$_1$ **-lt** ***int***$_2$	Указанное целочисленное значение ***int***$_1$ меньше заданного целочисленного значения ***int***$_2$
int$_1$ **-ne** ***int***$_2$	Указанные целочисленные значения ***int***$_1$ и ***int***$_2$ *не* равны
Логические операции	
! ***выражение***	Заданное ***выражение*** оказывается ложным (**FALSE**), а иначе — истинным (**TRUE**)
выражение$_1$ **-a** ***выражение***$_2$	Истинным (**TRUE**) оказывается как ***выражение***$_1$ так и ***выражение***$_2$
выражение$_1$ **-o** ***выражение***$_2$	Истинным (**TRUE**) оказывается ***выражение***$_1$ или ***выражение***$_2$

Команда `times`

Общая форма: `times`

Выполнение этой команды приводит к тому, что оболочка направляет в стандартный вывод общее количество времени, использованного оболочкой и всеми порожденными ею процессами. Для каждого из этих процессов выводятся два числа: первое обозначает накопленное пользователем время, а второе — время, накопленное системой. Однако команда `times` не сообщает время, использованное встроенными командами.

Пример:

```
$ times                          Вывести время, использованное процессами
1m5s 2m9s                        1 минута, 5 секунд пользовательского
                                 времени, 2 минуты, 9 секунд
                                 системного времени
8m22.23s 6m22.01s                Время, использованное
                                 порожденными процессами
$
```

Команда `trap`

Общая форма: `trap` *команды сигналы*

Эта команда предписывает оболочке выполнить указанные *команды* всякий раз, когда она получает один из перечисленных *сигналов*. Эти сигналы могут быть указаны по имени или номеру.

Если команда `trap` введена без аргументов, она выводит список назначенных обработчиков прерываний. Если же в качестве первого аргумента указана пустая строка, как следующем примере:

```
trap "" сигналы
```

то перечисленные *сигналы* игнорируются оболочкой при их получении.

А если команда `trap` используется в следующей форме:

```
trap сигналы
```

то обработка каждого из перечисленных *сигналов* сводится к стандартному действию, устанавливаемому по умолчанию.

Примеры:

```
trap "echo hangup >> $ERRFILE; exit"  HUP     При зависании вывести
                                               сообщение в журнал
                                               регистрации и выйти
trap "rm $TMPFILE; exit" 1 2 15                Удалить файл, указанный
                                               в переменной $TMPFILE,
                                               по сигналу 1, 2 или 15
trap "" 2                                      Игнорировать прерывания
trap 2                                         Восстановить стандартную
                                               обработку прерываний
```

В табл. А.12 перечислены значения, которые могут быть указаны в списке сигналов, обрабатываемых по команде `trap`.

Таблица А.12. Наименования и номера сигналов, перечисляемых в команде `trap`

Номер сигнала	Наименование сигнала	Когда генерируется
0	**EXIT**	При выходе из оболочки
1	**HUP**	При зависании
2	**INT**	При прерывании (например, нажатием клавиши <Delete> или комбинации клавиш <Ctrl+C>)
3	**QUIT**	При выходе из программы
6	**ABRT**	При аварийном или преждевременном завершении программы
9	**KILL**	При уничтожении процесса
14	**ALRM**	По истечении аварийной выдержки времени
15	**TERM**	По сигналу прерывания программы, посылаемому командой **kill** по умолчанию

Оболочка просматривает указанные *команды*, когда обнаруживает команду `trap`, и делает это снова при получении одного из перечисленных сигналов. Это, например, означает, что при обнаружении следующей команды:

```
trap "echo $count lines processed >> $LOGFILE; exit" HUP INT TERM
```

оболочка подставляет в данный момент значение переменной `count`, но *не* делает этого при получении одного из перечисленных сигналов. Получить значение переменной `count`, подставляемое при поступлении одного из перечисленных сигналов, все-таки можно, если заключить указанные команды в одиночные кавычки, как показано ниже.

```
trap 'echo $count lines processed >> $LOGFILE; exit' HUP INT TERM
```

Команда `true`

Общая форма: `true`

Эта команда возвращает нулевой код завершения.

Команда `type`

Общая форма: `type` *команды*

Эта команда выводит сведения об указанных *командах*.

Примеры:

```
$ type troff echo
troff is /usr/bin/troff
echo is a shell builtin
$
```

Команда `umask`

Общая форма: `umask` *маска*

Эта команда устанавливает стандартную *маску* для создания файлов. Заданная маска накладывается на создаваемые в дальнейшем файлы по логической операции И, чтобы определить режим доступа к файлу. Если команда `umask` указана без аргументов, она выводит текущую маску. А параметр **`-S`** предписывает вывести маску в символическом виде.

Примеры:

```
$ umask              Вывести текущую маску
0002                 Запретить запись в файл другим пользователям
$ umask 022          Запретить запись в файл всем членам группы
$
```

Команда `unalias`

Общая форма: `unalias` *имена*

По этой команде указанные *имена* удаляются из списка псевдонимов. Параметр **`-a`** предписывает удалить все псевдонимы.

Команда `unset`

Общая форма: `unset` *имена*

Выполнение этой команды приводит к стиранию определений переменных или функций, перечисленных среди указанных *имен*. Переменные, доступные только для чтения, не могут быть восстановлены в исходное состояние. Параметр **`-v`** команды `unset` указывает на то, что далее следует имя переменной, тогда как параметр **`-f`** — имя функции. Если же ни один из параметров не задан, то далее предполагаются имена переменных.

Примеры:

```
unset dblist files    Удалить определения переменных dblist и files
```

Команда `until`

Общая форма:

```
until команда_t
do
      команда
      команда

done
```

Сначала в приведенном выше цикле `until`, реализуемом описываемой здесь командой, выполняется указанная *команда*$_t$ и проверяется код ее завершения. Если этот код оказывается ненулевым, то выполняются команды, указанные в промежутке между операторами `do` и `done`. Затем указанная *команда*$_t$ выполняется снова и проверяется код ее завершения. Если и на этот раз код завершения оказывается ненулевым, то команды, указанные в промежутке между операторами `do` и `done`, выполняются повторно. Этот процесс продолжается до тех пор, пока указанная *команда*$_t$ не возвратит нулевой код завершения. Далее выполнение продолжается с команды, следующей сразу же после оператора `done`.

Указанная *команда*$_t$ выполняется в самом начале цикла `until`, и поэтому команды, заданные в промежутке между операторами `do` и `done`, могут быть вообще не выполнены, если в результате первой же проверки условия данного цикла возвратится нулевой код завершения.

Пример:

```
# ожидать по 60 секунд до тех пор, пока пользователь
# jack не войдет в систему
until who | grep jack > /dev/null
do
      sleep 60
done
echo jack has logged on
```

В приведенном выше примере цикл `until` продолжается до тех пор, пока команда `grep` не возвратит нулевой код завершения (т.е. обнаружит имя пользователя `jack` в результате, выводимом командой `who`). В этот момент цикл `until` завершается и далее выполняется команда `echo`.

Команда `wait`

Общая форма: `wait` *задание*

Выполнение этой команды приводит к тому, что оболочка приостанавливает свое выполнение до тех пор, пока не завершится процесс, который обозначает указанное *задание*. Если же *задание* не указано, то оболочка ожидает завершения всех порожденных процессов. А если указано несколько заданий, то команда `wait` будет ожидать завершения всех этих заданий. Команда `wait` удобна в тех случаях, когда требуется организовать ожидание завершения процессов, отправленных на выполнение в фоновый режим. Для получения идентификатора последнего процесса, отправленного на выполнение в фоновый режим, можно воспользоваться переменной `$!`.

Пример:

```
sort large_file > sorted_file &      Отсортировать в фоновом режиме
      ...                            Продолжить обработку
wait                                 А теперь подождать завершения
                                     сортировки
plotdata sorted_file
```

Команда `while`

Общая форма:

```
while команда1
do
      команда
      команда

done
```

Сначала в приведенном выше цикле `while`, реализуемом описываемой здесь командой, выполняется указанная *команда*$_1$ и проверяется код ее завершения.

Если этот код оказывается нулевым, то выполняются команды, указанные в промежутке между операторами do и done. Затем указанная *команда*$_t$ выполняется снова и проверяется код ее завершения. Если и на этот раз код завершения оказывается нулевым, то команды, указанные в промежутке между операторами do и done, выполняются повторно. Этот процесс продолжается до тех пор, пока указанная *команда*$_t$ не возвратит ненулевой код завершения. Далее выполнение продолжается с команды, следующей сразу же после оператора done.

Указанная *команда*$_t$ выполняется в самом начале цикла while, и поэтому команды, указанные в промежутке между операторами do и done, могут быть вообще не выполнены, если в результате первой же проверки условия данного цикла возвратится ненулевой код завершения.

Пример:

```
# заполнить остальную часть буфера пустыми строками

while [ $lines -le $maxlines ]
do
      echo >> $BUFFER
      lines=$((lines + 1))
done
```

Б

Дополнительные источники информации

Имеется немало источников информации по оболочкам, действующим в режиме командной строки систем Unix, Linux и Mac OS X, но в этом приложении выбрана литература и ссылки на веб-сайты, представляющие особую ценность для программирующих на языке оболочки. Все приведенные ниже ссылки на веб-сайты были актуальны на момент выхода данной книги из печати, но, как часто бывает в Интернете, некоторые из них могут оказаться недействительными на момент чтения книги.

Оперативно доступная документация

Если документация на вашу систему недоступна в печатном виде, воспользуйтесь командой `man`, чтобы получить нужную информацию о конкретной команде Unix из так называемых *оперативных страниц руководства*. Ниже приведена общая форма данной команды.

```
man команда
```

Если вы не знаете точно наименование искомой команды, то команда `man -k` поможет вам выявить ее в системе Linux или Unix, как демонстрируется в следующем примере из версии Ubuntu системы Linux:

```
$ man -k dvd
brasero              (1) - Simple and easy to use CD/DVD burning
                           application for ...
                           (Простая команда для записи CD/DVD)
btcflash             (8) - firmware flash utility for BTC DRW1008
                           DVD+/-RW recorder.
                           (Утилита для прошивки встраиваемого ПО
                            с помощью устройства записи BTC DRW1008
                            на диски DVD+/-RW)
dvd+rw-booktype      (1) - format DVD+-RW/-RAM disk with
                           a logical format
                           (Команда логического форматирования
                            дисков DVD+-RW/-RAM)
dvd+rw-format        (1) - format DVD+-RW/-RAM disk
                           (Команда форматирования дисков
                            DVD+-RW/-RAM)
dvd+rw-mediainfo     (1) - display information about dvd
```

```
                          drive and disk
                          (Команда для вывода сведений о
                           накопителе и дисках DVD)
dvd-ram-control      (1) - checks features of DVD-RAM discs
                          (Команда, проверяющая состояние
                           дисков DVD-RAM)
growisofs            (1) - combined genisoimage frontend/DVD
                          recording program.
                          (Утилита, объединенная с клиентской
                           программой genisoimage создания образов
                           и записи дисков DVD)
rpl8                 (8) - Firmware loader for DVD drives
                          (Загрузчик встроенного ПО с дисков DVD)
$
```

В некоторых системах документация доступна в диалоговом режиме по команде `info`. Для доступа к документации достаточно набрать сначала команду `info`, а после ее запуска ввести **h**, чтобы получить справку.

Документация, доступная в Интернете

Самым лучшим местом для поиска в Интернете сведений о стандарте POSIX служит веб-сайт, доступный по адресу `www.unix.org`. На этом веб-сайте поддерживается Открытая группа (Open Group) — международный консорциум, принимавший участие вместе с институтом IEEE в работе над созданием текущей спецификации стандарта POSIX, а также имеется полное описание данной спецификации. Для ее чтения нужно сначала зарегистрироваться на данном веб-сайте, причем бесплатно. Соответствующая документация доступна по ссылке `www.unix.org/online.html`.

Фонд свободного программного обеспечения (Free Software Foundation) поддерживает оперативно доступную документацию на самые разные утилиты Linux и Unix, включая компилятор Bash и C, по ссылке `www.fsf.org/manual`.

Дэвид Корн, являющийся разработчиком оболочки Korn, поддерживает вебстраницу по адресу `www.kornshell.com`. На этой странице содержится документация, загружаемое программное обеспечение, сведения о литературе по оболочке Korn, а также ссылки на информацию о других оболочках.

Если у вас имеется доступ только к системам Microsoft Windows, но вы все же стремитесь освоить программирование на языке оболочки или хотя бы опробовать систему Linux, рекомендуется установить пакет Cygwin, доступный по адресу `www.cygwin.com`. В состав базовой системы входит оболочка Bash и немало других утилит командной строки, благодаря которым система Linux действует аналогично Unix. Но самое главное, что пакет Cygwin свободно доступен для загрузки и применения.

Литература

Ниже приведен перечень рекомендованной для чтения литературы по программированию на языке оболочки в системах Unix.

Издательство O'Reilly & Associates

В издательстве O'Reilly & Associates (`www.ora.com`) вышло в свет немало литературы по системам Linux и Unix. Тематика книг этого издательства охватывает самые разные предметы. Эти книги можно приобрести на веб-сайте издательства, а также в розничной торговле через Интернет и в книжных магазинах. На веб-сайте издательства O'Reilly & Associates можно также найти немало полезных статей по системам Linux и Unix.

К рекомендуемой литературе по системам Linux и Unix относятся следующие книги.

- *Unix in a Nutshell, 4th Edition*, A. Robbins, O'Reilly & Associates, 2005 (в русском переводе вышла под названием *UNIX. Справочник*, А. Роббинс, издательство "Кудиц-Пресс", 2006 г.).
- *Linux in a Nutshell, 6th Edition*, E. Siever, S. Figgins, R. Love and A. Robbins, O'Reilly & Associates, 2009.

Для обучения программированию на языке Perl от начального до продвинутого уровня можно порекомендовать следующую литературу.

- *Learning Perl, 6th Edition*, R. L. Schwartz, B. Foy and T. Phoenix, O'Reilly & Associates, 2011.
- *Perl in a Nutshell, 2nd Edition*, S. Spainhour, E. Siever, and N. Patwardhan, O'Reilly & Associates, 2002.

Версии команд `awk` и `sed` по стандартам POSIX и GNU хорошо описаны в следующей книге.

- *Sed & Awk, 2nd Edition*, D. Dougherty and A. Robbins, O'Reilly & Associates, 1997 (ISBN 978-1-56592-225-9).

  Для пользователей системы Mac OS рекомендуется следующая книга.
- *Learning Unix for OS X*, D. Taylor, O'Reilly & Associates, 2012.

Издательство Pearson

Изучить основы программирования на языке оболочки в системе Unix можно в следующей книге.

- *Sams Teach Yourself Shell Programming in 24 Hours, 2nd Edition*, S. Veeraraghaven, Sams Publishing, 2002.

Хорошим пособием для обучения программированию на языках C и Perl в системе Unix служит следующая книга.

- *Sams Teach Yourself Unix in 24 Hours, 5th Edition*, D. Taylor, Sams Publishing, 2016.

В приведенной ниже книге рассматривается широкий круг вопросов, связанных с FreeBSD — надежной и свободно доступной версией UNIX, которая применяется вместо Linux на многих требовательных предприятиях. В данной книге приводятся подробные сведения о версии FreeBSD, которые вряд ли удастся найти где-нибудь еще.

- *FreeBSD Unleashed, 2nd Edition*, M. Urban and B. Tiemann, Sams Publishing, 2003.

Для начинающих изучать версию FreeBSD рекомендуется следующая книга, где подробно излагаются не только самые основы, но и особенности данной операционной системы.

- *Sams Teach Yourself FreeBSD in 24 Hours*, Michael Urban and Brian Tiemann, 2002.

Исчерпывающим справочником по оболочке C служит следующая книга.

- *The Unix C Shell Field Guide*, G. Anderson and P. Anderson, Prentice Hall, 1986.

Полное описание языка `awk` дается его создателями в следующей книге.

- *The AWK Programming Language*, A. V. Aho, B. W. Kernighan, and P. J. Weinberger, Addison-Wesley, 1988.

Следующая книга служит отличным пособием для совершенствования навыков программирования в среде Unix.

- *The Unix Programming Environment*, B. W. Kernighan and R. Pike, Prentice Hall, 1984 (в русском переводе вышла под названием *Unix. Программное окружение*, Брайан У. Керниган, Роберт Пайк, издательство "Символ-Плюс", 2003 г.).

И наконец, следующая книга служит удобным пособием для совершенствования навыков программирования в среде Linux.

- *Advanced Linux Programming*, M. Mitchell, J. Oldham, and A. Samuel, New Riders Publishing, 2001 (в русском переводе вышла под названием *Программирование для Linux. Профессиональный подход*, Марк Митчел, Джеффри Оулдем, Алекс Самьюэл, ИД "Вильямс", 2004 г.).

Предметный указатель

П

Ф

Ц

КОМАНДНАЯ СТРОКА LINUX И СЦЕНАРИИ ОБОЛОЧКИ. БИБЛИЯ ПОЛЬЗОВАТЕЛЯ

2-е издание

Ричард Блум,
Кристина Бреснахэн

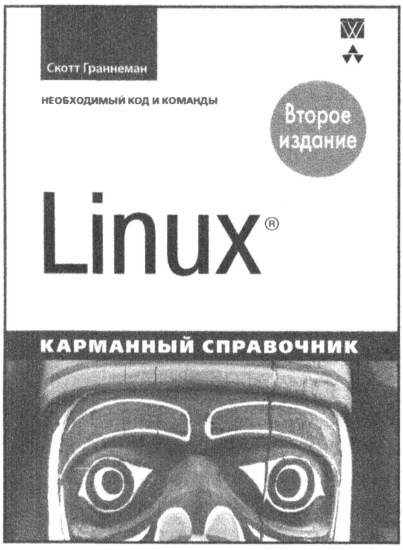
www.dialektika.com

В этой книге вы найдете все, что необходимо для освоения строковых команд и сценариев командных интерпретаторов Linux новичками и даже опытными разработчиками. Широким набором команд и средств упрощения работы, которые нелегко обнаружить самостоятельно во многих дистрибутивах Linux с интерфейсом рабочего стола, сумеют воспользоваться даже те пользователи, которые применяют систему Linux в собственных интересах. Но самое главное, это руководство включает весьма обширный массив практически применимых, удобных сценариев для опытных пользователей. С его помощью читатель быстро получит возможность автоматизировать фактически любую задачу в системе Linux.

ISBN 978-5-8459-1780-5 **в продаже**

LINUX
Карманный справочник
Второе издание

Скотт Граннеман



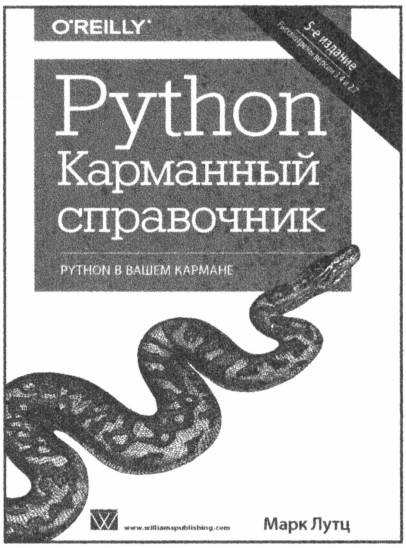
Книга представляет собой краткий справочник по основным командам операционной системы Linux. В книге содержится множество готовых к использованию фрагментов программ и команд для выполнения типичных задач в системе Linux. Книга будет полезна как для новичков, только приступающих к изучению Linux, так и для опытных пользователей, применяющих оболочку для решения разных задач: от администрирования до программирования.

www.williamspublishing.com

ISBN 978-5-8459-2101-7 **в продаже**

PYTHON
КАРМАННЫЙ СПРАВОЧНИК
ПЯТОЕ ИЗДАНИЕ

Марк Лутц



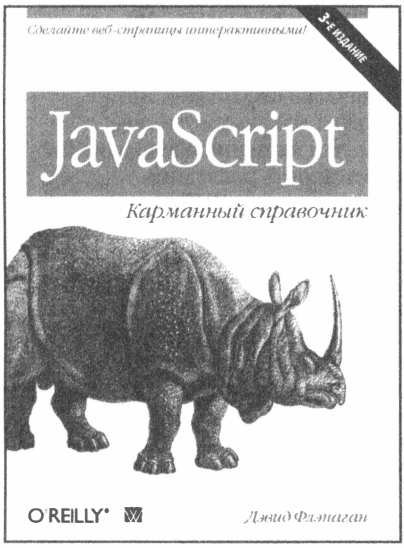
www.williamspublishing.com

Этот краткий справочник по Python составлен с учетом версий 3.4 и 2.7 и очень удобен для наведения быстрых справок при разработке программ на Python. В лаконичной форме здесь представлены все необходимые сведения о типах данных и операторах Python, специальных методах перегрузки операторов, встроенных функциях и исключениях, наиболее употребительных стандартных библиотечных модулях и других примечательных языковых средствах Python, в том числе и для объектно-ориентированного программирования. Справочник рассчитан на широкий круг читателей, интересующихся программированием на Python.

ISBN 978-5-8459-1912-0 **в продаже**

JAVASCRIPT
КАРМАННЫЙ СПРАВОЧНИК
3-Е ИЗДАНИЕ

Дэвид Флэнаган



www.williamspublishing.com

JavaScript – популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены наиболее важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений.

Главы 1–9 посвящены описанию синтаксиса последней версии языка (спецификация ECMAScript 5).

- Типы данных, значения и переменные
- Инструкции, операторы и выражения
- Объекты и массивы
- Классы и функции
- Регулярные выражения

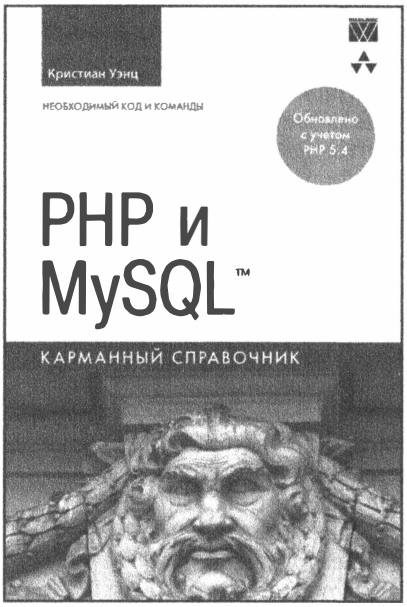
В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

ISBN 978-5-8459-1830-7

в продаже

PHP и MySQL™
Карманный справочник

Кристиан Уэнц



Эта книга, не претендуя на полноту описания всех возможностей, предоставляемых языком PHP, предлагает для рассмотрения темы, с которыми PHP-программист сталкивается практически ежедневно. Автор приложил максимум усилий, чтобы этот карманный справочник соответствовал последним стандартам языка PHP, и акцентировал внимание на новых возможностях версий PHP 5.3 и 5.4. Поскольку СУБД MySQL остается повсеместно принятым стандартом баз данных для приложений на PHP, она заслужила отдельной главы. Все листинги из книги доступны для загрузки, а их код протестирован на таких платформах, как Linux, Windows, Mac OS X и Solaris.

www.williamspublishing.com

ISBN 978-5-8459-2019-5 **в продаже**

ПРОГРАММИРОВАНИЕ КОМАНДНЫХ ОБОЛОЧЕК В UNIX, LINUX И OS X

4-Е ИЗДАНИЕ

Это полностью обновленное издание классического пособия по программированию командных оболочек в системах Unix. Следуя методике изложения материала, принятой в первоначальном издании, авторы книги уделили основное внимание стандартной оболочке POSIX, поясняя особенности разработки полезных программ в этой удобной среде, чтобы извлечь максимальную пользу из потенциала, заложенного в основу Unix и подобных ей операционных систем.

После краткого обзора команд Unix в книге подробно рассматривается поэтапный процесс создания программ или сценариев оболочки, их отладки и особенностей работы в среде оболочки. Все основные средства оболочки поясняются на многих практических примерах, что упрощает написание сценариев оболочки для конкретного применения. В книге описываются также основные средства оболочек Korn и Bash.

УЗНАЙТЕ КАК...

- Выгодно пользоваться многими утилитами, предоставляемыми системой Unix
- Писать эффективные сценарии оболочки
- Использовать встроенные в оболочку конструкции для выбора вариантов и организации циклов
- Применять эффективные механизмы заключения в кавычки
- Извлекать максимальную пользу из встроенной в оболочку предыстории команд и средств их редактирования
- Пользоваться регулярными выражениями в командах Unix
- Выгодно пользоваться специальными средствами оболочек Korn и Bash
- Выявлять основные отличия версий языка оболочки
- Изменять порядок реакции системы Unix на действия пользователя
- Настраивать среду оболочки
- Пользоваться функциями
- Отлаживать программы и сценарии оболочки

"Самая лучшая и действительно классическая книга для обучения программированию командных оболочек".
Dr. Dobb's Journal.

Стефан Кочан является автором нескольких популярных книг по ОС Unix и языку C, включая *Programming in C*, *Programming in Objective-C*, *Topics in C Programming* и *Exploring the Unix System*. Прежде он работал консультантом по программному обеспечению в компании AT&T Bell Laboratories, где составил и вел курсы по Unix и программированию на языке C.

Патрик Вуд работает техническим директором в филиале компании Electronics for Imaging, находящемся в штате Нью-Джерси. Он входил в состав инженерно-технического персонала компании Bell Laboratories, где в 1985 году познакомился со Стефаном Кочаном. Совместно они основали консультационную фирму Pipeline Associates, Inc. по ОС Unix, где Патрик занимал пост вице-президента. Кроме того, они совместно написали ряд книг, в том числе *Exploring the Unix System*, *Unix System Security*, *Topics in C Programming* и *Unix Shell Programming*.

Категория: операционные системы Unix
Предмет рассмотрения: программирование
Уровень: начальный — промежуточный

ISBN 978-5-9909445-3-4

ДИАЛЕКТИКА www.williamspublishing.com

Addison Wesley