

ПРОГРАММИРОВАНИЕ НА Shell (UNIX)

Учебное пособие

© А. Соловьев

1. ВВЕДЕНИЕ

Среди операционных систем особое место занимает Unix. Беспрецедентным является то, что ОС Unix может работать практически на всех выпускаемых платформах. UNIX - это стандарт де факто открытых и мобильных операционных систем. (поскольку название UNIX запатентовано компанией AT&T - различные юниксы называются различно: SCO UNIX, BSDI, Solaris, Linux, DG/UX, AIX и т.д.).

Это не только многозадачная, но и многопользовательская система. Она обеспечивает современный пользовательский интерфейс на базе системы X Window и межмашинную связь на базе протоколов TCP/IP и т.п.

ОС Unix была создана Кеном Томпсоном и Деннисом Ритчи в Bell Laboratories (AT&T). Широко распространяться Unix/v7 (версия 7) начала в 79 - 80-м годах. Вручение создателям Unix в 1983 году Международной премии А.Тьюринга в области программирования ознаменовало признание этой системы мировой научной (computer science) общественностью. Что также беспрецедентно.

Сколько операционных систем возшло и зашло на компьютерном небосклоне за время существования UNIX!

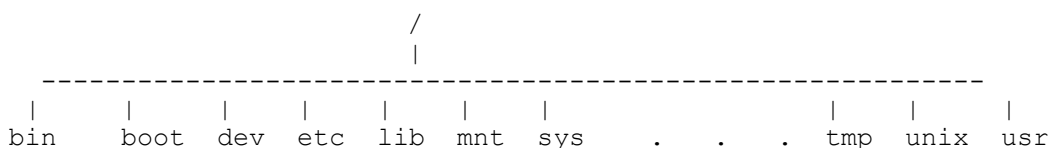
ОС Unix стоит на трех китах: язык Си, файловая система, командный язык. В дальнейшем к ним добавились система X Window и протоколы TCP/IP.

Язык Си, на котором написана сама операционная система, с одной стороны, сочетает в себе свойства языка высокого уровня: описание типов, программные структуры if, for, while и т.п., а с другой - содержит средства, присущие обычно языкам уровня ассемблера: регистровые переменные, адресную (ссылочную) арифметику, возможности работы с полями бит и отдельными битами и т.п.

2. ФАЙЛОВАЯ СИСТЕМА

Файловая система ОС Unix имеет иерархическую (древовидную) структуру. В вершинах дерева находятся каталоги (используют также термины - справочники, директории), содержащие списки файлов. Эти файлы в свою очередь могут быть либо снова

каталогами, либо обычными файлами, либо специальными файлами, представляющими различные устройства ввода-вывода.



Корневой каталог имеет имя "/". Он обычно содержит каталоги:

- bin** для наиболее используемых команд;
- usr** каталоги и обычные файлы, содержащие информацию, привлекаемую при решении задач пользователя;
- dev** для специальных файлов, представляющих устройства (дисплеи, диски...);
- etc** для хранения команд администратора системы;
- lib** важнейшие библиотеки;
- mnt** для подключения (примонтирования) новых файловых систем;
- sys** средства для изменения конфигурации системы;
- tmp** для хранения временных файлов;
- usr** каталоги и обычные файлы, содержащие информацию, привлекаемую при решении задач пользователя.

А также обычные (выполняемые) файлы:

- unix** ядро;
- boot** загрузчик.

Полные имена файлов будут: /bin, /usr, ..., /unix, /boot.

В свою очередь эти каталоги могут содержать каталоги следующего уровня. Например, каталог "usr", кроме прочего, содержит каталоги:

- bin** хранит дополнительные команды;
- games** игры;
- include** хранит фрагменты системных программ;
- lib** хранит дополнительные библиотеки.

полные имена этих файлов будут:

/usr/bin /usr/games /usr/include /usr/lib

Если в каталоге "/usr/include" содержится каталог "sys", который в свою очередь, содержит каталог "conf", то полное имя файла "conf" будет

/usr/include/sys/conf

Формальным признаком полного имени является то, что оно начинается со слэша ("/").

Относительное имя начинается не с "/", и определяют имя относительно своего местоположения. Если (пользователь?) в данный момент находится в директории /usr файловой системы, то он может обратиться к этому же файлу по относительному имени

```
include/sys/conf
```

Есть два специальных имени:

- . это "имя" текущего директория и
- .. это "имя" родительского директория (т.е. директория, находящегося на ступеньку выше данного на пути к корню).

В качестве имени файла как правило может использоваться любая последовательность из букв, цифр и подчеркиваний. Могут использоваться и другие символы, однако ряд этих символов при использовании в имени требует специального экранирования. (Лучше не пользоваться специальными символами в именах - иногда это может привести к сложностям в обращении к таким именам, поскольку спецсимволы могут иметь в shell некоторый специальный смысл).

В ряде систем длина имени ограничивается 14-ю символами (этого ограничения желательно придерживаться для переносимости файлов), однако в других системах допускаются более длинные имена - например, до 256 символов.

В общем случае не являются обязательными и какие-то расширения в именах. Хотя ряд команд требуют наличия некоторых фиксированных расширений в именах, например расширение ".c" для исходных файлов для Си-компилятора.

КСТАТИ. В ОС UNIX большие и маленькие буквы воспринимаются как различные, поэтому "FILE", "file" и "File" - это три различных имени!

ВАЖНОЕ ЗАМЕЧАНИЕ. Отдельные части файловой системы могут находиться на различных физических устройствах, например, на нескольких жестких и гибких дисках (или в различных частях одного диска). Соответствующие фрагменты (поддеревья файловой системы) монтируются (присоединяются) в единую файловую систему командой mount (обычно это функция администратора системы), после чего пользователь может обращаться к любым доступным файлам, при этом в имени никак не отражается устройство, на котором файл находится или создается (т.е. никаких "A:").

Командный язык ОС Unix - shell оперирует с командами. Более подробно о нем разговор далее, а пока рассмотрим несколько команд работы с файловой системой.

Например, в результате выполнения команды

```
ls -l /usr
```

где ls - имя команды; -l - флаг, говорящий о том, что выдача должна быть в длинном формате; /usr - имя каталога, который надо echo. На экран будет выведено

```
drwxrwxr-x 2 root 2048 nov 3 12:11 bin
-rwxr--r-- 1 root 861 may 11 20:11 boot
drwxrwxr-x 2 root 1024 jan 9 11:55 dev
drwxrwxr-x 1 root 4096 may 11 20:11 dos
drw-r--r-- 3 root 4096 nov 17 12:01 include
drwxr-xr-x 7 root 480 nov 17 12:30 lib
```

Первая строка означает, что это каталог (d-directory), где первая триада "rwx" разрешает владельцу каталога: r - читать, w - писать и x - выполнять (более точно, для файлов типа каталог w означает разрешение создавать файлы в каталоге и удалять их из него, а x разрешает доступ к файлам каталога); членам группы, в которую входит владелец, также разрешены все три операции. Последняя триада отражает права доступа прочих пользователей, которым разрешено только читать и выполнять (запрещено писать в этот файл, т.е. изменять содержимое каталога).

Далее, 2 - это число связей файла (т.е. где-то в системе есть еще одно имя, связанное с этим файлом); root - имя владельца, 2048 - число символов в файле, nov 3 12:11 - дата и время создания или последней модификации файла (3 ноября в 12-11); bin - имя файла (каталог команд).

Во второй строке указан обычный текстовый файл (boot), который прочие пользователи могут только читать.

Команда "pwd" (без флагов и аргументов) сообщает местоположение пользователя в файловой системе. С ее помощью выводится полное имя текущего каталога.

При входе в систему пользователь оказывается в определенной заранее вершине дерева. Пусть, например, это будет каталог "/usr".

Изменить местонахождение можно командой "cd <каталог>". Так можно перейти в каталог /usr/include/sys, набрав команду

```
cd /usr/include/sys
```

здесь указано полное имя, или

```
cd include/sys
```

здесь указано относительное имя.

Отличительный признак относительного имени - отсутствие символа "/" в начале.

Команда "cd .." осуществит переход вверх на предыдущий уровень. Из "/usr/include/sys" произойдет переход в "/usr/include", а команда "cd" (т.е. без параметров) осуществит

переход в начальный директорию пользователя (т.е. директорию, в котором пользователь оказывается при входе в систему).

Создать новые каталоги можно с помощью команды

```
mkdir <имена создаваемых каталогов>
```

Так команда "mkdir err new" создаст в данном каталоге два новых каталога с относительными именами "err" и "new".

Удалить пустой (не содержащий файлов) каталог можно с помощью команды

```
rmdir <имена удаляемых каталогов>
```

Удалить обычный файл можно командой

```
rm <имена удаляемых файлов>
```

Наиболее естественный для пользователя способ создания файлов - это использование текстового редактора "ed" или экранного редактора "red" (а также стандартных "vi" и "ex", или многочисленных прочих "фирменных").

В ОС Unix около 200 базовых команд - инструментальных средств, позволяющих пользователю решать многие свои проблемы, не прибегая к программированию на языках типа Си или использованию специальных пакетов.

Командой

```
rm файл-1
```

можно удалить "файл-1".

Командой

```
rmdir файл-1
```

можно удалить "файл-1", если это директорию, причем пустой (т.е. не содержит файлов).

Командой

```
mv старое-имя новое-имя
```

можно переименовать файл.

Командой

```
cp старое-имя новое-имя
```

можно скопировать файл (сохранив также старый).

Очень важна команда

```
chmod 755 расчет
```

которая превращает файл "расчет", подготовленный в редакторе, в командный, иначе "расчет" при попытке вызова не будет выполняться. Набор цифр здесь соответствует триадам двоичных представлений восьмеричных чисел и триадам прав доступа к файлам (rwx - чтение, запись, выполнение). То есть определяет, что создатель расчета может не только выполнить или распечатать текст этого расчета, но и вносить в него изменения (7:111 - rwx). А члены группы и прочие пользователи могут только читать и выполнять, но не могут изменять этот расчет (55:101101 - r-xr-x).

3. ПРОСТЕЙШИЕ СРЕДСТВА SHELL

Командный язык shell (в переводе - раковина, скорлупа) фактически есть язык программирования очень высокого уровня. На этом языке пользователь осуществляет управление компьютером. Обычно, после входа в систему вы начинаете взаимодействовать с командной оболочкой (если угодно - она начинает взаимодействовать с вами). Признаком того, что оболочка (shell) готова к приему команд служит выдаваемый ею на экран промптер. В простейшем случае это один доллар ("\$").

Shell не является необходимым и единственным командным языком (хотя именно он стандартизован в рамках POSIX [POSIX 1003.2] - стандарта мобильных систем). Например, немалой популярностью пользуется язык cshell, есть также kshell, bashell (из наиболее популярных в последнее время) и другие. Более того, каждый пользователь может создать свой командный язык. Может одновременно на одном экземпляре операционной системы работать с разными командными языками.

ОБРАТИТЕ ВНИМАНИЕ. shell - это одна из многих команд UNIX. То есть в набор команд оболочки (интерпретатора) "shell" входит команда "sh" - вызов интерпретатора "shell". Первый "shell" вызывается автоматически при вашем входе в систему и выдает на экран промптер. После этого вы можете вызывать на выполнение любые команды, в том числе и снова сам "shell", который вам создаст новую оболочку внутри прежней.

Так например, если вы подготовите в редакторе файл "f1":

```
echo Hello!
```

то это будет обычный текстовый файл, содержащий команду "echo", которая при выполнении выдает все написанное правее ее на экран. Можно сделать файл "f1" выполняемым с помощью команды "chmod 755 f1". Но его можно ВЫПОЛНИТЬ, вызвав явно команду (!) "sh" ("shell"):

```
sh f1
```

или

```
sh < f1
```

Файл можно выполнить и в текущем экземпляре "shell". Для этого существует специфическая команда "." (точка), т.е.

```
. f1
```

ВАЖНОЕ ПРЕДУПРЕЖДЕНИЕ. Не начинайте командные файлы с символа "#", хотя естественно начинать его с комментария. Дело в том, что такой командный файл в оболочке C-Shell ("csh") будет интерпретирован как выполняемый в "csh", в результате будет активизирован интерпретатор "csh".

СОВЕТ. Начинайте командный sh-файл с пустой строки или пустого оператора ":".

Поскольку UNIX - система многопользовательская, вы можете даже на персональном компьютере работать параллельно, скажем, на 12-ти экранах (переход с экрана на экран ALT/функциональная клавиша), имея на каждом экране нового (или одного и того же) пользователя со своей командной оболочкой. Можете и в графическом режиме X-Window также открыть большое число окон, а в каждом окне может быть свой пользователь со своей командной оболочкой...

Стержневым элементом языка shell является команда.

3.1. Структура команд

Команды в shell обычно имеют следующий формат:

```
<имя команды> <флаги> <аргумент(ы)>
```

Например:

```
ls -ls /usr/bin
```

ls имя команды выдачи содержимого директория,
-ls флаги ("-" - признак флагов, l - длинный формат, s - объем файлов в блоках).
/usr/bin директорий, для которого выполняется команда.

Эта команда выдаст на экран в длинном формате содержимое директория /usr/bin, при этом добавит информацию о размере каждого файла в блоках.

К сожалению, такая структура команды выдерживается далеко не всегда. Не всегда перед флагами ставится минус, не всегда флаги идут одним словом. Есть разнообразие и в представлении аргументов. К числу команд, имеющих экзотические форматы, относятся и такие "ходовые" команды, как cc, tar, dd, find и ряд других.

Как правило (но не всегда), первое слово (т.е. последовательность символов до пробела, табуляции или конца строки) shell воспринимает, как команду. Поэтому в командной строке

```
cat cat
```

первое слово будет расшифровано shell, как команда (конкатенации), которая выдаст на экран файл с именем "cat" (второе слово), находящийся в текущем директории.

3.2. Группировка команд.

Средства группировки:

; и	определяют последовательное выполнение команд;
<перевод строки>	
&	асинхронное (фоновое) выполнение предшествующей команды;
&&	выполнение последующей команды при условии нормального завершения предыдущей, иначе игнорировать;
 	выполнение последующей команды при ненормальном завершении предыдущей, иначе игнорировать.

При выполнении команды в асинхронном режиме (после команды стоит один амперсанд) на экран выводится номер процесса, соответствующий выполняемой команде, и система, запустив этот фоновый процесс, вновь выходит на диалог с пользователем.

Например, наберем (экзотическую) команду "find" в фоновом режиме для поиска в системе, начиная от корня "/", файла с именем "conf", а затем "pwd" в обычном режиме. На экране этот фрагмент будет выглядеть следующим образом:

\$ find / -name conf -print &	ввод команды "find"
288	номер (PID) фонового процесса
\$ pwd	ввод команды "pwd"
/mnt/lab/asu	результат работы "pwd"
\$	возвращение shell в промптер
/usr/include/sys/conf	результат работы "find"

Иногда необходимо, чтобы все фоновые процессы завершились, прежде чем будет выполняться какой-то расчет. Для этого служит специальная команда "wait [PID]". Эта команда ждет завершения указанного идентификатором (числом) фонового процесса. Если команда без параметра, то она ждет завершения всех фоновых процессов, дочерних для данного "sh".

Для группировки команд также могут использоваться фигурные "{}" и круглые "()" скобки. Рассмотрим примеры, сочетающие различные способы группировки: Если введена командная строка

```
k1 && k2; k3
```


где k1, k2 и k3 - какие-то команды, то "k2" будет выполнена только при успешном завершении "k1"; после любого из исходов обработки "k2" (т.е. "k2" будет выполнена, либо пропущена) будет выполнена "k3".

```
k1 && {k2; k3}
```

Здесь обе команды ("k2" и "k3") будут выполнены только при успешном завершении "k1".

```
{k1; k2} &
```

В фоновом режиме будет выполняться последовательность команд "k1" и "k2".

Фоновые процессы (как и теневую экономику) сложно уничтожить, поскольку традиционная команда "CTL/C" прерывает только процессы переднего плана. Для уничтожения фонового процесса надо знать его номер. При запуске фонового процесса на экран выдается число, соответствующее номеру (идентификатору) этого процесса (PID). Если этот номер забыт или надо убедиться, что этот процесс не закончен, с помощью команды

```
ps -aux
```

можно получить перечень идентификаторов процессов (PID), имена пользователей, текущее время, затраченное процессами, и т.д.

В выведенной таблице можно найти номера процессов, подлежащих уничтожению, например это "849" и "866". Тогда командой

```
kill -9 866 849
```

можно уничтожить эти процессы. При уничтожении процессов надо вы должны иметь то же имя пользователя, какое было приписано уничтожаемым процессам (или иметь имя привилегированного пользователя).

ПРЕДУПРЕЖДЕНИЕ. Если параллельно обрабатывается или создается файл с ОДНИМ именем (например, несколько пользователей вызвали в редактор один и тот же файл), то в системе продолжит существование тот вариант файла, который возвращен (записан) в систему последним. Это частая ошибка пользователей персональных компьютеров, которые редактируют один файл параллельно с нескольких экранов.

Круглые скобки "()", кроме выполнения функции группировки, выполняют и функцию вызова нового экземпляра интерпретатора shell.

Пусть мы находились в начальном каталоге "/mnt/lab/asu"

Тогда в последовательности команд

```
cd ../ ls; ls
```

две команды "ls" выдадут 2 экземпляра содержимого каталога "/mnt/lab", а последовательность

```
(cd ../; ls) ls
```

выдаст сначала содержимое каталога "/mnt/lab", а затем содержимое "/mnt/lab/asu", т.к. при входе в скобки вызывается новый экземпляр shell, в рамках которого и осуществляется переход. При выходе из круглых скобок происходит возврат в старый shell и в старый каталог.

3.3. Перенаправление команд

Стандартный ввод (вход) - "stdin" в ОС UNIX осуществляется с клавиатуры терминала, а стандартный вывод (выход) - "stdout" направлен на экран терминала. Существует еще и стандартный файл диагностических сообщений - "stderr", о котором речь будет чуть позже.

Команда, которая может работать со стандартным входом и выходом, называется ФИЛЬТРОМ.

Пользователь имеет удобные средства перенаправления ввода и вывода на другие файлы (устройства). Символы ">" и ">>" обозначают перенаправление вывода.

```
ls >f1
```

команда "ls" сформирует список файлов текущего каталога и поместит его в файл "f1" (вместо выдачи на экран). Если файл "f1" до этого существовал, то он будет затерт новым.

```
pwd >>f1
```

команда pwd сформирует полное имя текущего каталога и поместит его в конец файла "f1", т.е. ">>" добавляет в файл, если он непустой.

Символы "<" и "<<" обозначают перенаправление ввода.

```
wc -l <f1
```

подсчитает и выдаст на экран число строк в файле f1.

```
ed f2 <<!
```

создаст с использованием редактора файл "f2", непосредственно с терминала. Окончание ввода определяется по символу, стоящему правее "<<" (т. е. "!"). То есть ввод будет закончен, когда первым в очередной строке будет "!".

Можно сочетать перенаправления. Так

```
wc -l <f3 >f4 и wc -l >f4 <f3
```

выполняются одинаково: подсчитывается число строк файла "f3" и результат помещается в файл "f4".

Средство, объединяющее стандартный выход одной команды со стандартным входом другой, называется КОНВЕЙЕРОМ и обозначается вертикальной чертой "|".

```
ls | wc -l
```

список файлов текущего каталога будет направлен на вход команды "wc", которая на экран выведет число строк каталога.

Конвейером можно объединять и более двух команд, когда все они, возможно кроме первой и последней - фильтры:

```
cat f1 | grep -h result | sort | cat -b > f2
```

Данный конвейер из файла "f1" ("cat") выберет все строки, содержащие слово "result" ("grep"), отсортирует ("sort") полученные строки, а затем пронумерует ("cat -b") и выведет результат в файл "f2".

Поскольку устройства в ОС UNIX представлены специальными файлами, их можно использовать при перенаправлениях. Специальные файлы находятся в каталоге "/dev". Например, "lp" - печать; "console" - консоль; "ttyi" - i-ый терминал; "null" - фиктивный (пустой) файл (устройство).

Тогда, например,

```
ls > /dev/lp
```

выведет содержимое текущего каталога на печать, а `f1 < /dev/null` обнулит файл "f1".

```
sort f1 | tee /dev/lp | tail -20
```

В этом случае будет отсортирован файл "f1" и передан на печать, а 20 последних строк также будут выданы на экран.

Вернемся к перенаправлению выхода. Стандартные файлы имеют номера: 0 - stdin, 1 - stdout и 2 - stderr. Если вам не желательно иметь на экране сообщение об ошибке, вы можете перенаправить его с экрана в указанный вами файл (или вообще "выбросить", перенаправив в файл "пустого устройства" - /dev/null). Например при выполнении команды

```
cat f1 f2
```

которая должна выдать на экран последовательно содержимое файлов "f1" и "f2", выдаст вам, например, следующее

```
111111 222222
cat: f2: No such file or directory
```

где 111111 222222 - содержимое файла "f1", а файл "f2" отсутствует, о чем команда "cat" выдала сообщение в стандартный файл диагностики, по умолчанию, как и стандартный выход, представленный экраном.

Если вам не желательно такое сообщение на экране, его можно перенаправить в указанный вами файл:

```
cat f1 f2 2>f-err
```

сообщения об ошибках будут направляться (об этом говорит перенаправление "2>") в файл "f-err". Кстати, вы можете всю информацию направлять в один файл "ff", используя в данном случае конструкцию

```
cat f1 f2 >>ff 2>ff
```

Можно указать не только какой из стандартных файлов перенаправлять, но и в какой стандартный файл осуществить перенаправление.

```
cat f1 f2 2>>ff 1>&2
```

Здесь сначала "stderr" перенаправляется (в режиме добавления) в файл "ff", а затем стандартный выход перенаправляется на "stderr", которым к этому моменту является файл "ff". То есть результат будет аналогичен предыдущему.

Конструкция "1>&2" - означает, что кроме номера стандартного файла, в который перенаправить, необходимо впереди ставить "&"; вся конструкция пишется без пробелов.

- <- закрывает стандартный ввод.
- >- закрывает стандартный вывод.

3.4. Генерация имен файлов.

При генерации имен используют метасимволы:

- | | |
|----------------------|----------------------------------------------------------------------------------------------------------------------------|
| * | произвольная (возможно пустая) последовательность символов; |
| ? | один произвольный символ; |
| [...] | любой из символов, указанных в скобках перечислением и/или с указанием диапазона; |
| cat f* | выдаст все файлы каталога, начинающиеся с "f"; |
| cat *f* | выдаст все файлы, содержащие "f"; |
| cat program.? | выдаст файлы данного каталога с однобуквенными расширениями, скажем "program.c" и "program.o", но не выдаст "program.com"; |
| cat [a-d]* | выдаст файлы, которые начинаются с "a", "b", "c", "d". Аналогичный эффект дадут и команды "cat [abcd]*" и "cat [bdac]*". |

3.5. Командные файлы.

Для того, чтобы текстовый файл можно было использовать как команду, существует несколько возможностей.

Пусть с помощью редактора создан файл с именем "cmd", содержащий одну строку следующего вида:

```
date; pwd; ls
```

Можно вызвать shell как команду (!), обозначаемую "sh", и передать ей файл "cmd", как аргумент или как перенаправленный вход, т.е.

```
$ sh cmd      или      $ sh <cmd
```

В результате выполнения любой из этих команд будет выдана дата, затем имя текущего каталога, а потом содержимое каталога.

Более интересный и удобный вариант работы с командным файлом - это превратить его в выполняемый, т.е. просто сделать его командой, что достигается изменением кода защиты. Для этого надо разрешить выполнение этого файла.

Например,

```
chmod 711 cmd
```

сделает код защиты "rwx__x__x". Тогда простой вызов

```
cmd
```

приведет к выполнению тех же трех команд.

Результат будет тот же, если файл с содержимым

```
date; pwd; ls
```

представлен в виде:

```
date
pwd
ls
```

так как переход на другую строку также является разделителем в последовательности команд.

Таким образом, выполняемыми файлами могут быть не только файлы, полученные в результате компиляции и сборки, но и файлы, написанные на языке shell. Их выполнение происходит в режиме интерпретации с помощью shell-интерпретатора

Еще раз отметим, что shell-интерпретатор, это всего лишь одна из сотен команд ОС UNIX, имеющая равные с прочими привилегии.

4. СРЕДА SHELL (ПЕРЕМЕННЫЕ И ПАРАМЕТРЫ)

На языке shell можно писать командные файлы и с помощью команды "chmod" делать их выполняемыми. После этого они ни чем не отличаются от прочих команд ОС UNIX.

4.1. shell-переменные

Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний.

Значение shell-переменной - строка символов.

То, что в shell всего два типа данных: строка символов и текстовый файл, с одной стороны, позволяет легко вовлекать в программирование конечных пользователей, никогда ранее программированием не занимавшихся, а с другой стороны, вызывает некий внутренний протест у многих программистов, привыкших к существенно большему разнообразию и большей гибкости языковых средств.

Однако интересно наблюдать то, как высококлассные программисты, освоившись с "правилами игры" shell, пишут на нем программы во много раз быстрее, чем на Си, но, что особенно интересно, в ряде случаев эти программы работают даже быстрее, чем реализованные на Си. (Но это уже случаи "высшего пилотажа").

Имя переменной аналогично традиционному представлению об идентификаторе, т.е. именем может быть последовательность букв, цифр и подчеркиваний, начинающаяся с буквы или подчеркивания.

Для присваивания значений переменным может использоваться оператор присваивания "=".

```
var_1=13 - "13" - это не число, а строка из двух цифр.  
var_2="ОС UNIX" - здесь двойные кавычки (" ") необходимы, так как в строке есть пробел.
```

ВАЖНО: Обратим внимание на то, что, как переменная, так и ее значение должны быть записаны без пробелов относительно символа "=". Кстати, как видно из примеров, первым словом в командной строке может стоять не только имя команды, но и присваивание значения переменной. Об этом как раз и говорит наличие в беспробельной строке символов наличие (незаэкранированного) символа "=".

Возможны и иные способы присваивания значений shell-переменным. Так например запись,

```
DAT=`date`
```

приводит к тому, что сначала выполняется команда "date" (обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда), а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной, в данном случае "DAT".

Можно присвоить значение переменной и с помощью команды "read", которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме. Обычно команде "read" в командном файле предшествует команда "echo", которая позволяет предварительно выдать какое-то сообщение на экран. Например:

```
echo -n "Введите трехзначное число:"  
read x
```

При выполнении этого фрагмента командного файла, после вывода на экран сообщения

```
Введите трехзначное число:
```

интерпретатор остановится и будет ждать ввода значения с клавиатуры. Если вы ввели, скажем, "753" то это и станет значением переменной "x".

Одна команда "read" может прочитать (присвоить) значения сразу для нескольких переменных. Если переменных в "read" больше, чем их введено (через пробелы), оставшимся присваивается пустая строка. Если передаваемых значений больше, чем переменных в команде "read", то лишние игнорируются.

ПРЕДУПРЕЖДЕНИЕ. На самом деле интерпретатор для продолжения работы ждет лишь нажатия клавиши. Введенное вами число воспринимается им не как число, а как последовательность символов(!). Интерпретатор не проверяет, что вы ввели. Поэтому в качестве значения переменной может оказаться любая введенная абракадабра или просто нажатие, как значение пустой строки. (Для обеспечения проверки формата ввода следует написать свою команду).

При обращении к shell-переменной необходимо перед именем ставить символ "\$". Так команды

```
echo $var_2  
echo var_2
```

выдадут на экран

```
OC UNIX  
var_2
```

И еще один пример. Фрагмент командного файла:

```
echo "var_2 = $var_2"
```

выдаст на экран

```
var_2 = OC UNIX
```

В команде "echo" первое использование "var_2" - это просто текст, а второе ("\${var_2}") - это значение соответствующей переменной.

То что здесь присутствуют пробелы между именем переменной и символом присваивания, а также между символом присваивания и значением, так это потому, что здесь мы имеем дело лишь с текстом, куда подставлены значения переменных. Там, где действительно выполняется присваивание, пробелы в этих местах НЕДОПУСТИМЫ. Присваивание, скажем, w= означает присваивание переменной "w" пустой строки. Но и пустую строку лучше присваивать аккуратно, например w="".

Для того, чтобы имя переменной не сливалось со строкой, следующей за именем переменной, используются фигурные скобки.

Пусть a=/mnt/lab/asu/
тогда

```
cat /mnt/lab/asu/prim
```

и

```
cat ${a}prim
```

равноценны (т.е. "cat" выдаст на экран содержимое одного и того же файла).

Если также предположить, что в системе есть переменная "prim" и "prim=dir", то команда

```
echo ${a}$prim
```

выдаст на экран

```
/mnt/lab/asu/dir
```

4.2. Экранирование

Рассмотрим более подробно приемы экранирования, используемые в shell. В качестве средств экранирования используются двойные кавычки (" "), одинарные кавычки (') и бэк-слэш (\).

Из примеров очевидно их действие:

Можно в одной строке записывать несколько присваиваний.

```
x=22 y=33 z=$x
```



```
A="$x" B='$x' C=\$x
D="$x + $y + $z" E='$x + $y + $z' F=$x\ +\ $y\ +\ $z
```

(присваивание `G=$x + $y` не было бы выполнено из-за пробелов)

Тогда

```
echo A = $A    B = $B    C = $C
echo D = $D    E = $E    F = $F
eval echo evaluated A = $A
eval echo evaluated B = $B
eval echo evaluated C = $C
```

Выдадут на экран

```
A = 22 B = $x C = $x
D = 22 + 33 + 22 E = $x + $y + $z F = 22 + 33 + 22
evaluated A = 22
evaluated B = 22
evaluated C = 22
```

ВНИМАНИЕ. В трех последних случаях использована своеобразная команда "eval" (от evaluate - означивать), которая в подставленной в нее (в качестве аргумента) команде выполняет означивание переменных (если таковые имеются). В результате значение "A" остается прежним, поскольку "A" имеет значение "22". А переменные "B" и "C" имеют значение "\$x". За счет означивания, которое было выполнено командой "eval" - evaluated "B" и "C" дают значения "22".

Еще один пример на "eval".

Пусть

```
w=\$v v=\$u u=5
```

В результате выполнения команд

```
echo $w
eval echo $w
eval eval echo $w
```

на экран будет выведено

```
$v
$u
5
```

Приведем еще примеры, связанные с экранированием перевода строки. Пусть переменной "string" присвоено значение "массива" 2x3:

```
abc
def
```

Обратим внимание, что для избежания присваивания лишних пробелов вторая строка массива начата с первой позиции следующей строки:

```
string="abc
def"
```

Тогда три варианта записи переменной в команде "echo"

```
echo $string
echo '$string'
echo "$string"
```

дадут соответственно три различных результата:

```
abc def
$string
abc
def
```

а последовательность команд

```
echo "строка первая
строка вторая" > f1
echo 'строка первая
строка вторая' > f2
cat f1 f2
```

даст выдаст последовательно одинаковые файлы f1 и f2:

```
строка первая
строка вторая
строка первая
строка вторая
```

Заметим также, что бэк-слэш (\) не только экранирует следующий за ним символ, что позволяет использовать специальные символы просто как символы, представляющие сами себя (он может экранировать и сам себя - \\), но в командном файле бэк-слэш позволяет объединять строки в одну (экранировать конец строки).

Например, приводившийся ранее пример командной строки:

```
cat f1 | grep -h result | sort | cat -b > f2
```

может быть записан в командном файле, скажем, как

```
cat f1 | grep -h \
result | sort | cat -b > f2
```

Кстати, эффект продолжения командной строки обеспечивает и символ конвейера. В данном случае это может дать более симпатичный результат, например:

```
cat f1          |
grep -h result  |
sort            |
cat -b > f2
```

4.3. Манипуляции с shell-переменными

Несмотря на то, что shell-переменные в общем случае воспринимаются как строки, т. е. "35" - это не число, а строка из двух символов "3" и "5", в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда "expr".

Проиллюстрируем некоторые на примерах:

Выполнение командного файла:

```
x=7 y=2
a=`expr $x + $y` ; echo a=$a
a=`expr $a + 1` ; echo a=$a
b=`expr $y - $x` ; echo b=$b
c=`expr $x '*' $y` ; echo c=$c
d=`expr $x / $y` ; echo d=$d
e=`expr $x % $y` ; echo e=$e
```

выдаст на экран

```
a=9
a=10
b=-5
c=14
d=3
e=1
```

ВНИМАНИЕ. Операция умножения ("*") обязательно должна быть заэкранирована, поскольку в shell этот значок воспринимается, как спецсимвол, означающий, что на это место может быть подставлена любая последовательность символов. Следует обратить также внимание на обязательные пробелы, отделяющие переменные и знаки операций.

С командой "expr" возможны не только (целочисленные) арифметические операции, но и строковые:

```
A=`expr 'cocktail' : 'cock'` ; echo $A
B=`expr 'cocktail' : 'tail'` ; echo $B
C=`expr 'cocktail' : 'cook'` ; echo $C
D=`expr 'cock' : 'cocktail'` ; echo $D
```

На экран будут выведены числа, показывающее число совпадающих символов в цепочках (от начала). Вторая из строк не может быть длиннее первой :

```
4
0
0
0
```

И наконец, об условной замене переменных.

Если переменные, скажем "x", "y", "z", не определены, то при обращении к переменным

\${x-new}	в качестве значения "x" будет выдано "new",
\${y=new}	в качестве значения "y" будет присвоено "new",
\${z?new}	в качестве значения "z" будет выдано "z: new" и

соответствующая процедура прекращается.

Во всех этих случаях, если переменная была к этому времени определена, то ее значение используется обычным образом.

А в следующем случае наоборот, пусть переменная "v" имеет какое-то значение, тогда

`${z+new}` в качестве значения "z" будет выдано "new", а если не было присвоено значение, то пустая строка.

4.4. Экспорт переменных

В ОС UNIX существует понятие процесса. Процесс возникает тогда, когда запускается на выполнение какая-либо команда (расчет).

Например, при наборе на клавиатуре "p <Enter>" порождается процесс расчета "p". В свою очередь "p" может породить другие процессы. Допустим, что "p" вызывает расчеты "p1" и "p2", которые последовательно порождают соответствующие процессы.

У каждого процесса есть своя среда - множество доступных ему переменных. Например, до запуска расчета "p" уже существовала среда, в которой уже были определены некоторые переменные (о стандартных переменных речь пойдет несколько позже). Запуск "p" порождает новую среду; уже в ней будут порождены расчеты "p1" и "p2".

Переменные локальны в рамках процесса, в котором они объявлены, т.е. где им присвоены значения (описание переменных отсутствует - они все одного типа). Для того, чтобы они были доступны и другим порождаемым процессам, надо передать их явным образом. Для этого используется встроенная команда "export".

Пример.

Пусть расчет (командный файл) "p", имеющий вид:

```
# расчет p
echo Расчет p
varX=0 varY=1
echo varX=$varX varY=$varY
export varY
p1 # вызов расчета p1
p2 # вызов расчета p2
echo Снова расчет p: varX=$varX varY=$varY
```

вызывает командные файлы "p1" и "p2", имеющие вид:

```
# расчет p1
echo Расчет p1
echo varX=$varX varY=$varY
varX=a varY=b
echo varX=$varX varY=$varY
export varX
# расчет p2
```

```

echo Расчет p2
echo varX=$varX varY=$varY
varX=A varY=B
echo varX=$varX varY=$varY
export varY

```

На экран будут выданы следующая информация:

```

Расчет p
varX=0 varY=1
Расчет p1
varX= varY=1
varX=a varY=b
Расчет p2
varX= varY=1
varX=A varY=B
Снова расчет p: varX=0 varY=1

```

Из примера видно, что значения переменных экспортируются только в вызываемые расчеты (и не передаются "вверх" и "вбок"). Экспортировать переменные можно и командой "set" с флагом "-a".

НА ВСЯКИЙ СЛУЧАЙ заметим, что на передачу значений переменных никакого влияния не оказывает "физическое" взаимное расположение (файлов) расчетов в файловой системе.

4.5. Параметры

В командный файл могут быть переданы параметры. В shell используются позиционные параметры (т.е. существенна очередность их следования). В командном файле соответствующие параметрам переменные (аналогично shell-переменным) начинаются с символа "\$", а далее следует одна из цифр от 0 до 9:

Пусть расчет "ехамр-1" вызывается с параметрами "sock" и "tail". Эти параметры попадают в новую среду под стандартными именами "1" и "2". В (стандартной) переменной с именем "0" будет храниться имя вызванного расчета.

При обращении к параметрам перед цифрой ставится символ доллара "\$" (как и при обращении к переменным):

- \$0** соответствует имени данного командного файла;
- \$1** первый по порядку параметр;
- \$2** второй параметр и т.д.

Пусть командный файл с именем "ехамр-1" имеет вид

```

echo Это расчет $0:
sort $2 >> $1
cat $1

```

а файлы "sock" и "tail" содержат соответственно

```

sock:

```

Это отсортированный файл:

```
tail:
  1
  3
  2
```

Тогда после вызова команды

```
examp-1 cock tail
```

на экране будет

```
Это расчет examp-1:
Это отсортированный файл:
1
2
3
```

Поскольку число переменных, в которые могут передаваться параметры, ограничено одной цифрой, т.е. 9-ю ("0", как уже отмечалось имеет особый смысл), то для передачи большего числа параметров используется специальная команда "shift".

Рассмотрим ее действие на примере.

Пусть командный файл "many" вызывается с 13-ю параметрами

```
many 10 20 30 40 50 60 70 80 90 100 110 120 130
```

И имеет вид

```
###
# many: Передача большого числа параметров.
echo "$0: Много параметров"
echo " Общее число параметров = $#
Исходное состояние: $1 $5 $9 "
shift
echo "1 сдвиг: первый=$1 пятый=$5 девятый=$9"
shift 2
echo "1 + 2 = 3 сдвига: первый=$1 пятый=$5 девятый=$9"
perem=`expr $1 + $2 + $3`
echo $perem
```

В результате первого применения команды "shift" второй параметр расчета вызывается как \$1, третий параметр вызывается как \$2, ... десятый параметр, который был исходно недоступен, вызывается как \$9. Но стал недоступным первый параметр!

После выполнения этого расчета на экране будет:

```
many: Много параметров
Общее число параметров = 13
Исходное состояние: 10 50 90
1 сдвиг: первый=20 пятый=60 девятый=100
1 + 2 = 3 сдвига: первый=40 пятый=80 девятый=120
150
```

Своеобразный подход к параметрам дает команда "set".

Например, фрагмент расчета

```
set a b c
echo первый=$1 второй=$2 третий=$3
```

выдаст на экран

```
первый=a второй=b третий=c
```

т.е. команда "set" устанавливает значения параметров. Это бывает очень удобно.

Например, команда "date" выдает на экран текущую дату, скажем, "Mon May 01 12:15:10 2000", состоящую из пяти слов, тогда

```
set `date`
echo $1 $3 $5
```

выдаст на экран

```
Mon 01 2000
```

Команда "set" позволяет также осуществлять контроль выполнения программы, например:

set -v	на терминал выводятся строки, читаемые shell.
Set +v	отменяет предыдущий режим.
Set -x	на терминал выводятся команды перед выполнением.
Set +x	отменяет предыдущий режим.

Команда "set" без параметров выводит на терминал состояние программной среды (см далее).

4.6. Подстановки shell-интерпретатора

Перед началом непосредственной интерпретации и выполнением команд, содержащихся в командных файлах, shell выполняет различные виды подстановок:

1. **ПОДСТАНОВКА РЕЗУЛЬТАТОВ.** Выполняются все команды, заключенные в обратные кавычки, и на их место подставляется результат.
2. **ПОДСТАНОВКА ЗНАЧЕНИЙ ПАРАМЕТРОВ И ПЕРЕМЕННЫХ.** То есть слова, начинающиеся на "\$", заменяются соответствующими значениями переменных и параметров.
3. **ИНТЕРПРЕТАЦИЯ ПРОБЕЛОВ.** Заэкранированные пробелы игнорируются.

4. **ГЕНЕРАЦИЯ ИМЕН ФАЙЛОВ.** Проверяются слова на наличие в них спецсимволов ("*", "?", "[]") и выполняются соответствующие генерации.

4.7. Программная среда

Каждый процесс имеет среду, в которой он выполняется. Shell использует ряд переменных этой среды.

Если вы наберете команду "set" без параметров, то на экран будет выдана информация о ряде стандартных переменных, созданных при входе в систему (и передаваемых далее всем вашим новым процессам "по наследству"), а также переменных, созданных и экспортируемых вашими процессами.

Конкретный вид и содержание выдаваемой информации в немалой степени зависит от того, какая версия UNIX используется и как инсталлирована система.

Вот лишь часть того, что выдала мне команда "set":

```
HOME=/home/sae
PATH=/usr/local/bin:/usr/bin:/bin:./usr/bin/X11:
IFS=
LOGNAME=sae
MAIL=/var/spool/mail/sae
PWD=/home/sae/STUDY/SHELL
PS1=${PWD}:" "
PS2=>
SHELL=/bin/bash
TERM=linux
TERMCAP=console|con80x25|dumb|linux:li#25:co#80::
UID=501
perem=stroka
x=5
```

Прокомментируем эти присваивания значений переменным.

HOME=/home/sae

это имя домашнего директория, в котором пользователь (в данном случае я) оказывается после входа в систему. То есть, правильно набрав имя и пароль, я окажусь в директории "/home/sae".

PATH=/bin:/usr/bin:./usr/local/bin :/usr/bin/X11	<p>эта переменная задает последовательность файлов (ТРОПУ), которые просматривает "shell" в поисках команды. Имена файлов разделяются здесь двоеточиями. Последовательность просмотра соответствует очередности следования имен в тропе. НО ПЕРВОНАЧАЛЬНО поиск происходит среди так называемых встроенных команд. В число встроенных команд входят наиболее часто используемые команды, например "echo", "cd", "pwd", "date". После этого система просматривает директорий "/bin", в котором могут находиться команды "sh", "cp", "mv", "ls" и т.п. Затем директорий "/usr/bin" с командами "cat", "cc", "expr", "nroff", "man" и многими другими. Далее поиск происходит в текущем директории (".", или другое обозначение "пусто", т.е. ""), где скорее всего находятся написанные вами команды (расчеты).</p>
IFS=	<p>После набора командной строки и нажатия <Enter> "shell" (после выполнения необходимых подстановок) распознает имя, соответствующее команде и осуществляет ее поиск в директориях, перечисленных в тропе. Если команда размещена вне этих директориев, она не будет найдена. Если присутствует несколько команд с одинаковым именем, то вызвана будет та, которая расположена в директории, просматриваемом первым.</p>
LOGNAME=sae MAIL=/var/spool/mail/sae PWD=/home/sae/STUDY/SHELL PS1=\${PWD}:" "	<p>Тропу, как и прочие переменные, можно легко менять, добавляя, переставляя или исключая директории. (Кстати, представленная тропа получена из "настоящей" путем сокращений и перестановок). (Внутренний Разделитель Полей) перечисляет символы, которые служат для разделения слов (полей). Таковыми являются "пробел", "табуляция" и "перевод строки", поэтому здесь слева от присваивания ничего не видно и занято две строки.</p>
PS2=>	<p>имя входа ("имя" пользователя). имя файла, в который поступает (электронная) почта. имя текущего директория вид промтера. В данном случае в промптере будет выдаваться имя текущего директория двоеточие и пробел. То есть здесь будет "/home/sae/STUDY/SHELL:". этот промтер (здесь ">") используется как приглашение к продолжению ввода (в очередной строке) незаконченной команды. Например, наберите открывающую скобку "(" и после нажатия <Enter> в следующей строке вы увидите этот промптер. Если пока не знаете, что дальше делать, наберите закрывающую скобку ")" и он исчезнет.</p>

`SHELL=/bin/bash`

эта переменная указывает оболочку, которую использует пользователь. В данном случае используется не (стандартный) shell ("sh"), а "продвинутая" версия .

`TERM=linux`

указание типа терминала. -"bash", написанная тем же автором (Bourne-Again SHell)

```
TERMCAP=console|con80x25|dumb|
linux:li#25:co#80::
```

`UID=501`

(TERMinal CAPacity) это (очень сильно) обрезанная строка задания параметров терминала.

`perem=stroka`

идентификатор пользователя (мой "501").

переменные, которые ввел пользователь.

`x=5`

Исходная среда устанавливается автоматически при входе в систему с использованием файлов типа `"/etc/rc"` и `"/etc/.profile"`.

ВАЖНОЕ ЗАМЕЧАНИЕ. Один из способов просто изменить среду (например, тропу поиска команд, вид промтера, вид оболочки, цвет экрана и т.п.) можно, разместив эту информацию в своем домашнем директории в специализированном файле `".profile"` (`${HOME}/.profile`), присвоив нужные значения переменным среды. То есть вызвать этот файл в редактор и написать, что пожелаете). Тогда при каждом вашем входе в систему этот файл будет автоматически выполняться и устанавливать новую среду. Этот файл должен **ОБЯЗАТЕЛЬНО** размещаться в вашем **ДОМАШНЕМ** директории (директории входа).

Если вы внесли изменения в `".profile"`, то для переноса этих изменений в среду необходимо выполнить этот файл. Для этого можно выйти и заново войти в систему, а можно воспользоваться (специально для этого случая созданной) командой `."` без выхода из системы, т.е.

```
. .profile
```

Следует иметь в виду, что имена файлов, начинающиеся с точки, вообще имеют особый статус. Так, они не выдаются на экран простой командой `"ls"` - необходимо вызывать эту команду с флагом `"-a"`. Кстати, и не уничтожаются огульно командой `"rm *"`.

Дописать новый свой директорий `"my"` в тропу команд можно, записав в `".profile"`, например

```
PATH=${PATH}:/home/sae/my
```

или

```
PATH=${PATH}:${HOME}/my
```

Как правило, устанавливаемые переменные среды следует экспортировать. Например,

```
export TERM PATH REDKEYS MAIL
```

Кроме определения переменных в ".profile" можно выполнить команды, например команда

```
stty -lcase
```

установит терминал в режим "большие и маленькие буквы"; а команда

```
cat заставка
```

выдаст на экран заставку, которую вы сами подготовите в файле "заставка" с учетом ваших эстетических пристрастий и художественных способностей.

Сам интерпретатор shell автоматически присваивает значения следующим переменным (параметрам):

- ? значение, возвращенное последней командой;
- \$ номер процесса;
- ! номер фонового процесса;
- # число позиционных параметров, передаваемых в shell;
- * перечень параметров, как одна строка;
- @ перечень параметров, как совокупность слов;
- флаги, передаваемые в shell.

При обращении к этим переменным (т.е. при использовании их в командном файле - shell-программе) следует впереди ставить "\$".

Пример. Вызов расчета

```
specific par1 par2 par3
```

имеющего вид

```
###  
# specific: Специальные параметры (переменные)  
echo $0 - имя расчета  
echo $? - код завершения  
echo $$ - идентификатор последнего процесса  
echo $! - идентификатор последнего фонового процесса  
echo  
echo $* - значения параметров, как строки  
echo @$ - значения параметров, как слов  
echo  
set -au  
echo $- - режимы работы интерпретатора
```

Выдаст на экран

```
specific - имя расчета  
0 - код завершения  
499 - идентификатор последнего процесса  
98 - идентификатор последнего фонового процесса  
par1 par2 par3 - значения параметров, как строки  
par1 par2 par3 - значения параметров, как слов  
au - режимы работы интерпретатора
```

Код "0" соответствует нормальному завершению процесса.

Важную роль при создании уникальных файлов играет специальная переменная "\$\$", значение которой соответствует номеру процесса, выполняющего данный расчет. Каждый новый расчет, выполняемый компьютером, инициирует один или несколько процессов, автоматически получающих номера по порядку. Поэтому, используя номер процесса в качестве имени файла, можно быть уверенным, что каждый новый файл будет иметь новое имя (не запишется на место уже существующего). Достоинство является и главным недостатком такого способа именования файлов. Неизвестно, какие имена будут присвоены файлам. И, если в рамках данного процесса можно найти файл "не глядя", т.е., обратившись к нему, используя \$\$, то потом такие файлы можно легко потерять. Это создает дополнительные проблемы при отладке программ.

"echo" без параметров выводит пустую строку.

Различия \$* и \$@ состоит в том, что первая переменная может быть представлена как

```
"par1 par2 par3"
```

а вторая как

```
"par1" "par2" "par3"
```

Пример, иллюстрирующий различия "\$*" и "\$@" будет рассмотрен в связи с оператором "for".

Для иллюстрации мы установили командой "set" режимы интерпретатора ("a" - все последующие переменные экспортируются; "u" - отсутствие параметра считать ошибкой), что и отразилось в специальной переменной "\$-".

5. ПРОГРАММНЫЕ СТРУКТУРЫ

Как во всяком языке программирования в тексте на языке shell могут быть комментарии. Для этого используется символ "#". Все, что находится в строке (в командном файле) левее этого символа, воспринимается интерпретатором как комментарий. Например,

```
# Это комментарий.  
## И это.  
### И это тоже.
```

Как во всяком процедурном языке программирования в языке shell есть операторы. Ряд операторов позволяет управлять последовательностью выполнения команд. В таких операторах часто необходима проверка условия, которая и определяет направление продолжения вычислений.

5.1. Команда test ("[" ")

Команда `test` проверяет выполнение некоторого условия. С использованием этой (встроенной) команды формируются операторы выбора и цикла языка `shell`.

Два возможных формата команды:

```
test условие
```

или

```
[ условие ]
```

мы будем пользоваться вторым вариантом, т.е. вместо того, чтобы писать перед условием слово `"test"`, будем заключать условие в скобки, что более привычно для программистов.

На самом деле `shell` будет распознавать эту команду по открывающей скобке `"["`, как слову `(!)`, соответствующему команде `"test"`. Уже этого достаточно, чтобы предупредить о распространенной ошибке начинающих: Между скобками и содержащимся в них условием обязательно должны быть пробелы.

Пробелы должны быть и между значениями и символом сравнения или операции (как, кстати, и в команде `"expr"`). Не путать с противоположным требованием для присваивания значений переменным.

В `shell` используются условия различных "типов".

УСЛОВИЯ ПРОВЕРКИ ФАЙЛОВ:

-f file	файл "file" является обычным файлом;
-d file	файл "file" - каталог;
-c file	файл "file" - специальный файл;
-r file	имеется разрешение на чтение файла "file";
-w file	имеется разрешение на запись в файл "file";
-s file	файл "file" не пустой.

Примеры. Вводя с клавиатуры командные строки в первом случае получим подтверждение (код завершения `"0"`), а во втором - опровержение (код завершения `"1"`). `"specific"` - имя существующего файла.

```
[ -f specific ] ; echo $?  
0  
[ -d specific ] ; echo $?  
1
```

УСЛОВИЯ ПРОВЕРКИ СТРОК:

str1 = str2	строки "str1" и "str2" совпадают;
str1 != str2	строки "str1" и "str2" не совпадают;
-n str1	строка "str1" существует (непустая);

-z str1 строка "str1" не существует (пустая).

Примеры.

```
x="who is who"; export x; [ "who is who" = "$x" ]; echo $?
0
x=abc ; export x ; [ abc = "$x" ] ; echo $?
0
x=abc ; export x ; [ -n "$x" ] ; echo $?
0
x="" ; export x ; [ -n "$x" ] ; echo $?
1
```

ВАЖНОЕ ЗАМЕЧАНИЕ. Команда "test" дает значение "истина" (т.е. код завершения "0") и просто если в скобках стоит непустое слово.

```
[ privet ] ; echo $?
0
[ ] ; echo $?
1
```

Кроме того, существуют два стандартных значения условия, которые могут использоваться вместо условия (для этого не нужны скобки).

```
true ; echo $?
0
false ; echo $?
1
```

УСЛОВИЯ СРАВНЕНИЯ ЦЕЛЫХ ЧИСЕЛ:

x -eq y	"x" равно "y",
x -ne y	"x" не равно "y",
x -gt y	"x" больше "y",
x -ge y	"x" больше или равно "y",
x -lt y	"x" меньше "y",
x -le y	"x" меньше или равно "y".

То есть в данном случае команда "test" воспринимает строки символов как целые (!) числа. Поэтому во всех остальных случаях "нулевому" значению соответствует пустая строка. В данном же случае, если надо обнулить переменную, скажем, "x", то это достигается присваиванием "x=0".

Примеры.

```
x=abc ; export x ; [ abc -eq "$x" ] ; echo $?
["": integer expression expected before -eq
x=321 ; export x ; [ 321 -eq "$x" ] ; echo $?
0
x=3.21 ; export x ; [ 3.21 -eq "$x" ] ; echo $?
["": integer expression expected before -eq
x=321 ; export x ; [ 123 -lt "$x" ] ; echo $?
0
```

СЛОЖНЫЕ УСЛОВИЯ:

Реализуются с помощью типовых логических операций:

- ! (not) инвертирует значение кода завершения.
- o (or) соответствует логическому "ИЛИ".
- a (and) соответствует логическому "И".

ПРЕДУПРЕЖДЕНИЕ. Не забывайте о пробелах.

Примеры.

```
[ ! privet ] ; echo $?
1
x=privet; export x; [ "$x" -a -f specific ] ; echo $?
0
x="";export x; [ "$x" -a -f specific ] ; echo $?
1
x="";export x; [ "$x" -a -f specific -o privet ] ; echo $?
0
x="";export x; [ "$x" -a -f specific -o ! privet ] ; echo $?
1
```

СОВЕТ. Не злоупотреблять сложными условиями.

5.2. Условный оператор "if"

В общем случае оператор "if" имеет структуру

```
if условие
then список
    [elif условие
    then список]
[else список]
fi
```

Здесь "elif" сокращенный вариант от "else if" может быть использован наряду с полным, т.е. допускается вложение произвольного числа операторов "if" (как и других операторов). Разумеется "список" в каждом случае должен быть осмысленный и допустимый в данном контексте.

Конструкции

```
[elif условие
then список]
```

и

```
[else список]
```

не являются обязательными (в данном случае для указания на необязательность конструкций использованы квадратные скобки - не путать с квадратными скобками команды "test!").

Самая усеченная структура этого оператора

```
if условие
```

```
    then список
fi
```

если выполнено условие (как правило это ком получен код завершения "0", то выполняется "список", иначе он пропускается.

Обратите внимание, что структура обязательно завершается служебным словом "fi". Число "fi", естественно, всегда должно соответствовать числу "if".

Примеры.

Пусть написан расчет "if-1"

```
if [ $1 -gt $2 ]
then pwd
else echo $0 : Hello!
Fi
```

Тогда вызов расчета

```
if-1 12 11
```

даст

```
/home/sae/STUDY/SHELL
```

а

```
if-1 12 13
```

даст

```
if-1 : Hello!
```

Возможно использовать в условии то свойство shell, что команды могут выдавать различный код завершения. Это напоминает приемы программирования на Си. Пусть расчет "if-2" будет

```
if a=`expr "$1" : "$2"`
then echo then a=$a code=$?
else echo else a=$a code=$?
Fi
```

тогда вызов

```
if-2 by by
```

даст

```
then a=2 code=0
```

а


```
if-2  by be
```

даст

```
else a=0 code=1
```

Еще пример на вложенность

```
###
# if-3: Оценка достижений
echo -n " А какую оценку получил на экзамене?: "
read z
if [ $z = 5 ]
then echo Молодец !
elif [ $z = 4 ]
then echo Все равно молодец !
elif [ $z = 3 ]
then echo Все равно !
elif [ $z = 2 ]
then echo Все !
else echo !
fi
```

Можно обратить внимание на то, что желательно использовать сдвиги при записи программ, чтобы лучше выделить вложенность структур.

5.3. Оператор вызова ("case") Оператор выбора "case" имеет структуру:

```
case строка in
  шаблон) список команд;;
  шаблон) список команд;;
  ...
esac
```

Здесь "case" "in" и "esac" - служебные слова. "Строка" (это может быть и один символ) сравнивается с "шаблоном". Затем выполняется "список команд" выбранной строки. Непривычным будет служебное слово "esac", но оно необходимо для завершения структуры.

Пример.

```
###
# case-1: Структура "case".
# Уже рассматривавшийся в связи со
# структурой "if" пример проще и
# нагляднее можно реализовать с
# помощью структуры "case".
echo -n " А какую оценку получил на экзамене?: "
read z
case $z in
  5) echo Молодец ! ;;
  4) echo Все равно молодец ! ;;
  3) echo Все равно ! ;;
  2) echo Все ! ;;
  *) echo ! ;;
esac
```

Непривычно выглядят в конце строк выбора ";;", но написать здесь ";" было бы ошибкой. Для каждой альтернативы может быть выполнено несколько команд. Если эти команды

будут записаны в одну строку, то символ ";" будет использоваться как разделитель команд.

Обычно последняя строка выбора имеет шаблон "*", что в структуре "case" означает "любое значение". Эта строка выбирается, если не произошло совпадение значения переменной (здесь \$z) ни с одним из ранее записанных шаблонов, ограниченных скобкой ") ". Значения просматриваются в порядке записи.

```
###
# case-2: Справочник.
#         Для различных фирм по имени выдается
#         название холдинга, в который она входит
case $1 in
    ONE|TWO|THREE) echo Холдинг: ZERO      ;;
                  MMM|WWW) echo Холдинг: Not-Net ;;
    Hi|Hello|Howdoing) echo Холдинг: Привет! ;;
                    *) echo Нет такой фирмы ;;
esac
```

При вызове "case-2 Hello" на экран будет выведено:

```
Холдинг: Привет!
```

А при вызове "case-2 HELLO" на экран будет выведено:

```
Нет такой фирмы
```

Коль скоро слово "case" переводится как "выбор", то это как бы намек на то, что можно эту структуру использовать для реализации простейших меню.

```
###
# case-3: Реализация меню с помощью команды "case"
echo "Назовите файл, а затем (через пробел)
наберите цифру, соответствующую требуемой
обработке:
    1 - отсортировать
    2 - выдать на экран
    3 - определить число строк "
read x y # x - имя файла, y - что сделать
case $y in
    1) sort < $x      ;;
    2) cat < $x       ;;
    3) wc -l < $x     ;;
    *) echo "
        Мы не знаем
        такой команды ! " ;;
esac
```

Разумеется, желания могут быть более сложные и на месте отдельных команд могут быть последовательности команд или вызовы более сложных расчетов.

Напишем команду "case-4", которая добавляет информацию к файлу, указанного первым параметром (если параметр один), со стандартного входа, либо (если 2 параметра) из файла, указанного в качестве первого параметра:

```
###
# case-4: Добавление в файл.
#         Использование стандартной переменной.
```

```
# "$#" - число параметров при вводе расчета
# ">>" - перенаправление с добавлением в файл
case $# in
    1) cat >> $1 ; ;
    2) cat >> $2 < $1 ; ;
    *) echo "Формат: case-4 [откуда] куда" ; ;
esac
```

"\$1" (при "\$#=1") - это имя файла, в который происходит добавление со стандартного входа.

"\$1" и "\$2" (при "\$#=2") - это имена файлов, из которого ("\$1") и в который ("\$2") добавлять.

Во всех других случаях (*) выдается сообщение о том, каким должен быть правильный формат команды.

5.4. Оператор цикла с перечислением ("for")

Оператор цикла "for" имеет структуру:

```
for имя [in список значений]
do
    список команд
done
```

где "for" - служебное слово определяющее тип цикла, "do" и "done" - служебные слова, выделяющие тело цикла. Не забывайте про "done"! Фрагмент "in список значений" может отсутствовать.

Пусть команда "lsort" представлена командным файлом

```
for i in f1 f2 f3
do
    proc-sort $i
done
```

В этом примере имя "i" играет роль параметра цикла. Это имя можно рассматривать как shell-переменную, которой последовательно присваиваются перечисленные значения (i=f1, i=f2, i=f3), и выполняется в цикле команда "procsort".

Часто используется форма "for i in *", означающая "для всех файлов текущего каталога".

Пусть "proc-sort" в свою очередь представляется командным файлом

```
cat $1 | sort | tee /dev/lp > ${1}_sorted
```

т.е. последовательно сортируются указанные файлы, результаты сортировки выводятся на печать ("/dev/lp") и направляются в файлы f1_sorted f2_sorted и f3_sorted

Можно сделать более универсальной команду "lsort", если не фиксировать перечень файлов в команде, а передавать произвольное их число параметрами.

Тогда головная программа будет следующей:

```
for i
do
    proc-sort $i
done
```

Здесь отсутствие после "i" служебного слова "in" с перечислением имен говорит о том, что список поступает через параметры команды. Результат предыдущего примера можно получить, набрав

```
lsort f1 f2 f3
```

Усложним ранее рассматривавшуюся задачу (под именем "case-2") определения холдинга фирмы. Теперь можно при вызове указывать произвольное количество фирм. При отсутствии в структуре оператора "for" фрагмента "in список значений", значения берутся из параметров вызывающей команды.

```
###
# holding: Справочник.
#           Для различных фирм по имени выдается
#           название холдинга, в который она входит
for i
do
    case $i in
        ONE|TWO|THREE) echo Холдинг: ZERO ;;
        MMM|WWW)      echo Холдинг: Not-Net ;;
        Hi|Hello|Howdoing) echo Холдинг: Привет! ;;
        *)             echo Нет такой фирмы ;;
    esac
done
```

При вызове "holding Hello HELLO ONE" на экране будет:

```
Холдинг: Привет!
Нет такой фирмы
Холдинг: Not-Net
```

Еще пример.

```
###
# subdir: Выдает имена всех поддиректориев
#         директория с именем $dir
#         for i in $dir/*
#         do
#             if [ -d $i ]
#             then echo $i
#             fi
#         done
```

Следующий расчет иллюстрирует полезный, хотя и с долей трюкачества, способ повторения одних и тех же действий несколько раз. Переменная "i" принимает здесь пять значений: 1, 2, 3, 4, 5, но внутри цикла эта переменная отсутствует и поэтому ее значение никакой роли не играет и ни чего не меняет. С таким же успехом переменная "i" могла

принимать значения, скажем ф о к у с , а в результате точно также было бы пять раз повторено одно и то же вычисление содержимого цикла без изменений.

```
###
# print-5: Организации пятикратного выполнения команды
for i in 1 2 3 4 5
do
    cat file-22 > /dev/lp
done
```

Расчет "print-n" иллюстрирует еще одну полезную возможность в использовании цикла "for". Здесь, после "for i ...", отсутствуют "in ..." и перечень имен, т.е. перечнем имен для "i" становится перечень параметров, а следовательно количество печатаемых экземпляров можно менять.

```
###
# print-n: Задание числа копий
#           через параметры
for i
do
    cat file-22 > /dev/lp
done
```

Смысл не изменится, если первую строку расчета записать как

```
for i in $*
```

поскольку значение "\$*" - есть список значений параметров.

Отметим различие в специальных переменных "\$*" и "\$@", представляющих перечень параметров. Первый представляет параметры, как строку, а второй, как совокупность слов.

Пусть командный файл "cmp" имеет вид:

```
for i in "$*"
do
    echo $i
done
echo
for i in "$@"
do
    echo $i
done
```

При вызове

```
cmp aa bb cc
```

на экран будет выведено

```
aa bb cc
aa
bb
cc
```

5.5. Оператор цикла с истинным условием ("while")

Структура "while", также обеспечивающая выполнение расчетов, предпочтительнее тогда, когда неизвестен заранее точный список значений параметров или этот список должен быть получен в результате вычислений в цикле.

Оператор цикла "while" имеет структуру:

```
while условие
do
    список команд
done
```

где "while" - служебное слово определяющее тип цикла с истинным условием. Список команд в теле цикла (между "do" и "done") повторяется до тех пор, пока сохраняется истинность условия (т.е. код завершения последней команды в теле цикла равен "0") или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие должно выполняться.

```
###
# print-50: Структура "while"
#           Расчет позволяет напечатать 50
#           экземпляров файла "file-22"
n=0
while [ $n -lt 50 ]      # пока n < 50
do
    n=`expr $n + 1`
    cat file-22 > /dev/lp
done
```

Обратим внимание на то, что переменной "n" вначале присваивается значение 0, а не пустая строка, так как команда "expr" работает с shell-переменными как с целыми числами, а не как со строками.

```
n=`expr $n + 1`
```

т.е. при каждом выполнении значение "n" увеличивается на 1.

Как и вообще в жизни, можно реализовать то же самое и сложнее. Расчет "pr-br" приведен для иллюстрации бесконечного цикла и использования команды "break", которая обеспечивает прекращение цикла.

```
###
# pr-br: Структура "while" с "break"
#           Расчет позволяет напечатать 50
#           экземпляров файла "file-22"
n=0
while true
do
    if [ $n -lt 50 ]      # если n < 50
    then n=`expr $n + 1`
    else break
    fi
    cat file-22 > /dev/lp
done
```

Команда "break [n]" позволяет выходить из цикла. Если "n" отсутствует, то это эквивалентно "break 1". "n" указывает число вложенных циклов, из которых надо выйти, например, "break 3" - выход из трех вложенных циклов.

В отличие от команды "break" команда "continue [n]" лишь прекращает выполнение текущего цикла и возвращает на НАЧАЛО цикла. Она также может быть с параметром. Например, "continue 2" означает выход на начало второго (если считать из глубины) вложенного цикла.

Команда "exit [n]" позволяет выйти вообще из процедуры с кодом возврата "0" или "n" (если параметр "n" указан). Эта команда может использоваться не только в циклах. Даже в линейной последовательности команд она может быть полезна при отладке, чтобы прекратить выполнение (текущего) расчета в заданной точке.

5.6. Оператор цикла с ложным условием ("until")

Оператор цикла "until" имеет структуру:

```
until условие
do
    список команд
done
```

где "until" - служебное слово определяющее тип цикла с ложным условием. Список команд в теле цикла (между "do" и "done") повторяется до тех пор, пока сохраняется ложность условия или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие не должно выполняться.

Отличие от оператора "while" состоит в том, что условие цикла проверяется на ложность (на ненулевой код завершения последней команды тела цикла) проверяется ПОСЛЕ каждого (в том числе и первого!) выполнения команд тела цикла.

Программистов, знакомых с операторами "until" в других языках может вначале сбивать такая семантика этого оператора.

Примеры.

```
until false
do
    read x
    if [ $x = 5 ]
    then echo enough ; break
    else echo some more
    fi
done
```

Здесь программа с бесконечным циклом ждет ввода слов (повторяя на экране фразу "some more"), пока не будет введено "5". После этого выдается "enough" и команда "break" прекращает выполнение цикла.

Другой пример ("Ожидание полдня") иллюстрирует возможность использовать в условии вычисления.

```
until date | grep 12:00:
do
    sleep 30
done
```

Здесь каждые 30 секунд выполняется командная строка условия. Команда "date" выдает текущую дату и время. Команда "grep" получает эту информацию через конвейер и пытается совместить заданный шаблон "12:00:" с временем, выдаваемым командой "date". При несовпадении "grep" выдает код возврата "1", что соответствует значению "ложь", и цикл "выполняет ожидание" в течение 30 секунд, после чего повторяется выполнение условия. В полдень (возможно с несколькими секундами) произойдет сравнение, условие станет истинным, "grep" выдаст на экран соответствующую строку и работа цикла закончится.

5.7. Пустой оператор

Пустой оператор имеет формат

```
:
```

Ничего не делает. Возвращает значение "0". Например, в конструкции "while :" или ставить в начале командного файла, чтобы гарантировать, что файл не будет принят за выполняемый файл для "csh".

5.8. Функции в shell

Функция позволяет подготовить список команд shell для последующего выполнения.

Описание функции имеет вид:

```
имя ()
{
    список команд
}
```

после чего обращение к функции происходит по имени. При выполнении функции не создается нового процесса. Она выполняется в среде соответствующего процесса. Аргументы функции становятся ее позиционными параметрами; имя функции - ее нулевой параметр. Прервать выполнение функции можно оператором "return [n]", где (необязательное) "n" - код возврата.

Пример. Вызов на выполнение файла "fun"

```
echo $$
fn() # описание функции
{
    echo xx=$xx
    echo $#
    echo $0: $$ $1 $2
    xx=yy ; echo xx=$xx
```



```

return 5
}
xx=xx ; echo xx=$xx
fn a b          # вызов функции "fn" с параметрами
echo $?
echo xx=$xx

```

содержащего описание и вызов функции "fn", выдаст на экран:

```

749
xx=xx
xx=xx
2
fun: 749 a b
xx=yу
5
xx=yу

```

5.9. Обработка прерываний ("trap")

Бывает необходимо защитить выполнение программы от прерывания.

Наиболее часто приходится встречаться со следующими прерываниями, соответствующими сигналам:

- 0** выход из интерпретатора,
- 1** отбой (отключение удаленного абонента),
- 2** прерывание от ,
- 9** уничтожение (не перехватывается),
- 15** окончание выполнения.

Для защиты от прерываний существует команда "trap", имеющая формат:

```
trap 'список команд' сигналы
```

Если в системе возникнут прерывания, чьи сигналы перечислены через пробел в "сигналы", то будет выполнен "список команд", после чего (если в списке команд не была выполнена команда "exit") управление вернется в точку прерывания и продолжится выполнение командного файла.

Например, если перед прекращением по прерываниям выполнения какого то командного файла необходимо удалить файлы в "/tmp", то это может быть выполнено командой "trap":

```
trap 'rm /tmp/* ; exit 1' 1 2 15
```

которая предшествует прочим командам файла. Здесь, после удаления файлов будет осуществлен выход "exit" из командного файла.

Команда "trap" позволяет и просто игнорировать прерывания, если "список команд" пустой. Так например, если команда "cmd" выполняется очень долго, а пользователь

решил отключиться от системы, то для продолжения выполнения этой команды можно написать, запустив команду в фоновом режиме:

```
( trap '' 1; cmd ) &
```

Программирование на shell здесь описано достаточно полно, но далеко не исчерпывающе :-). Поэтому знакомство с литературой не только желательно, но и необходимо. Тем более, что описание даже стандартных команд (в силу очень больших объемов) здесь отсутствует.

Список литературы содержит некоторые из (быстро растущего перечня) книг на русском языке.

При подготовке материалов по программированию на shell прежде всего использованы книги [9, 1, 10, 11].

6. ЛИТЕРАТУРА

1. Кристиан К. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985. -318 с.
2. Готье Р. Руководство по операционной системе UNIX. -М.: Финансы и статистика, 1985. -232 с.
3. Браун П. Введение в операционную систему UNIX. -М.: Мир, 1987. -287 с.
4. Томас Р., Йейтс Дж. Операционная система UNIX. Руководство для пользователей. -М.: Радио и связь, 1986. -352 с.
5. Банахан М., Раттер Э. Введение в операционную систему UNIX. -М.: Радио и связь, 1986. -341 с.
6. Тихомиров В.П., Давидов М.И. Операционная система UNIX: Инструментальные средства программирования. -М.: Финансы и статистика, 1988. -206 с.
7. Баурн С. Операционная система UNIX. -М.: Мир, 1986. -462 с.
8. Беляков М.И. и др. Инструментальная мобильная операционная система ИНМОС. -М.: Финансы и статистика, 1985 -231 с.

9. Топхем Д., Чьюнг Х.В. Юникс и Ксеникс. -М.: Мир, 1988. -392 с.
10. Беляков М.И., Рабовер Ю.И., Фридман А.Л. Мобильная операционная система. -М.: Радио и связь, 1991 -208 с.
11. Керниган Б.В., Пайк Р. UNIX - Универсальная среда программирования. -М.: Финансы и статистика, 1992 -304 с.