Лабораторная работа № 8. Возможности командной оболочки

Редактирование ввода

Уже некоторое время поработав в Linux, понабирав команды в командной строке, Мефодий пришёл к выводу, что в общении с оболочкой не помешают кое-какие удобства. Одно из таких удобств -- возможность редактировать вводимую строку с помощью клавиши `Backspace` (удаление последнего символа), "`^W`" (удаление слова) и "`^U`" (удаление всей строки) -- предоставляет сам терминал Linux. Эти команды работают для *любого* построчного ввода: например, если запустить программу `cat` без параметров, чтобы та немедленно отображала вводимые с терминала строки. Если по каким-то причинам в строчку на экране влез мусор, можно нажать "`^R`" (redraw) -- система выведет в новой строке содержимое входного буфера.

Мефодий не забыл, что `cat` без параметров следует завершать командой "`^D`" (конец ввода). Эту команду, как и предыдущие, интерпретирует при вводе с терминала система. Система же превращает некоторые другие управляющие символы (например, "`^C`" или "`^Z`") в **сигналы**. В действительности *все* управляющие символы, интерпретируемые системой, можно перенастроить с помощью команды `stty`. Полный список того, что можно настраивать, выдаёт команда `stty -a`:

```
[methody@localhost methody]$ stty -a
 localhost 38400 baud; rows 30; columns 80; line = 0;
 intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
 eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
 lnext = ^V; flush = ^O; min = 1; time = 0;
 -parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
 -ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
 -iuclc -ixany -imaxbel -iutf8
 opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
 echoctl echoke
```

Настройки терминальной линии

При виде столь обширных возможностей Мефодий немедленно взялся читать руководство (`man stty`), однако нашёл в нём не так уж много для себя полезного. Из управляющих символов (строки со второй по четвёртую) интересны "`^S`" и "`^Q`", с помощью которых можно, соответственно, приостановить и возобновить выдачу на терминал (если текста вывелось уже много, а прочесть его не успеваешь). Можно заметить, что настройка `erase` (удаление одного символа) соответствует управляющему символу, который возвращается клавишей `Backspace` именно виртуальной консоли Linux -- "`^?`". На многих терминалах клавиша `Backspace` возвращает *другой* символ -- "`^H`". Если необходимо *переопределить* настройку `erase`, можно воспользоваться командой "`stty erase ^H`", причём "`^H`" (для удобства) разрешено вводить и как *два* символа: "`^`" и "`H`".

Наконец, чтобы *лишить* передаваемый символ его управляющих функций (если, например, требуется передать программе на ввод *символ* с кодом `3`, т. е. "`^C`"), непосредственно перед вводом этого символа нужно подать команду "`^V`" (`lnext`):

```
[methody@localhost methody]$ cat | hexdump -C
 Сейчас нажмём Ctrl+C
[methody@localhost methody]$ cat | hexdump -C
 Теперь Ctrl+V, Ctrl+C, enter и Ctrl+D^C
 00000000  f4 c5 d0 c5 d2 d8 20 43  74 72 6c 2b 56 2c 20 43  |Теперь Ctrl+V, C|
 00000010  74 72 6c 2b 43 2c 20 45  6e 74 65 72 20 c9 20 43  |trl+C, enter и C|
 00000020  74 72 6c 2b 44 03 0a                              |trl+D..|
 00000027
```

Экранирование управляющих символов

Здесь Мефодий прервал, как и собирался, работу первого из `cat`. При этом до `hexdump`, фильтра, переводящего входной поток в шестнадцатеричное представление, дело даже не дошло, потому что `cat` не успел обработать ни одной строки. Во втором случае "`^C`" после "`^V`" потеряло управляющий смысл и отобразилось при вводе. С ключом "`-C`" `hexdump` выводит также и текстовое представление входного потока, заменяя непечатные символы точками. Так на точки были заменены и "`^C`" (ASCII-код `03`), и возвращаемый `Enter` символ конца строки (ASCII-код `0a`, в десятичном виде -- `10`). Ни "`^V`", ни "`^D`" на вход `hexdump` , конечно, не попали: их, как управляющие, обработала система.

Прочие настройки `stty` относятся к обработке текста при выводе на терминал и вводе с него. Они интересны только в том смысле, что при их изменении работать с командной оболочкой становится неудобно. Например, настройка `echo` определяет, будет ли *система* отображать на экране всё, что вводит пользователь. При включённом `echo` нажатие любой алфавитно-цифровой клавиши (ввод символа) приводит к тому, что система (устройство типа `tty`) *выведет* этот символ на терминал. Настройка отключается, когда с клавиатуры вводится пароль. При этом трудно отделаться от ощущения, что ввода с клавиатуры не происходит. Ещё хуже обстоит дело с настройками, состоящими из кусков вида "`i`", "`o`", "`cr`" и "`nl`". Эти настройки управляют преобразованием при вводе и выводе исторически сложившегося обозначения конца строки *двумя* символами в *один*, принятый в Linux. Может случиться так, что клавиша `Enter` терминала возвращает как раз неправильный символ конца строки, а преобразование отключено. Тогда вместо `Enter` следует использовать "`^J`" -- символ, *на самом деле* соответствующий концу строки.

Во всех случаях, когда терминал находится в непонятном состоянии -- не реагирует на `Enter`, не показывает ввода, не удаляет символов, выводит текст "ступеньками" и т. п., рекомендуется "лечить" настройки терминала с помощью `stty sane` -- специальной формы `stty`, сбрасывающей настройки терминала в некоторое пригодное к работе состояние. Если непонятное состояние терминала возникло однократно, например, после аварийного завершения экранной программы (редактора `vim` или оболочки `mc`), то можно воспользоваться командой `reset`. Она заново настраивает терминал в *полном* соответствии с системной конфигурацией (указанной в файле `/etc/inittab`) и `terminfo`.

Если терминал ведёт себя странно, последовательность "^J stty sane^J" может его вылечить!

Редактирование командной строки

Даже не изучая специально возможностей командной оболочки, Мефодий активно использовал некоторые из них, не доступные при вводе текста *большинству* утилит (в частности, ни `cat`, ни `hexdump`). Речь идёт о клавишах `Стрелка влево` и `Стрелка вправо`, с помощью которых можно перемещать курсор по командной строке, и клавише `Del`, удаляющей символ *под* курсором, а не позади него. На предыдущих занятиях он уже убедился, что эти команды работают в `bash`, но не работают для `cat`. Более того, для простого командного интерпретатора -- `sh` -- они тоже не работают.

Следовательно, возможности редактора командной строки специфичны для разных командных оболочек. Однако самые необходимые команды редактирования поддерживаются во всех разновидностях shell сходным образом. По словам Гуревича "во всех видах Linux обязательно есть `bash`, а если ты достаточно опытен, чтобы устанавливать и настраивать пакеты, можешь установить `zsh`, у него возможностей больше, чем может понадобиться одному человеку". Поэтому Мефодий занялся изучением документации по `bash`, что оказалось делом непростым, ибо в `bash.info` он насчитал более восьми с половиной тысяч строк. Даже про редактирование командной строки написано столько, что за один раз прочесть трудно.

Попытка "наскоком" узнать *всё* про работу в командной строке принесла некоторую пользу. Во-первых, перемещаться в командной строке можно не только по одному символу вперёд и назад, но и по словам: команды `ESCF`/`ESCB` или `Alt+F`/`Alt+B` соответственно (от forward и bckward), работают также клавиши `&home&` и `&end&`, или, что то же самое, "`^A`" и "`^E`". А во-вторых, помимо работы с *одной* командной строкой, существует ещё немало других удобств, о которых и пойдёт речь на этом занятии.

История команд

Двумя другими клавишами со стрелками -- вверх и вниз -- Мефодий тоже активно пользовался, не подозревая, что задействует этим весьма мощный механизм `bash` -- работу с **историей команд**. Все команды, набранные пользователем, `bash` запоминает и позволяет обращаться к ним впоследствии. По стрелке вверх (можно использовать и "`^P`", previous), список поданных команд "прокручивается" от последней к первой, а по стрелке вниз ("`^N`", next) -- обратно. Соответствующая команда отображается в командной строке как только что набранная, её можно отредактировать и подать оболочке (подгонять курсор к концу строки при этом не обязательно).

Если необходимо добыть из истории какую-то давнюю команду, проще не гонять список истории стрелками, а *поискать* в ней с помощью команды "^R" (reverse search). При этом выводится подсказка специального вида ("(reverse-i-search)"), подстрока поиска (окружённая символами ` и ') и последняя из команд в истории, в которой эта подстрока присутствует:

```
[methody@localhost methody]$
 ^R | (reverse-i-search)`': 
 i  | (reverse-i-search)`i': ls i
 n  | (reverse-i-search)`in': info
 f  | (reverse-i-search)`inf': info
 o  | (reverse-i-search)`info': info
 ^R | (reverse-i-search)`info': man info
 ^R | (reverse-i-search)`info': info "(bash.info.bz2)Commands For History"
```

Поиск по истории команд

Пример представляет символы вводимые Мефодием (в левой части до "|") и содержимое последней строки терминала. Это "кадры" работы с одной и той же строкой, показывающие, как она меняется при наборе. Набрав "info", Мефодий продолжил поиск этой подстроки, повторяя "^R" до тех пор, пока не наткнулся на нужную ему команду, содержащую подстроку "info". Осталось только передать её bash с помощью Enter.

Чтобы история команд могла сохраняться *между* сеансами работы пользователя, bash записывает её в файл .bash_history, находящийся в домашнем каталоге пользователя. Делается это в момент *завершения* оболочки: накопленная за время работы история дописывается в конец этого файла. При следующем запуске bash считывает .bash_history целиком. История хранится не вечно, количество запоминаемых команд в .bash_history ограничено (обычно 500 командами, но это можно и перенастроить).

Сокращения

Поиск по истории -- удобное средство: длинную командную строку можно не набирать целиком, а выискать и использовать. Однако *давнюю* команду придётся добывать с помощью нескольких "^R" -- а можно и совсем не доискаться, если она уже выбыла оттуда. Для того, чтобы оперативно заменять короткие команды длинными, стоит воспользоваться **сокращениями** (aliases). В конфигурационных файлах командного интерпретатора пользователя обычно *уже* определено несколько сокращений, список которых можно посмотреть с помощью команды alias без параметров:

```
[methody@localhost methody]$ alias
 alias cd..='cd ..'
 alias cp='cp -i'
 alias l='ls -lapt'
 alias ll='ls -laptc'
 alias ls='ls --color=auto'
 alias md='mkdir'
 alias mv='mv -i'
 alias rd='rmdir'
 alias rm='rm -i'
```

Просмотр заранее определённых сокращений

С сокращениями Мефодий уже сталкивался на предыдущих занятиях, где команда `ls` отказалась работать в согласии с теорией. Выяснилось, что по команде `ls` вместо *утилиты* `/bin/ls bash` запускает собственную команду-сокращение, превращающееся в команду `ls --color=auto`. *Повторно* появившуюся в команде подстроку "`ls`" интерпретатор уже не обрабатывает, во избежание вечного цикла. Например, команда `ls -al` превращается в результате в `ls --color=auto -al`. Точно так же любая команда, начинающаяся с `rm`, превращается в `rm -i` (interactive), что Мефодия крайне раздражает, потому что ни одно удаление не обходится без вопросов в стиле "rm: удалить обычный файл `файл'?".

```
[methody@localhost methody]$ unalias cp rm mv
[methody@localhost methody]$ alias pd=pushd
[methody@localhost methody]$ alias pp=popd
[methody@localhost methody]$ pd /bin
 /bin ~
[methody@localhost bin]$ pd /usr/share/doc
 /usr/share/doc /bin ~
[methody@localhost doc]$ cd /var/tmp
[methody@localhost tmp]$ dirs
 /var/tmp /bin ~
[methody@localhost tmp]$ pp
 /bin ~
[methody@localhost bin]$ pp
 ~
[methody@localhost methody]$ pp
 -bash: popd: directory stack empty
```

Использование сокращений и pushd/popd

От надоедливого "`-i`" Мефодий избавился с помощью команды `unalias`, а заодно ввёл сокращения для полюбившихся ему команд `bash` -- `pushd` и `popd`. Эти команды[1], подобно `cd`, меняют текущий каталог. Разница состоит в том, что `pushd` все каталоги, которые пользователь делает текущими, запоминает в особом списке (стеке). Команда `popd` удаляет последний элемент этого стека, и делает текущим каталогом предпоследний. Обе команды вдобавок выводят содержимое стека каталогов (то же самое делает и команда `dirs`). Команда `cd` в `bash` также работает со стеком каталогов: она *заменяет* его последний элемент новым.

[1] Они названы по аналогии с операциями работы со стеком -- push и pop

команда-сокращение	Внутренняя команда shell, задаваемая пользователем. Обычно заменяет одну более длинную команду, которая часто используется при работе в командной строке. Сокращения не наследуются с окружением.

Достраивание

Сокращения позволяют быстро набирать *команды*, однако никак не затрагивают имён *файлов*, которые чаще всего и оказываются параметрами этих команд. Бывает, что набранной строки -- пути к файлу и нескольких первых букв его имени -- достаточно для *однозначного* указания на этот файл, потому что по введённому пути больше файлов, чьё имя начинается на эти буквы, просто нет. Чтобы не дописывать оставшиеся буквы (а имена файлов в Linux могут быть весьма длинными), Гуревич посоветовал Мефодию нажать клавишу `Tab`. И -- о чудо! -- `bash` *сам* достроил начало имени файла до целого (снова воспользуемся методом "кадров"):

```
[methody@localhost methody]$ ls -al /bin/base
 Tab    | [methody@localhost methody]$ ls -al /bin/basename
-rwxr-xr-x  1 root root 12520 Июн  3 18:29 /bin/basename
[methody@localhost methody]$ base
 Tab    | [methody@localhost methody]$ basename
 Tab    | [methody@localhost methody]$ basename ex
 Tab    | [methody@localhost methody]$ basename examples/
 Tab    | [methody@localhost methody]$ basename examples/-filename-with-
-filename-with-
```

Использование достраивания

Дальше -- больше. Оказывается, и имя команды можно вводить не целиком: оболочка догадается достроить набираемое слово именно до команды, раз уж это слово стоит в начале командной строки. Таким образом, команду `basename examples/-filename-with-` Мефодий набрал за *восемь* нажатий на клавиатуру ("base" и четыре `Tab`)! Ему не пришлось вводить начало имени файла в каталоге `examples`, потому что файл там был всего один.

Выполняя **достраивание** (completion), `bash` может вывести не всю строку, а только ту её часть, относительно которой у него нет сомнений. Если дальнейшее достраивание может пойти *несколькими* путями, то однократное нажатие `Tab` приведёт к тому, что `bash` растерянно пискнет[2], а повторное -- к выводу *под* командной строкой списка всех возможных вариантов. В этом случае надо подсказать командной оболочке продолжение: дописать несколько символов, определяющих, по какому пути пойдёт достраивание, и снова нажать `Tab`.

Поиск ключевого слова "completion" по документации `bash` выдал так много информации, что Мефодий обратился к Гуревичу за помощью. Однако

[2] Все терминалы должны уметь выдавать звуковой сигнал при *выводе* управляющего символа "`^G`". Для этого не нужно запускать никаких дополнительных программ: "настоящие" терминалы имеют встроенный динамик, а виртуальные консоли обычно пользуются системным ("пищалкой"). В крайнем случае разрешается привлекать внимание пользователя другими способами: например, эмулятор терминала `screen` пишет в служебной строке "wuff-wuff" ("гав-гав").

тот ответил, что не использует `bash`, и поэтому не в состоянии объяснять тонкости его настройки. Если в `bash` -- *несколько* типов достраивания (по именам файлов, по именам команд и т. п.), то в `zsh` их *сколько угодно*: существует способ запрограммировать любой алгоритм достраивания и задать шаблон командной строки, в которой именно этот способ будет применяться.

Генерация имён файлов

Достраивание очень удобно, когда цель пользователя -- задать *один* конкретный файл в командной строке. Если же нужно работать сразу с *несколькими* файлами -- например для перемещения их в другой каталог с помощью `mv`, достраивание не помогает. Необходим способ задать одно "общее" имя для группы файлов, с которыми будет работать команда. В подавляющем большинстве случаев это можно сделать при помощи **шаблона**.

Шаблоны

Шаблон в командном интерпретаторе используется примерно в тех же целях, что и **регулярное выражение** - для поиска строк определённой структуры среди множества разнообразных строк. В отличие от регулярного выражения, шаблон всегда примеряется к строке целиком, кроме того, он устроен значительно проще (а значит, и беднее).

Символы в шаблоне разделяются на обычные и **специальные**. Обычные символы соответствуют таким же символам в строке, а специальные обрабатываются особым образом:

- Шаблону, состоящему только из обычных символов, соответствует *единственная* строка, состоящая из тех же символов в том же порядке. Например, шаблону "`abc`" соответствует строка `abc`, но не `aBc` или `ABC`, потому что большие и маленькие буквы различаются.
- Шаблону, состоящему из единственного спецсимвола "`*`", соответствует *любая* строка любой длины (в том числе и пустая).
- Шаблону, состоящему из единственного спецсимвола "`?`", соответствует *любая* строка длиной в *один* символ, например, `a`, `+` или `@`, но не `ab` или `8888`.
- Шаблону, состоящему из любых символов, заключённых в квадратные скобки "`[`" и "`]`" соответствует строка длиной в *один* символ, причём этот символ должен *встречаться* среди заключённых в скобки. Например, шаблону "`[bar]`" соответствуют только строки `a`, `b` и `r`, но не `c`, `B`, `bar` или `ab`. Символы внутри скобок можно не перечислять полностью, а задавать *диапазон*, в начале которого стоит символ с наименьшим ASCII-кодом, затем следует "`-`", а затем -- символ с наибольшим ASCII-кодом. Например, шаблону "`[0-9a-fA-F]`" соответствует одна шестнадцатеричная цифра (скажем, `5`, `e` или `C`). Если после "`[`" в шаблоне следует "`!`", то ему соответствует строка из одного символа *не* перечисленного между скобками.

- Шаблону, состоящему из *нескольких* частей, соответствует строка, которую можно разбить на столько же подстрок (возможно, пустых), причём первая подстрока будет отвечать первой части шаблона, вторая -- второй части и т. д. Например, шаблону "`a*b?c`" будут соответствовать строки `ab@c` ("`*`" соответствует пустая подстрока), `a+b=c` и `aaabbc`, но не соответствовать `abc` ("`?`" соответствует подстрока `c`, а для "`c`" соответствия не находится), `@ab@c` (нет соответствия для "`a`") или `aaabbbc` (из трёх `b` первое соответствует "`b`", второе -- "`?`", а вот третье приходится на "`c`").

шаблон	Строка специального формата, используемая в процедурах текстового поиска. Говорят, что строка **соответствует** шаблону, если можно по определённым правилам каждому символу строки поставить в соответствие символ шаблона. В Linux наиболее популярны шаблоны в формате командного интерпретатора и **регулярные выражения**.

Использование шаблонов

Шаблоны используются в нескольких конструкциях shell. Главное место их применения -- командная строка. Если оболочка видит в командной строке шаблон, она немедленно заменяет его на список *файлов*, имена которых ему соответствуют. Команда, которая затем вызывается, получает в качестве параметров список файлов уже безо всяких шаблонов, как если бы этот список пользователь ввёл вручную. Эта способность командного интерпретатора называется **генерацией имён файлов**.

```
[methody@localhost methody]$ ls .bash*
 .bash history  .bash logout  .bash profile  .bashrc
[methody@localhost methody]$ /bin/e*
 /bin/ed /bin/egrep /bin/ex
[methody@localhost methody]$ ls *a*
 -filename-with-
[methody@localhost methody]$ ls -dF *[ao]*
 Documents/  examples/  loop  to.sort*
```

Использование шаблона в командной строке

Мефодий, как это случается с новичками, немедленно натолкнулся на несколько "подводных камней", неизбежных в ситуации, когда конечный результат неизвестен. Только первая команда сработала *не* вопреки его ожиданиям: шаблон "`.bash*`" был превращён командной оболочкой в список файлов, начинающихся на `.bash`, этот список получила в качестве параметров командной строки `ls`, после чего честно его вывела. С "`/bin/e*`" Мефодию повезло -- этот шаблон превратился в список файлов из каталога `/bin`, начинающихся на "`e`", и *первым* файлом в списке оказалась безобидная утилита `/bin/echo`. Поскольку в командной строке ничего, кроме шаблона, не было, именно строка `/bin/echo` была воспринята оболочкой в качестве *команды*[3], которой -- в качестве *параметров* -- были переданы *остальные* элементы списка -- `/bin/ed`, `/bin/egrep` и `/bin/ex`.

[3] Можно вспомнить про *нулевой* параметр командной строки, обсуждавшийся на предыдущих занятиях

Что же касается `ls *a*`, то, по расчётам Мефодия, эта команда должна была выдать список файлов в текущем каталоге, имя которых *содержит* "`a`". Вместо этого на экран вывелось имя файла из подкаталога `examples`... Впрочем, никакой чёрной магии тут нет. Во-первых, имена файлов вида "`.bash*`" хотя и содержат "`a`", но начинаются на точку, и, стало быть, считаются **скрытыми**. Скрытые файлы попадают в результат генерации имён только если точка в начале указана *явно* (как в первой команде примера). Поэтому по шаблону "`*a*`" в домашнем каталоге Мефодия `bash` нашёл только подкаталог с именем `examples`, его-то он и передал в качестве параметра утилите `ls`. Что вывелось на экран в результате образовавшейся команды `ls examples`? Конечно, содержимое каталога. Шаблон в последней команде из примера, "`*[ao]*`", был превращён в список файлов, чьи имена содержат "`a`" или "`o`" -- `Documents examples loop to.sort`, а ключ "`-d`" потребовал у `ls` показывать информацию о каталогах, а не об их содержимом. В соответствии с ключом "`-F`", `ls` расставил "`/`" после каталогов и "`*`" после исполняемых файлов.

Ещё одно отличие генерации имён от стандартной обработки шаблона -- в том, что символ "`/`", разделяющий элементы пути, *никогда* не ставится в соответствие "`*`" или диапазону. Происходит это не потому, что искажён алгоритм, а потому, что при генерации имён шаблон применяется именно к элементу пути, внутри которого уже нет "`/`". Например, получить список файлов, которые находятся в каталогах `/usr/bin` и `/usr/sbin` и содержат подстроку "`ppp`" в имени, можно с помощью шаблона "`/usr/*bin/*ppp*`". Однако *одного* шаблона, который бы включал в этот список ещё и каталоги `/bin` и `/sbin` -- то есть подкаталоги *другого* уровня вложенности -- по стандартным правилам сделать нельзя[4].

Если перед любым специальным символом стоит "`\`", этот символ лишается специального значения, **экранируется**: пара "`\`*символ*" заменяется командным интерпретатором на "*символ*" и передаётся в командную строку безо всякой дальнейшей обработки:

```
[methody@localhost methody]$ echo *o*
 Documents loop to.sort
[methody@localhost methody]$ echo \*o\*
 *o*
[methody@localhost methody]$ echo "*o*"
 *o*
[methody@localhost methody]$ echo *y*
 *y*
[methody@localhost methody]$ ls *y*
 ls: *y*: No such file or directory
```

Экранирование специальных символов и обработка "пустых" шаблонов

[4] Генерация имён файлов в `zsh` предусматривает специальный шаблон "`**`", которому соответствуют подстроки с *любым* количеством "`/`". Пользоваться им следует *крайне осторожно*, понимая, что при генерации имён по такому шаблону выполняется операция, аналогичная не `ls`, а `ls -R` или `find`. Так, использование "`/**`" в начале шаблона вызовет просмотр *всей* файловой системы!

Мефодий заметил, что шаблон, которому не соответствует *ни одного* имени файла, `bash` раскрывать не стал, как если бы все "`*`" в нём были экранированы. В самом деле, какое из двух зол меньшее: изменять *интерпретацию* спецсимволов в зависимости от содержимого каталога, или сохранять логику интерпретации с риском превратить команду с параметрами в команду *без параметров*? Если бы, допустим, шаблон, которому не нашлось соответствия, попросту удалялся, то команда `ls *y*` превратилась бы в `ls` и неожиданно выдала бы содержимое всего каталога. Авторы `bash` (как и Стивен Борн, автор самой первой командной оболочки -- `sh`) выбрали более непоследовательный, но и более безопасный первый способ[5].

Лишить специальные символы их специального значения можно и другим способом. На предыдущих занятиях было рассказано, что разделители (пробелы, символы табуляции и символы перевода строки) перестают восприниматься таковыми, если часть командной строки, их содержащую, окружить двойными или одинарными кавычками. В кавычках перестаёт "работать" и генерация имён (как это видно из примера), и интерпретация других специальных символов. Двойные кавычки, однако, допускают выполнение **подстановок** переменной окружения и результата работы команды, описанных в двух следующих разделах.

Окружение

На предыдущих занятиях уже упоминалось, что системный вызов `fork()`, создавая точную копию родительского процесса, копирует также и **окружение**. Необходимость в "окружении" происходит вот из какой задачи. Акт передачи данных от *этого конкретно* родительского процесса дочернему, и, что ещё важнее, системе, должен обладать свойством **атомарности** (т.е. операции выступают вместе как *неделимая* единица работы). Если использовать для этой цели *файл* (например, конфигурационный файл запускаемой программы), всегда сохраняется вероятность, что за время между изменением файла и последующим чтением запущенной программой кто-то -- например, другой процесс того же пользователя -- снова изменит этот файл[6]. Хорошо бы, чтобы *изменение* данных и их *передача* делались бы *одной* операцией. Например, завести для каждого процесса такой "файл", содержимое которого, во-первых, мог бы изменить *только* этот процесс, и, во-вторых, оно автоматически копировалось бы в аналогичный "файл" дочернего процесса при его порождении.

Эти свойства и реализованы в понятии "**окружение**". Каждый запускаемый процесс система снабжает неким информационным пространством, которое этот процесс вправе изменять как ему

---

[5] Авторы `zsh` пошли по другому пути: в этой версии shell использование шаблона, которому не соответствует ни одно имя файла, приводит к *ошибке*, и соответствующая команда не выполняется.

[6] Эта ситуация называется "race condition" ("состояние гонки"), и часто встречается в плохо спроектированных системах, где есть хотя бы *два* параллельных процесса.

заблагорассудится. Правила пользования этим пространством просты: в нём можно задавать именованные хранилища данных (**переменные окружения**), в которые записывать какую угодно информацию (присваивать значение переменной окружения), а впоследствии эту информацию считывать (подставлять значение переменной). Дочерний процесс -- точная копия родительского, поэтому его окружение -- также точная копия родительского. Если про дочерний процесс известно, что он использует значения некоторых переменных из числа передаваемых ему с окружением, родительский может заранее указать, каким из копируемых в окружении переменных нужно изменить значение. При этом, с одной стороны, никто (кроме системы, конечно) не сможет вмешаться в процесс передачи данных, а с другой стороны, одна и та же утилита может быть использована *одним и тем же* способом, но в изменённом окружении -- и выдавать различные результаты:

```
[methody@localhost methody]$ date
 Птн Ноя  5 16:20:16 MSK 2004
[methody@localhost methody]$ LC_TIME=C date
 Fri Nov  5 16:20:23 MSK 2004
```

Утилита `date` пользуется переменной окружения `LC_TIME`	
окружение	Набор данных, приписанных системой процессу. Процесс может пользоваться информацией из окружения для настройки, изменять и дополнять его. Окружение представлено в виде **переменных окружения** и их значений. При порождении процесса окружение родительского процесса **наследуется** дочерним (копируется).

Работа с переменными в shell

В последнем примере Мефодий воспользовался подсмотренным у Гуревича приёмом: присвоил некоторое значение переменной окружения в командной строке *перед* именем команды. Командный интерпретатор, увидев "=" внутри первого слова командной строки, приходит к выводу, что это -- операция присваивания, а не имя команды, и запоминает, как надо изменить окружение команды, которая последует после. Переменная окружения `LC_TIME` предписывает использовать определённый язык при выводе даты и времени а значение "`C`" соответствует "стандартному системному" языку (чаще всего -- английскому).

Если рассматривать shell в качестве высокоуровневого языка программирования, его переменные -- самые обычные *строковые* переменные. Записать значение в переменную можно с помощью операции присваивания, а прочесть его оттуда -- с помощью операции **подстановки** вида `$`*`переменная`*:

```
[methody@localhost methody]$ A=dit
[methody@localhost methody]$ C=dah
[methody@localhost methody]$ echo $A $B $C
 dit dah
[methody@localhost methody]$ B=" "
[methody@localhost methody]$ echo $A $B $C
 dit dah
[methody@localhost methody]$ echo "$A $B $C"
 dit   dah
[methody@localhost methody]$ echo '$A $B $C'
 $A $B $C
```

Подстановка значений переменных

Как видно из примера, значение *неопределённой* переменной (`B`) в shell считается пустым и при подстановке не выводится никаких предупреждений. Сама подстановка происходит, как и генерация имён, *перед* разбором командной строки, набранной пользователем. Поэтому вторая команда `echo` в примере получила, как и первая *два* параметра ("`dit`" и "`dah`"), несмотря на то, что переменная `B` была к тому времени определена и содержала **разделитель-пробел**. А вот третья и четвёртая команды `echo` получили по *одному* параметру. Здесь сказалось различие между одинарными и двойными кавычками в shell: внутри двойных кавычек действует подстановка значений переменных.

Переменные, которые командный интерпретатор `bash` определяет *после* запуска, не принадлежат окружению, и, стало быть, не наследуются дочерними процессами. Чтобы переменная `bash` попала в окружение, её надо *проэкспортировать* командой `export`:

```
[methody@localhost methody]$ echo "$Qwe -- $LANG"
  -- ru RU.KOI8-R
[methody@localhost methody]$ Qwe="Rty" LANG=C
[methody@localhost methody]$ echo "$Qwe -- $LANG"
 Rty -- C
[methody@localhost methody]$ sh
sh-2.05b$ echo "$Qwe -- $LANG"
  -- C
sh-2.05b$ exit
[methody@localhost methody]$ echo "$Qwe -- $LANG"
 Rty -- C
[methody@localhost methody]$ export Qwe
[methody@localhost methody]$ sh
sh-2.05b$ echo "$Qwe -- $LANG"
 Rty -- C
sh-2.05b$ exit
```

Экспорт переменных shell в окружение

Здесь Мефодий завёл *новую* переменную `Qwe` и изменил значение переменной окружения `LANG`, доставшейся стартовому `bash` от программы `login`. В результате запущенный дочерний процесс `sh` получил *изменённое* значение `LANG` и никакой переменной `Qwe` в окружении. После `export Qwe` эта переменная была добавлена в окружение и, соответственно, передалась `sh`.

Переменные окружения, используемые системой и командным интерпретатором

Во время сеанса работы пользователя стартовый командный интерпретатор получает от `login` довольно богатое окружение, к которому добавляет и собственные настройки. Просмотреть окружение в `bash` можно с помощью команды `set`. Большинство заранее определённых переменных используются либо самой командной оболочкой, либо утилитами системы, поэтому их изменение приводит к тому, что оболочка или утилиты начинают работать слегка иначе.

Весьма примечательна переменная окружения `PATH`. В ней содержится список каталогов, элементы которого разделяются двоеточиями. Если команда в командной строке -- не собственная команда shell (вроде `cd`) и не представлена в виде *пути* к запускаемому файлу (как `/bin/ls` или `./script`), то shell будет искать эту команду среди имён запускаемых файлов во всех каталогах `PATH`, и только в них. Точно так же будут поступать и другие утилиты, использующие **библиотечную функцию** `execlp()` **или** `execvp()` (запуск программы).

По этой причине исполняемые файлы невозможно запускать просто по имени, если они лежат в текущем каталоге, и текущий каталог не входит в `PATH`. Мефодий в таких случаях пользовался кратчайшим из возможных путей, "`./`" (например, вызывая сценарий `./script`).

```
[methody@localhost methody]$ echo $PATH
 /home/methody/bin:/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/usr/games
[methody@localhost methody]$ mkdir /home/methody/bin
[methody@localhost methody]$ mv to.sort loop script bin/
[methody@localhost methody]$ script
 Hello, Methody!
```

Использование `PATH`

Пути, указанные в `PATH` не обязаны существовать на самом деле. Обнаружив в `$PATH` элемент `/home/methody/bin` (подкаталог `bin` домашнего каталога), Мефодий решил, что гораздо правильнее будет складывать исполняемые файлы не куда попало, а именно в этот каталог: во-первых, это упорядочит структуру домашнего каталога, а в во-вторых, позволит запускать эти файлы по имени. Но для начала пришлось этот каталог *создать*. Порядок, при котором собственные программы лежат в специальном каталоге, куда *безопаснее* беспорядка, при котором поиск запускаемого файла по имени начинается с *текущего* каталога. Если в текущем каталоге окажется программа `ls`, и этот каталог -- не `/bin`, а, допустим, `/home/shogun/dontrunme`, стоит ожидать подвоха...

Переменных окружения, влияющих на работу *разных* утилит, довольно много. Например, переменные семейства `LC_` (полный их список выдаётся командой `locale`), определяющие язык, на котором выводятся диагностические сообщения, стандарты на формат даты, денежных единиц, чисел, способы преобразования строк и т. п. Очень важна переменная `TERM`,

определяющая *тип* терминала: как известно разные терминалы имеют различные управляющие последовательности, поэтому программы, желающие эти последовательности использовать, обязательно сверяются с переменной `TERM`[7]. Если какая-то утилита требует редактирования файла, этот файл передаётся программе, путь к которой хранится в переменной `EDITOR` (обычно это `/usr/bin/vi`, о котором речь пойдёт далее). Наконец, некоторые переменные вроде `UID`, `USER` или `PWD` просто содержат полезную информацию, которую можно было бы добыть и другими способами.

Некоторые переменные окружения предназначены специально для `bash`: они задают его свойства и особенности поведения. Таковы переменные семейства `PS` (Prompt String). В этих переменных хранится строка-подсказка, которую командный интерпретатор выводит в разных состояниях. В частности, содержимое `PS1` -- это подсказка, которую shell показывает, когда вводит командную строку, а `PS2` -- когда пользователь нажимает Enter, а интерпретатор по какой-то причине считает, что ввод командной строки не завершён (например, не закрыты кавычки). С `$PS2` (символом ">") Мефодий уже сталкивался на предыдущих занятиях.

```
[methody@localhost methody]$ cd examples/
[methody@localhost examples]$ echo $PS1
 [\u@\h \W]\$
[methody@localhost examples]$ PS1="--> "
-->
-->
PS1="\t \w "
22:11:47 ~
22:11:48 ~
22:11:48 ~ PS1="\u@\h:\w \$ "
methody@localhost:~/examples $
methody@localhost:~/examples $
methody@localhost:~/examples $ cd
methody@localhost:~ $
methody@localhost:~ $
```

Использование переменной окружения `PS1`

Мефодий экспериментировал с `PS1`, параллельно читая документацию по `bash` ("`(bash.info)Printing a Prompt`"). Оказывается, в этом командном интерпретаторе содержимое `PS1` не просто подставляется при выводе, оно ещё и преобразуется, позволяя выводить всякую полезную информацию: имя пользователя (соответствует подстроке "`\u`", user), имя компьютера ("`\h`", host), время ("`\t`", time), путь к текущему каталогу ("`\w`", work directory) и т. п. В примере Мефодий заменил в конце концов "`\W`" (показывающую *последний* элемент пути, то есть собственное имя текущего каталога) на "`\w`", *полный* путь, потому что "`\w`" обладает свойством выделять в полном пути домашний каталог и заменять его на "~". Такое преобразование значений переменных

[7] В действительности такие программы обычно используют библиотеку `curses`, оперируя *не зависящими* от типа терминала понятиями (вроде "очистка экрана" или "позиционирование курсора", а процедуры из `curses` преобразуют их в управляющие последовательности конкретного терминала, сверившись сначала с `$TERM`, а затем -- с содержимым соответствующего раздела *базы данных по терминалам*, которая называется `terminfo`.

семейства `PS1` происходит только когда их использует `bash` в качестве подсказки, а при обычной подстановке этого не происходит.

Язык программирования sh

Некогда Мефодий выучил несколько языков программирования, и уже собрался было написать кое-какие нужные программы на Си или Python, однако Гуревич его остановил. Большая часть того, что нужно начинающему пользователю Linux, делается с помощью одной правильной команды, или вызовом нескольких команд в конвейере. От пользователя только требуется оформить решение задачи в виде **сценария** на shell. На самом же деле уже самый первый из командных интерпретаторов, `sh`, был настоящим высокоуровневым языком программирования -- если, конечно, считать все утилиты системы его операторами. При таком подходе от `sh` требуется совсем немного: возможность вызывать утилиты, возможность свободно манипулировать результатом их работы и несколько алгоритмических конструкций (условия и циклы).

К сожалению, *программирование* на shell, а также и других, более мощных интерпретируемых языках в Linux, не помещается в рамки нашего курса. Так что, пока Мефодий читает документацию по `bash` и самозабвенно упражняется в написании сценариев, нам остаётся только поверхностно рассмотреть свойства shell как языка программирования и *интегратора* команд. Заметим попутно, что писать сценарии для `bash` -- непрактично, так как исполняться они смогут лишь при помощи `bash`. Если же ограничить себя рамками `sh`, совместимость с которым объявлена и в `bash`, и в `zsh`, и в `ash` (наиболее близком к `sh` по возможностям), и в других командных интерпретаторах, выполняться эти сценарии смогут любым из `sh`-подобных интерпретаторов, и не только в Linux.

Интеграция процессов

Каждый процесс Linux при завершении передаёт родительскому **код возврата** (exit status), который равен нулю, если процесс считает, что его работа была успешной, или *номеру ошибки* в противном случае. Командный интерпретатор хранит код возврата последней команды в специальной переменной "`?`". Что более важно, код возврата используется в условных операторах: если он равен нулю, условие считается выполненным, а если нет -- невыполненным:

```
[methody@localhost methody]$ grep Methody bin/script
echo 'Hello, Methody!'
[methody@localhost methody]$ grep -q Methody bin/script ; echo $?
0
[methody@localhost methody]$ grep -q Shogun bin/script ; echo $?
1
[methody@localhost methody]$ if grep -q Shogun bin/script ; then echo "Yes"; fi
[methody@localhost methody]$ if grep -q Methody bin/script ; then echo "Yes"; fi
Yes
```

Оператор `if` использует код возврата программы `grep`

Условный оператор `if` запускает команду-*условие*, `grep -q`, которая ничего не выводит на экран, зато возвращает `0`, если шаблон найден, и `1`, если нет. В зависимости от кода возврата этой команды, `if` выполняет или не выполняет *тело*: список команд, заключённый между `then` и `fi`. Точка с запятой *разделяет* операторы в `sh`; либо она, либо перевод строки необходимы перед `then` и `fi`, иначе всё, что идёт после `grep` интерпретатор посчитает параметрами этой утилиты.

Множеством функций обладает команда `test`: она умеет сравнивать числа и строки, проверять ярлык объекта файловой системы и наличие самого этого объекта. У "`test`" есть второе имя: "`[`" (как правило, `/usr/bin/[` -- символьная или даже жёсткая ссылка на `/usr/bin/test`), позволяющее оформлять оператор `if` более привычным образом:

```
[methody@localhost methody]$ if test -f examples ; then ls -ld examples ; fi
[methody@localhost methody]$ if [ -d examples ] ; then ls -ld examples ; fi
drwxr-xr-x  2 methody methody 4096 Окт 31 15:26 examples
[methody@localhost methody]$ A=8; B=6; if [ $A -lt $B ] ; then echo "$A<$B" ; fi
[methody@localhost methody]$ A=5; B=6; if [ $A -lt $B ] ; then echo "$A<$B" ; fi
5<6
```

Оператор `test`

Команда `test -f` проверяет, на является ли её аргумент файлом; поскольку `examples` -- это каталог, результат -- неуспешный. Команда `[ -d` -- то же самое, что и `test -d` (не каталог ли первый параметр), только *последним* параметром этой команды -- исключительно для красоты -- должен быть символ "`]`". Результат -- успешный, и выполняется команда `ls -ld`. Команда `test` *`параметр1`* `-lt` *`параметр3`* проверяет, является ли *`параметр1`* числом, меньшим, чем (less then) *`параметр3`*. В более сложных случаях оператор `if` удобнее записывать "лесенкой", выделяя переводами строки окончание условия и команды внутри тела (их может быть много).

Второй тип подстановки, которую shell делает внутри двойных кавычек -- это **подстановка вывода** команды. Подстановка вывода имеет вид "`` `команда` ``" (другой вариант -- "`$(команда)`"). Как и подстановка значения переменной, она происходит *перед* тем, как начнётся разбор командной строки: выполнив команду и получив от неё какой-то текст, shell примется разбирать его, как если бы этот текст пользователь набрал вручную. Это очень удобное средство, если то, что выводит команда, необходимо передать самому интерпретатору:

```
[methody@localhost methody]$ A=8; B=6
[methody@localhost methody]$ expr $A + $B
14
[methody@localhost methody]$ echo "$A + $B = `expr $A + $B`"
8 + 6 = 14
[methody@localhost methody]$ A=3.1415; B=2.718
[methody@localhost methody]$ echo "$A + $B = `expr $A + $B`"
expr: нечисловой аргумент
3.1415 + 2.718 =
[methody@localhost methody]$ echo "$A + $B" | bc
5.8595
[methody@localhost methody]$ C=`echo "$A + $B" | bc`
[methody@localhost methody]$ echo "$A + $B = $C"
3.1415 + 2.718 = 5.8595
```

Подстановка вывода команды

Сначала для арифметических вычислений Мефодий хотел воспользоваться командой `expr`, которая работает с параметрами командной строки. С целыми числами `expr` работает неплохо, и её результат можно подставить прямо в аргумент команды `echo`. С действительными числами умеет работать утилита-**фильтр** `bc`; арифметическое выражение пришлось сформировать с помощью `echo` и передать по конвейеру, а результат -- поместить в переменную `C`. Во многих современных командных оболочках есть *встроенная* целочисленная арифметика вида "`$(( выражение ))`".

Сценарии

В языке `sh` много внимания было уделено удобству написания **сценариев**. В частности, параметры командной строки, переданные сценарию, доступны в нём в виде переменных, имена которых совпадают с порядковым номером параметра:

```
[methody@localhost methody]$ cat > bin/two
#!/bin/sh
echo "Running $0: $*"
$1 $3
$2 $3
[methody@localhost methody]$ chmod +x bin/two
[methody@localhost methody]$ bin/two file "ls -ld" examples
Running bin/two: file ls -ld examples
examples: directory
drwxr-xr-x  2 methody methody 4096 Окт 31 15:26 examples
[methody@localhost methody]$ two "ls -s" wc "bin/two bin/loop" junk
Running /home/methody/bin/two: ls -s wc bin/two bin/loop junk
4 bin/loop  4 bin/two
 4  9 44 bin/two
 1  5 26 bin/loop
 5 14 70 итого
```

Использование позиционных параметров в сценарии

Как видно из примера, форма "`$номер_параметра`" позволяет обратиться и к *нулевому* параметру -- команде, а *вся* строка параметров хранится в переменной "`*`". Кроме того, свойство подстановки выполняться *до* разбора

командной строки позволило Мефодию передать в качестве *одного* параметра `"ls -ld"` или `"bin/two bin/loop"`, а интерпретатору -- разбить эти параметры на имя команды и ключи и два имени файла соответственно.

В `sh` есть и **оператор цикла** `while`, формат которого аналогичен `if`, и более удобный именно в сценариях оператор обхода списка `for` (список делится на слова так же, как и командная строка -- с помощью **разделителей**):

```
[methody@localhost methody]$ for Var in Wuff-Wuff Miaou-Miaou; do echo $Var; done
Wuff-Wuff
Miaou-Miaou
[methody@localhost methody]$ for F in `date`; do echo -n "<$F>"; done; echo
<Сбт><Ноя><6><12:08:38><MSK><2004>
[methody@localhost methody]$ cat > /tmp/setvar
QWERTY="$1"
[methody@localhost methody]$ sh /tmp/setvar 1 2 3; echo $QWERTY

[methody@localhost methody]$ . /tmp/setvar 1 2 3; echo $QWERTY
1
```

Использование `for` и операции "."

Во втором `for` Мефодий воспользовался **подстановкой** вывода команды `date`, каждое слово которой вывел с помощью `echo -n` в одну строку, а в конце команды пришлось вывести один перевод строки вручную.

Вторая половина примера иллюстрирует ситуацию, с которой Мефодий столкнулся во время своих экспериментов: все переменные, определяемые в сценарии, куда-то пропадают после окончания его работы. Оно и понятно: для обработки сценария всякий раз запускается *новый* интерпретатор (**дочерний процесс**!), и *все* его переменные принадлежат именно ему и с завершением процесса уничтожаются. Таким образом достигается отсутствие *побочных эффектов*: запуская программу, пользователь может быть уверен, что та не изменит окружения командной оболочки. Однако в *некоторых* случаях требуется обратное: запустить сценарий, который нужным образом настроит окружение. Единственный выход -- отдавать такой сценарий на обработку *текущему*, а не новому, интерпретатору (т. е. тому, что разбирает команды пользователя). Это и делается с помощью специальной команды ".". Если вдруг в передаваемом сценарии обнаружится команда `exit`, `exec` или какая-нибудь другая, приводящая к завершению работы интерпретатора, завершится именно *текущая* командная оболочка, чем сеанс работы пользователя в системе может и закончиться.

Настройка командного интерпретатора

Научившись (главным образом в результате чтения документации и непрестанных экспериментов) создавать работающие сценарии, Мефодий решил приступить к настройке командной оболочки, поскольку, как он слышал, для этого используются именно сценарии.

Привязка к клавишам

Оказалось, что настройка *управляющих клавиш* в `bash` не выглядит как сценарий, и даже имеет отношение не только к `bash`, а ко *всем* программам, использующим **библиотеку** терминального ввода `readline`. **Конфигурационный файл** `readline` называется `.inputrc` и состоит, в основном, из команд вида `"управляющая_последовательность": функция`, где `управляющая_последовательность` -- это символы, при получении которых `readline` выполнит `функцию` работы с вводимой строкой. Список всех функций `readline` можно узнать у `bash` по команде `bind -l`, а список всех *привязок* этих функций к клавиатурным последовательностям -- по команде `bind -p`. Мефодий вписал в `.inputrc` такие две строки:

```
"\e[5~": backward-word
"\e[6~": forward-word
```

Настройка `.inputrc`

Упомянутые в примере функции позволяют перемещать курсор в командной строке *по словам*, а ESC-последовательности возвращаются, соответственно, клавишами `Page Up` и `Page Down` виртуальной консоли Linux (сочетание "`e`" означает в `.Inputrc` клавишу ESC, то есть "`^[`", символ с ASCII-кодом `27`).

К одной и той же функции `readline` можно привязать *сколько угодно* управляющих последовательностей: например, клавиша `&home&` делает то же, что и "`^A`", `Стрелка вверх` -- то же, что и "`^P`", а `Del` -- то же, что и "`^D`" (только не в *пустой* строке!). Этим отчасти решается проблема *несовместимости* управляющих последовательностей терминалов: если в каком-нибудь терминале *другого* типа `Page Up` ил `Page Down` будут возвращать другие последовательности, Мефодий просто добавит в `.inputrc` ещё одну пару команд. Правда, Гуревич советовал вовсе отказаться от редактирования `.inputrc`, а воспользоваться утилитой `tput`, которая обращается к переменной `TERM` и базе данных по терминалам `terminfo` и готова выдать верную для любого данного терминала информацию по `kpp` (key previous page) и `knp` (key next page). Выдачу `tput` можно "скормить" той же `bind`, и получить команду, которая работает на *любом* терминале: `bind ""`tput kpp`": backward-word"` (кавычки, экранированные обратной косой чертой, `"`, передадутся `bind` в неизменном виде).

Стартовые сценарии

Настройка оболочки -- это в первую очередь настройка **окружения**. В начале сеанса работы (при запуске **стартового командного интерпретатора**) выполняется с помощью команды "." сценарий из файла со специальным именем -- `/etc/profile`. Это -- т. н. **общесистемный профиль**, стартовый сценарий, выполняющийся при входе в систему *любого*, кто использует командную оболочку, подобную `sh`. Следом выполняется **персональный**

профиль (или просто **профиль**) пользователя, сценарий, находящийся в домашнем каталоге, и называющийся `.profile`. Этот сценарий пользователь может видоизменять, как ему заблагорассудится.

Что касается `bash`, то структура его стартовых файлов сложнее. Прежде всего, `~/.profile` выполняется только если в домашнем каталоге нет файла `.bash_profile` или `.bash_login`, иначе стартовый сценарий берётся оттуда. В эти файлы можно помещать команды, несовместимые с другими версиями shell, например, управление **сокращениями** или привязку функций к клавишам. Кроме того, каждый *интерактивный* (взаимодействующий с пользователем), но не стартовый `bash` выполняет системный и персональный конфигурационные сценарии `/etc/bashrc` и `~/.bashrc`. Чтобы стартовый `bash` также выполнял `~/.bashrc`, соответствующую команду необходимо вписать в `~/.bash_profile`. Далее, каждый *неинтерактивный* (запущенный для выполнения сценария) `bash` сверяется с переменной окружения `BASH_ENV` и, если в этой переменной записано имя существующего файла, выполняет команды оттуда. Наконец, при завершении стартового `bash` выполняются команды из файла `~/.bash_logout`.

Пример настроек

Ниже приведены примеры конфигурационных файлов, которые Мефодий, сам или с помощью Гуревича, разместил в домашнем каталоге.

```
PS1="\u@\h:\w \$ "
EDITOR="/usr/bin/vim"
export PS1 EDITOR

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

Пример файла `.bash_profile`

В этом файле вызывается `~/.bashrc` (если он существует).

```
# User specific aliases and functions
if [ -r ~/.alias ]; then
        . ~/.alias
fi

# Source global definitions
if [ -r /etc/bashrc ]; then
        . /etc/bashrc
fi
```

Пример файла `.bashrc`

Мефодий решил, что сокращения удобнее будет хранить в отдельном файле -- `~/.alias`. Кроме того, вызывается сценарий `bashrc`, который Мефодий

обнаружил в каталоге `/etc`. Этот файл не входит в число автоматически выполняемых `bash`, поэтому его выполнение надо задавать явно.

```
alias > ~/.alias
```

Пример файла `.bash_logout`

Заметив, что команда `alias` выдаёт список сокращений в том же формате, в котором они и задаются, Мефодий придумал, как обойтись без редактирования файла `~/.alias`. Отныне все сокращения, определённые к моменту завершения сеанса работы, будут записываться обратно в `.alias`. Туда попадут и те сокращения, что прочлись во время выполнения `.bashrc`, и те, что впоследствии были определены вручную.

```
alias l='ls -FAC'
alias ls='ls --color=auto'
alias pd='pushd'
alias pp='popd'
alias v='ls -ali'
alias vi='/usr/bin/vim'
```

Пример файла `.alias`

Последняя запись в файле `.alias` относится к инструменту, которым Мефодий создавал все эти файлы: текстовому редактору `vim`. О текстовых редакторах речь пойдёт на следующем занятии.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Команды для удаления слова и всей строки
2. Назначение команды `stty`
3. Назначение команд "`^S`" и "`^Q`"
4. Порядок перемещения в командной строке по словам
5. Понятие «история команд». Порядок работы с историей команд, поиск по истории команд
6. Понятие «команда-сокращение» (aliases). Порядок работы и настройки сокращений
7. Понятие «достраивание». Порядок использования достраивания.
8. Понятие «шаблон».
9. Особенности обработки обычных и специальных символов
10. Использование «генерации имен файлов», экранирования специальных символов
11. Понятие «окружение»
12. Работа с переменными в `shell`
13. Назначение переменных окружения `PATH`, `LC_`, `TERM`, `EDITOR`, `UID`, `USER` или `PWD`, `PS`
14. Назначение операторов `if` и `test`
15. Использование «позиционных» параметров в сценариях
16. Использование циклов в сценариях
17. Понятие стартовых сценариев