08/14/12 Edited by: xuzhezhao Last edited: 09/05/12

Advanced Bash-Scripting Guide 学习笔记

Specia Characters

1. 分号（colon）代表空命令（null command）

- 类似于 shell 内置的 true 功能，执行后的 exit 值是 0，即$?=0。

  This is the shell equivalent of a "NOP" (no op, a do-nothing operation). **It may be considered a synonym for the shell builtin true.** The ":" command is itself a Bash builtin, and its exit status is true (0).

- **Endless loop**

```
while :
do
operation-1
operation-2
...
operation-n
done
# Same as:
while true
do
...
done
```

- **在 if/then 结构中作为占位符（Placeholder in if/then test）**

```
if condition
then：  # 注意“：”前面有空格，不加分号的话会产生语法错误
```

```
# Do nothing and branch ahead
else
# Or else ...
        take-some-action
fi
```

- **作为命令占位符（Provide a placeholder where a binary operation is expected）**

```
: ${username=`whoami`}
   #可以用分号预留一个命令的位置
# ${username=`whoami`}
# Gives an error without the leading :
#+ unless "username" is a command or builtin...
```

- **Evaluate string of variables using parameter substitution**

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
# Prints error message
#+ if one or more of essential environmental variables not set.
```

- **配合重定向，将文件变成空文件（truncates a file to zero length）**

In combination with the > redirection operator, truncates a file to zero length, without changing its permissions. If the file did not previously exist, creates it.

```
: > data.xxx
# File "data.xxx" now empty.
# Same effect as
cat /dev/null >data.xxx
# However, this does not fork a new process, since ":" is a builtin.
```

In combination with the >> redirection operator, has no effect on a pre-existing target file (**: >> target_file**). If the file did not previously exist, creates it.

- **还有一种不建议的用法，用作注释（May be used to begin a comment line, although this is not recommended）**

```
: This is a comment that generates an error, ( if [ $x -eq 3] ).
```

2. ！（bang）：取反 reverse (or negate) the sense of a test or exit status

The ! operator inverts the exit status of the command to which it is applied (see Example 6-2). It also inverts the meaning of a test operator. This can, for example, change the sense of equal (=) to not-equal (!=). The ! operator is a **Bash keyword**.

3. *（asterisk）：通配符（wild card）、乘号（multiplication）

在正则表达式（regular expression）中代表 0 个或任意多个前一个字符。

****：取幂运算符，或者是 extended file-match globbing.**

4. ？：test operator

In a **double-parentheses construct（双括号结构中），**the ? can serve as an element of a C-style trinary operator（三目运算符）.

condition?result-if-true:result-if-false

```
(( var0 = var1<98?9:21 ))
if [ "$var1" -lt 98 ]
then
var0=9
else
var0=21
fi
```

In a parameter substitution expression, the ? tests whether a variable has been set.

5. $：变量代换 Variable substitution (contents of a variable).

- **$var**：显示 var 内容
- **$**：正则表达式的结尾标志。end-of-line. In a regular expression, a "$" addresses the end of a line of text.

- **${ }：**参数代换（Parameter substitution）.

- **$' ... '：**将 ascii 码转成字符形式，**如 echo $'\111'。**Quoted string expansion. This construct expands single or multiple escaped octal or hex values into ASCII [16] or Unicode characters.

- **$*,$@：**positional parameters.

- **$?：**上一命令的返回值。exit status variable. The $? variable holds the exit status of a command, a function, or of the script itself.

- **$$：**script 所在执行环境中的进程 ID，process ID variable. The $$ variable holds the process ID of the script in which it appears.

6. （ ）：命令组（command group）

```
(a=hello; echo $a)
```

括号中的命令是在子 shell 中执行（A listing of commands within parentheses starts a subshell），子 shell 要的变量在父 shell 中不可见。

数组初始化（array initialization）

```
Array=(element1 element2 element3)
echo ${Array[0]}
; echo ${Array[1]}; echo ${Array[2]}
```

7. { }：Brace expansion

```
echo \"{These,words,are,quoted}\"
# "These" "words" "are" "quoted"
# " prefix and suffix
cat {file1,file2,file3} > combined_file
# Concatenates the files file1, file2, and file3 into combined_file.
cp file22.{txt,backup}
```

```
# Copies "file22.txt" to "file22.backup"
```

注意：No spaces allowed within the braces unless the spaces are quoted or escaped.

```
echo {file1,file2}\ :{\ A," B",' C'}
```

- **{a..z}：Extended Brace expansion**

```
echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
# Echoes characters between a and z.
echo {0..3} # 0 1 2 3
# Echoes characters between 0 and 3.
base64_charset=( {A..Z} {a..z} {0..9} + / = )
# Initializing an array, using extended brace expansion.
# From vladz's "base64.sh" example script.
```

- **{ }：Block of code [curly brackets].** Also referred to as an inline group, this construct, in effect, creates

an anonymous function (a function without a name). However, unlike in a "standard" function, the

variables inside a code block remain visible to the remainder of the script.

```
a=123
{ a=321; }  #大括号内的命令必须要以分号或者换行来结束
echo "a = $a"
# a = 321
(value inside code block)
# Thanks, S.C.
```

- **重定向 Code blocks and I/O redirection**

```
#!/bin/bash
# Reading lines in /etc/fstab.
File=/etc/fstab
{
read line1
read line2
} < $File
echo "First line in $File is:"
echo
```

```
echo  "$line1"
echo "Second line in $File is:"
echo "$line2"
exit 0
# Now, how do you parse the separate fields of each line?
# Hint: use awk, or . . .
# . . . Hans-Joerg Diers suggests using the "set" Bash builtin.
```

注意：一般情况下大括号里的命令在当前进程中执行，不会在子进程中执行。但也有例外，如 a code block in braces as part of a pipe may run as a subshell.

```
ls | { read firstline; read secondline; }
#  Error. The code block in braces runs as a subshell,
#+ so the output of "ls" cannot be passed to variables within the block.
echo "First line is $firstline; second line is $secondline"  # Won't work.
```

- **placeholder for text**

Used after xargs -i (replace strings option). The {} double curly brackets are a placeholder for output text.

```
ls . | xargs -I {} -t cp ./{} $1
 #关于 xargs 用法参看 htm 资料
# From "ex42.sh" (copydir.sh) example.
```

8. > &> >& >> < <> ：redirection

9. |：管道命令 pipe

Passes the output (stdout of a previous command to the input (stdin) of the next one, or to the shell. This is a method of chaining commands together.

管道后的命令是在子进程中执行（A pipe runs as a child process, and therefore cannot alter script variables）。

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"
# variable = initial_value
```

如果管道内的命令中断了，将会中断整个管道命令的执行，并产生一个 **SIGPIPE** 信号（If

one of the commands in the pipe aborts, this prematurely terminates execution of the pipe. Called a broken pipe, this condition sends a SIGPIPE signal）。

10. >|：force redirection (even if the noclobber option is set). This will forcibly overwrite an existing file.

11. ||：或运算（OR logical operator ）

In a test construct, the || operator causes a return of 0 (success) if either of the linked test conditions is true.

12. &：Run job in background

Within a script, commands and even loops may run in the background.

但是在 script 里面使用 & 后台运行可能会有问题，这会使这个 script 挂起（A command run in the background within a script may cause the script to hang, waiting for a keystroke. Fortunately, there is a remedy for this. ）。

如下面的例子：

```
#!/bin/bash
# background-loop.sh
for i in 1 2 3 4 5 6 7 8 9 10
# First loop.
do
echo -n "$i "
done &  # Run this loop in background.
# Will sometimes execute after second loop.
echo
# This 'echo' sometimes will not display.
for i in 11 12 13 14 15 16 17 18 19 20
do
echo -n "$i "
done
echo
# Second loop.
# This 'echo' sometimes will not display.
# ==================================================
# The expected output from the script:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20
```

```
# Sometimes, though, you get:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (The second 'echo' doesn't execute. Why?)
# Occasionally also:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (The first 'echo' doesn't execute. Why?)
# Very rarely something like:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# The foreground loop preempts the background one.
exit 0
# Nasimuddin Ansari suggests adding sleep 1
#+ after the echo -n "$i" in lines 6 and 14,
#+ for some real fun.
```

13. &&：与运算（AND logical operator ）

In a test construct, the && operator causes a return of 0 (success) only if both the linked test conditions are true.

14. -：option, prefix

redirection from/to stdin or stdout [dash].

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

```
bash$ file -
abc
standard input:             ASCII text
```

添加一行到文件开头（prepending lines to a file）

```
file=data.txt
title="***This is the title line of data text file***"
echo $title | cat - $file >$file.new
```

```
cd - ：上一个工作目录，变量为$OLDPWD
```

15. --：double dash

- **作为命令参数的结尾**，可以用来处理文件名含“-” 的文件（Used with a Bash builtin, it means the end of options to that particular command）。

```
touch -- -test; rm -- -test
```

- **与 set 一起用来设置位置参数**（The double-dash is also used in conjunction with set.）。Note that in this context the "-" is not itself a Bash operator, but rather an option recognized by certain UNIX utilities that write to stdout, such as tar, cat, etc.

```
xzz@xzz-ubuntu:~$ variable="one two three four five"
xzz@xzz-ubuntu:~$ set -- $variable
xzz@xzz-ubuntu:~$ echo $1
one
xzz@xzz-ubuntu:~$ echo $2
two
xzz@xzz-ubuntu:~$ echo $3
three
xzz@xzz-ubuntu:~$ echo $4
four
xzz@xzz-ubuntu:~$ set --    #取消位置参数的设置
xzz@xzz-ubuntu:~$ echo $1
```

16. ~+：current working directory
~-：previous working directory

17. ^、^^：大小写转换

```
#!/bin/bash4

var=veryMixedUpVariable
echo ${var}          # veryMixedUpVariable
echo ${var^}         # VeryMixedUpVariable
#       *           First char --> uppercase.
echo ${var^^}        # VERYMIXEDUPVARIABLE
#       **          All chars  --> uppercase.
echo ${var,}         # veryMixedUpVariable
#       *           First char --> lowercase.
echo ${var,,}        # verymixedupvariable
#       **          All chars  --> lowercase.
```

Variable and Parameters

18. 变量（Variable）

- **变量的名字实际上是一个指向变量值的指针**（A variable's name is, in fact, a reference, a pointer to the memory location(s) where the actual data associated with that variable is kept）。
- **未初始化的变量就有值，但在参与数学运算时值为 0**（An uninitialized variable has no value, however it evaluates as 0 in an arithmetic operation）。

19. 变量赋值

```
a=`ls -l`       # Assigns result of 'ls -l' command to 'a'
echo $a         # Unquoted, however, it removes tabs and newlines.
echo
echo "$a"       # The quoted variable preserves whitespace.
              # (See the chapter on "Quoting.")
```

20. bash 变量类型

不像其他的程序语言，**bash 中的变量没有分成各种类型**，bash 的变量肯定是字符串，但根据情况的不同，bash 变量也可以参与数学运算，此时的变量必须仅包含数字（Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, Bash variables are character strings, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits）。

21. 特殊变量类型（Specia Varibale Types）

- **本地（局部）变量（**Local variables **）**
  只在子 shell 或当前函数中有效。

- **环境（全局）变量（**Environmental variables **）**
  子 shell（ inherit）继承父 shell 的环境变量，子 shell 中定义的变量在父 shell 中无效。
  In a more general context, each process has an "environment", that is, a group of variables that the process may reference. In this sense, the shell behaves like any other process.

22. 位置变参数（Positional parameters）

- 从命令行传递给脚本的参数，**$0，$1，$2**……
- **$0 代表脚本的名字，$1 是脚本后的第一个参数，9 以后的参数要这样表示${10}，${11}。**（$0 is the name of the script itself, $1 is the first argument, $2 the second, $3 the third, and so forth. After $9, the arguments must be enclosed in brackets, for example, **${10}, ${11}, ${12}**）
- **$*：**所有位置参数，参数以分隔符分开
- **$@：**代表所有位置参数
- **$#：**位置参数的个数，不包括$0

23. 间接引用（ indirect referencing）

```
args=$#           # Number of args passed.
lastarg=${!args}
# Note: This is an *indirect reference* to $args ...

# Or:      lastarg=${!#}          (Thanks, Chris Monson.)
# This is an *indirect reference* to the $# variable.
# Note that lastarg=${!$#} doesn't work.
```

24. shift

使用 shift 时要注意一个问题

```
#  However, as Eleni Fragkiadaki, points out,
#+ attempting a 'shift' past the number of
#+ positional parameters ($#) returns an exit status of 1,
#+ and the positional parameters themselves do not change.
#  This means possibly getting stuck in an endless loop. . . .
```

```
#  For example:
#      until [ -z "$1" ]
#      do
#        echo -n "$1 "
#        shift 20    #  If less than 20 pos params,
#      done          #+ then loop never ends!
#
# When in doubt, add a sanity check. . . .
#          shift 20 || break
```

Quoting

25. Quoting

Quoting means just that, bracketing a string in quotes. This has the effect of **protecting special characters in the string from reinterpretation or expansion by the shell or shell script.** A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in `globbing` and Regular Expressions).

In everyday speech or writing, when we "quote" a phrase, we set it apart and give it special meaning. In a Bash script, when wequote a string, we **set it apart and protect its literal meaning.**

- **某些程序可以将引号中的内容也进行扩展**（Certain programs and utilities reinterpret or expand special characters in a quoted string. An important use of quoting is protecting a command-line parameter from the shell, but still letting the calling program expand it）。如 grep：

```
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
 file2.txt:This is the First line of file2.txt.
```

- **引用强制 echo 打印换行符**（Quoting can also suppress echo's "appetite" for newlines）

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh
```

```
bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bo bo  13 Aug 21 12:57 t.sh
 -rw-rw-r--  1 bo bo  78 Aug 21 12:57 u.sh
```

26. Quoting Variables

- **双引号（double quotes）：**When referencing a variable, it is generally advisable to **enclose its name in double quotes（双引号）.** This prevents reinterpretation of all special characters within the quoted string -- **except $, ` (backquote), and \ (escape).**
  An argument enclosed in double quotes presents itself as a single word, even if it contains whitespace separators.

  A more elaborate example:

```
variable2=""    # Empty.

COMMAND $variable2 $variable2 $variable2
        # Executes COMMAND with no arguments.
COMMAND "$variable2" "$variable2" "$variable2"
        # Executes COMMAND with 3 empty arguments.
COMMAND "$variable2 $variable2 $variable2"
        # Executes COMMAND with 1 argument (2 spaces).
```

- **单引号（single quote）：**Within **single quotes**, every special character except ' gets interpreted literally. Consider single quotes ("**full quoting**") to be a **stricter method** of quoting than **double quotes** ("**partial quoting**").

  注意：在单引号中不能再有单引号（Since even the escape character (\) gets a literal interpretation within single quotes, trying to enclose a single quote within single quotes will not yield the expected result）。

- **Example 5-1. Echoing Weird Variables（很奇怪的结果，没有搞明白）**

```
#!/bin/bash
# weirdvars.sh: Echoing weird variables.

echo

var="'(]\\{}\$\""
```

```
echo $var       # '(]\{}$"
echo "$var"     # '(]\{}$"     Doesn't make a difference.

echo

IFS='\'
echo $var       # '(] {}$"     \ converted to space. Why?
echo "$var"     # '(]\{}$"

# Examples above supplied by Stephane Chazelas.

echo

var2="\\\\\""
echo $var2      #   "
echo "$var2"    # \"
echo
# But ... var2="\\\\"" is illegal. Why?
var3='\\\\'
echo "$var3"    # \\\\
# Strong quoting works, though.

exit
```

27. 转义（Escaping）

The escape (\) preceding a character tells the shell to interpret that character literally.

注意：With certain commands and utilities, such as echo and sed, **escaping a character may have the opposite effect** - it can toggle on a special meaning for that character.

Exit and Exit Status

28. Exit

The exit command terminates a script, just as in a C program. It can also return a value, which is available to the script's parent process.

29. ！：reverse the exit status

Logical not qualifier, reverses the outcome of a test or command, and this affects its exit status.

```
# Preceding a _pipe_ with ! inverts the exit status returned.
ls | bogus_command     # bash: bogus_command: command not found
echo $?                # 127

! ls | bogus_command   # bash: bogus_command: command not found
echo $?                # 0
```

Tests

30. ((...))、let 结构返回值

The **((...))** and **let** ... constructs return an exit status, according to whether the arithmetic expressions they evaluate expand to a non-zero value.

```
(( 0 && 1 ))               # Logical AND
echo $?     # 1     ***
let "num = (( 0 && 1 ))"
echo $num   # 0
# But ...
let "num = (( 0 && 1 ))"
echo $?     # 1     ***

(( 200 || 11 ))            # Logical OR
echo $?     # 0     ***


(( 200 | 11 ))             # Bitwise OR
echo $?                    # 0     ***
let "num = (( 200 | 11 ))"
echo $num                  # 203
let "num = (( 200 | 11 ))"
echo $?                    # 0     ***
```

Again, note that **the *exit status* of an arithmetic expression is *not* an error value.**

```
var=-2 && (( var+=2 ))
echo $?                 # 1

var=-2 && (( var+=2 )) && echo $var
                        # Will not echo $var!
```

31. if...then

An **if can test any command**, not just conditions enclosed within brackets.

```
word=Linux
letter_sequence=inu
if echo "$word" | grep -q "$letter_sequence"
# The "-q" option to grep suppresses output.
then
  echo "$letter_sequence found in $word"
else
  echo "$letter_sequence not found in $word"
fi


if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
  then echo "Command succeeded."
  else echo "Command failed."
fi
```

Both **if** and **then** are **keywords**. Keywords (or commands) begin statements, and before a new statement on the same line begins, the old one must terminate.

```
if [ -x "$filename" ]; then
```

elif : elif is a contraction for else if.

- **注意：**The "**if COMMAND**" construct returns the **exit status of COMMAND.**

32. test：shell builtin 命令

```
type test
test is a shell builtin
```

If, for some reason, you wish to use /usr/bin/test in a Bash script, then specify it by full pathname.

```
if /usr/bin/test -z "$1"      # Equivalent to "test" builtin.

if [ -z "$1" ]              # Functionally identical to above code blocks.

if /usr/bin/[ -z "$1" ]       # Again, functionally identical to above.
# if /usr/bin/[ -z "$1"       # Works, but gives an error message.
```

33. [：与 test 相同

```
type '['
[ is a shell builtin
```

The if test condition-true construct is the exact equivalent of if [condition-true]. As it happens, **the left bracket ' [', is a token which invokes the test command.** The closing right bracket,] , in an if/test should not therefore be strictly necessary, however newer versions of Bash require it.

34. [[...]]：extended test command

- The [[ ]] construct is the more versatile Bash version of [ ]. This is the extended test command, adopted from ksh88.
- **可进行逻辑运算。**Using the **[[...]]** test construct, rather than [...] can **prevent many logic errors** scripts. For example, **the &&, ||, <, and >operators work within a [[ ]] test, despite giving an error within a [ ] construct.**
- **可以实现进制的自动转换。**Arithmetic evaluation of octal / hexadecimal constants takes place automatically within a [[...]] construct.

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"       # 15 is not equal to 017
fi      # Doesn't evaluate within [ single brackets ]!
```

```
if [[ "$decimal" -eq "$octal" ]]
then
 echo "$decimal equals $octal"            # 15 equals 017
else
 echo "$decimal is not equal to $octal"
fi     # Evaluates within [[ double brackets ]]!

if [[ "$decimal" -eq "$hex" ]]
then
 echo "$decimal equals $hex"              # 15 equals 0x0f
else
 echo "$decimal is not equal to $hex"
fi     # [[ $hexadecimal ]] also evaluates!
```

35. ((...)) : expands and evaluates an arithmetic expression

- **表达式中的值为 0，就返回 1 或 false，为非 0 就返回 0 或 true。**If the expression evaluates as zero, it returns an exit status of 1, or "false". A non-zero expression returns an exit status of 0, or "true". This is in marked contrast (相反) to using the test and [ ] constructs previously discussed.

- **可以用在 if...then 结构中。**((...)) also useful in an if-then test.

```
var1=5
var2=4

if (( var1 > var2 ))
then #^      ^      Note: Not $var1, $var2. Why?
 echo "$var1 is greater than $var2"
fi     # 5 is greater than 4

exit 0
```

36. token

A token is a symbol or short string with a special meaning attached to it (a meta-meaning). In Bash, certain tokens, such as **[** and **.** (dot-command), may expand to keywords and commands.

37. Example 7-4. Testing for broken links

38. integer comparison

- **< : is less than (within double parentheses)**
  (("$a" < "$b"))
- **<=、>、>= : 同上**

39. string comparison

- **= : if ["$a" = "$b"]**

- **== : a synonym for =**
  The == comparison operator behaves differently within a double-brackets test than within single brackets.

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching).
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).

[ $a == z* ]     # File globbing and word splitting take place.
[ "$a" == "z*" ] # True if $a is equal to z* (literal matching).

#glob 是一种特殊的模式匹配。最常见的即是通配符的扩展，如 bash 下输入“ls *.log”命令，详细 man glob。
```

- **!= : not equal**
  This operator uses pattern matching within a [[...]] construct.

- **< : is less than, in ASCII alphabetical order**

```
if [[ "$a" < "$b" ]]
if [ "$a" \< "$b" ]
```

  Note that the "<" needs to be escaped within a [ ] construct.

- **> : is greater than, in ASCII alphabetical order**

- **-z : string is null, that is, has zero length**

- **-n : string is not null.**
  The -n test requires that the string be quoted within the test brackets. Using an unquoted string with ! -z, or even just the unquoted string alone within test brackets (see Example 7-6) normally works, however, this is an unsafe practice. Always quote a tested string.

40. compound comparison

- **-a : logical and**

exp1 -a exp2 returns true if both exp1 and exp2 are true.

- **-o : logical or**
  exp1 -o exp2 returns true if either exp1 or exp2 is true.

But, as *rihad* points out:

```
[ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ]   # true
[ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ]   # (no output)
# ^^^^^^^ False condition. So far, everything as expected.

# However ...
[ 1 -eq 2 -a -n "`echo true 1>&2`" ]       # true
# ^^^^^^^ False condition. So, why "true" output?

# Is it because both condition clauses within brackets evaluate?
[[ 1 -eq 2 && -n "`echo true 1>&2`" ]]     # (no output)
# No, that's not it.

# Apparently && and || "short-circuit" while -a and -o do not.
```

41. 嵌套式 if/then 结构（Nested *if/then* Condition Tests）

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# Same result as:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

Operations and Related Topics

42. 最大公约数算法（greatest common divisor）

```
gcd (){

 dividend=$1            #  Arbitrary assignment.
 divisor=$2            #! It doesn't matter which of the two is larger.
                       #  Why not?

 remainder=1            #  If an uninitialized variable is used inside
                       #+ test brackets, an error message results.

 until [ "$remainder" -eq 0 ]
 do    #   Must be previously initialized!
   let "remainder = $dividend % $divisor"
   dividend=$divisor     # Now repeat with 2 smallest numbers.
   divisor=$remainder
 done                  # Euclid's algorithm

}                      # Last $dividend is the gcd.
```

43. Arithmetic Operations

```
: $((n = $n + 1))
#  ":" necessary because otherwise Bash attempts
#+ to interpret "$((n = $n + 1))" as a command.
echo -n "$n "

(( n = n + 1 ))
#  A simpler alternative to the method above.
#  Thanks, David Lombard, for pointing this out.
echo -n "$n "

: $[ n = $n + 1 ]
#  ":" necessary because otherwise Bash attempts
#+ to interpret "$[ n = $n + 1 ]" as a command.
```

```
#  Works even if "n" was initialized as a string.
echo -n "$n "


# Now for C-style increment operators.
# Thanks, Frank Wang, for pointing this out.

let "n++"         # let "++n"  also works.

(( n++ ))         # (( ++n ))  also works.

: $(( n++ ))      # : $(( ++n )) also works.

: $[ n++ ]        # : $[ ++n ] also works
```

注意：bash 中浮点数将作为字符串处理，使用 bc 或者数学库函数处理浮点运算（Bash does not understand floating point arithmetic. It treats numbers containing a decimal point as strings.
Use bc in scripts that that need floating point calculations or math library functions.)

+=：可以用在字符串的添加上

```
PATH+=:/opt/bin
```

44. 位运算符（bitwise operators）

The bitwise operators seldom make an appearance in shell scripts. Their chief use seems to be manipulating and testing values read from ports or sockets. "Bit flipping" is more relevant to compiled languages, such as C and C++, which provide direct access to system hardware. However, see vladz's ingenious use of bitwise operators in his base64.sh (Example A-54) script.

45. 逗号运算符

与 C 语言用法相同，主要应用在 for 循环中。The comma operator chains together two or more arithmetic operations. All the operations are evaluated (with possibleside effects.

46. 进制转换

bash 中默认是十进制，八进制前面加 0，十六进制前面加 0x 或 0X，用“**进制#数**”来指定进制。

```
# Octal: numbers preceded by '0' (zero)
let "oct = 032"
echo "octal number = $oct"              # 26
```

```
# Expresses result in decimal.
# --------- ------ -- -------

# Hexadecimal: numbers preceded by '0x' or '0X'
let "hex = 0x32"
echo "hexadecimal number = $hex"        # 50

echo $((0x9abc))                        # 39612
#  double-parentheses arithmetic expansion/evaluation
# Expresses result in decimal.
```

```
# Other bases: BASE#NUMBER
# BASE between 2 and 64.
# NUMBER must use symbols within the BASE range, see below.

let "bin = 2#111100111001101"
echo "binary number = $bin"             # 31181

let "b32 = 32#77"
echo "base-32 number = $b32"            # 231

let "b64 = 64#@_" #64 进制的表示字符要有 64 个，所以就加上了特殊字符来表示
echo "base-64 number = $b64"            # 4031
# This notation only works for a limited range (2 - 64) of ASCII characters.
# 10 digits + 26 lowercase characters + 26 uppercase characters + @ + _

#  Important note:
#  --------------
#  Using a digit out of range of the specified base notation
#+ gives an error message.

let "bad_oct = 081"
# (Partial) error message output:
#  bad_oct = 081: value too great for base (error token is "081")
#             Octal numbers use only digits in the range 0 - 7.
```

47. ((...))：支持 C-style

```
(( a++ )) #  Post-increment 'a', C-style.
(( ++a ))   #  Pre-increment 'a', C-style.
```

```
(( a-- )) #  Post-decrement 'a', C-style.
(( --a ))     #  Pre-decrement 'a', C-style.

(( t = a<45?7:11 )) # C-style trinary operator.

#  Note that, as in C, pre- and post-decrement operators
#+ have different side-effects.
n=1; let --n && echo "True" || echo "False"  # False
n=1; let n-- && echo "True" || echo "False"  # True
```

48. To avoid confusion or error in a complex sequence of test operators, break up the sequence into bracketed sections

```
if [ "$v1" -gt "$v2"  -o  "$v1" -lt "$v2"  -a  -e "$filename" ]
# Unclear what's going on here...

if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e "$filename" ]]
# Much better -- the condition tests are grouped in logical sections.
```

Another Look at Variables

49. Internal Variables

- **$BASH**
  The path to the Bash binary itself
- **$BASH_ENV**
  An environmental variable pointing to a Bash startup file to be read when a script is invoked
- **$BASH_SUBSHELL**
  A variable indicating the subshell level. This is a new addition to Bash, version 3.
- **$BASHPID**
  Process ID of **the current instance of Bash**. This is not the same as the $ $ variable, but it often gives the same result. $$ is the PID of parent bash.
- **$BASH_VERSINFO[n]**

```
 BASH_VERSINFO[0] = 3                 # Major version no.
# BASH_VERSINFO[1] = 00                  # Minor version no.
```

```
# BASH_VERSINFO[2] = 14                 # Patch level.
# BASH_VERSINFO[3] = 1                  # Build version.
# BASH_VERSINFO[4] = release            # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture
                                        # (same as $MACHTYPE).
```

- **$BASH_VERSION**

  The version of Bash installed on the system**.** Checking $BASH_VERSION is a good method of determining which shell is running. $SHELL does not necessarily give the correct answer.

- **$CDPATH**

  A colon-separated list of search paths available to the cd command, similar in function to the $PATH variable for binaries. The $CDPATH variable may be set in the local ~/.bashrc file.

```
bash$ CDPATH=/usr/share/doc
bash$ cd zip   # /usr/share/doc/目录下面有 zip 这个文件夹
/usr/share/doc/zip
```

- **$DIRSTACK**

  The top value in the directory stack (affected by pushd and popd).

  This builtin variable corresponds to the dirs command, however dirs shows the entire contents of the directory stack.

- **$EDITOR**

  The default editor invoked by a script, usually vi or emacs.

- **$EUID**

  "effective" user ID number

- **$FUNCNAME**

  Name of the current function

- **$GLOBIGNORE**

  A list of filename patterns to be excluded from matching in `globbing`.

- **$GROUPS[array]**

  Groups current user belongs to.

  This is a listing (array) of the group id numbers for current user, as recorded in /etc/passwd and /etc/group.

- **$HOME**

  Home directory of the user, usually /home/username

- **$HOSTNAME**

  The hostname command assigns the system host name at bootup in an init script. However, the gethostname() function sets the Bash internal variable $HOSTNAME. See also Example 10-7.

- **$HOSTTYPE**

  host type

  Like $MACHTYPE, identifies the system hardware.

- **$IFS**

  internal field separator

  This variable determines how Bash recognizes fields, or word boundaries, when it interprets character strings.

  $IFS defaults to whitespace (space, tab, and newline), but may be changed, for example, to parse a comma-separated data file. Note that $* uses the first character held in $IFS. See Example 5-1.

```
IFS='\'
echo $var       # '(] {}$"     \ converted to space. Why?
echo "$var"     # '(]\{}$"

bash$ echo "$IFS" | cat -vte
 ^I$
 $
(Show whitespace: here a single space, ^I [horizontal tab],
 and newline, and display "$" at end-of-line.)


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
(Read commands from string and assign any arguments to pos params.)
```

Example 9-1. $IFS and whitespace

```
IFS=:
var=":a::b:c:::"              # Same pattern as above,
#                               #+ but substituting ":" for " "  ...
output_args_one_per_line $var
# []
# [a]
# []
# [b]
# [c]
# []
# []

# Note "empty" brackets.
```

```
# The same thing happens with the "FS" field separator in awk.
```

- **$IGNOREEOF**

  Ignore EOF: how many end-of-files (control-D) the shell will ignore before logging out.
- **$LC_COLLATE**

  Often set in the `.bashrc` or /etc/profile files, this variable controls collation order in filename expansion and pattern matching. If mishandled, LC_COLLATE can cause unexpected results in `filename globbing`.

  As of version 2.05 of Bash, filename globbing no longer distinguishes between lowercase and uppercase letters in a character range between brackets. For example, ls [A-M]* would match both File1.txt and file1.txt. To revert to the customary behavior of bracket matching, set LC_COLLATE to C by an export LC_COLLATE=C in/etc/profile and/or ~/.bashrc.
- **$LC_CTYPE**

  This internal variable controls character interpretation in `globbing` and pattern matching.
- **$LINENO**

  This variable is the line number of the shell script in which this variable appears. It has significance only within the script in which it appears, and is chiefly useful for debugging purposes.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_  # Save it.

echo "At line number $LINENO, variable \"v1\" = $v1"
echo "Last command argument processed = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

- **$MACHTYPE**

  machine type

  Identifies the system hardware
- **$OLDPWD**

  Old working directory ("OLD-Print-Working-Directory", previous directory you were in).
- **$OSTYPE**

  operating system type
- **$PATH**

  Path to binaries, usually /usr/bin/, /usr/X11R6/bin/, /usr/local/bin, etc.

  When given a command, the shell automatically does a hash table search on the

directories listed in the path for the executable. The path is stored in the environmental variable, $PATH, a list of directories, separated by colons. Normally, the system stores the $PATH definition in /etc/profile and/or ~/.bashrc (see Appendix G).

The current "working directory", ./, is usually omitted from the $PATH as a security measure.

- **$PIPESTATUS**

  Array variable holding exit status(es) of last executed foreground pipe.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo ${PIPESTATUS[0]}
0
bash$ echo ${PIPESTATUS[1]}
127

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```

The members of the $PIPESTATUS array hold the exit status of each respective command executed in a pipe. $PIPESTATUS[0]holds the exit status of the first command in the pipe, $PIPESTATUS[1] the exit status of the second command, and so on.

- **$PPID**

  The $PPID of a process is the process ID (pid) of its parent process.

  Compare this with the pidof command.

- **$PROMPT_COMMAND**

  A variable holding a command to be executed just before the primary prompt, $PS1 is to be displayed.

- **$PS1**

  This is the main prompt, seen at the command-line.

- **$PS2**

  The secondary prompt, seen when additional input is expected. It displays as ">".

- **$PS3**

  The tertiary prompt, displayed in a select loop (see Example 11-29).

- **$PS4**

The quartenary prompt, shown at the beginning of each line of output when invoking a script with the -x option. It displays as "+".

- **$PWD**
  Working directory (directory you are in at the time)
- **$REPLY**
  The default value when a variable is not supplied to read. Also applicable to select menus, but only supplies the item number of the variable chosen, not the value of the variable itself.

```
# REPLY is the default value for a 'read' command.

echo
echo -n "What is your favorite vegetable? "
read

echo "Your favorite vegetable is $REPLY."
#  REPLY holds the value of last "read" if and only if
#+ no variable supplied.
```

- **$SECONDS**
  The number of seconds the script has been running.
- **$SHELLOPTS**
  The list of enabled shell options, a readonly variable.
- **$SHLVL**
  Shell level, how deeply Bash is nested. [3] If, at the command-line, $SHLVL is 1, then in a script it will increment to 2.
  This variable is not affected by subshells. Use $BASH_SUBSHELL when you need an indication of subshell nesting.
- **$TMOUT**
  If the $TMOUT environmental variable is set to a non-zero value time, then the shell prompt will time out after $time seconds. This will cause a logout.

Example 9-2. Timed Input

- **$UID**
  User ID number
  This is the current user's real id, even if she has temporarily assumed another identity through su.
- The variables **$ENV, $LOGNAME, $MAIL, $TERM, $USER,**
  and **$USERNAME** are not Bash builtins. These are, however, often set as environmental variables in one of the Bash startup files. **$SHELL**, the name of the user's login shell, may be set from /etc/passwd or in an "init" script, and it is likewise

not a Bash builtin.

50. Positional Parameters

- **$#**
  Number of command-line arguments or positional parameters (see Example 36-2)
- **$***
  All of the positional parameters, **seen as a single word**
  "$*" must be quoted.
- **$@**
  Same as $*, but **each parameter is a quoted string,** that is, the parameters are passed on intact, without interpretation or expansion. This means, among other things, that each parameter in the argument list is seen as a separate word.
  Of course, "$@" should be quoted.

Example 9-6. *arglist*: Listing arguments with $* and $@

```
xzz@xzz-ubuntu:~$ ./pos.sh one two three
Listing args with "$*":
Arg #1 = one two three
Entire arg list seen as single word.

Listing args with "$@":
Arg #1 = one
Arg #2 = two
Arg #3 = three
Arg list seen as separate words.

Listing args with $* (unquoted):
Arg #1 = one
Arg #2 = two
Arg #3 = three
Arg list seen as separate words.
```

The $* and $@ parameters sometimes display inconsistent and puzzling behavior, depending on the setting of $IFS.

51. Other Special Parameters

- **$-**
  Flags passed to script (using set). See Example 15-16.
  注意：This was originally a ksh construct adopted into Bash, and unfortunately it does

not seem to work reliably in Bash scripts. One possible use for it is to have a script self-test whether it is interactive.

- **$!**
  PID (process ID) of last job run in background

```
IMEOUT=30   # Timeout value in seconds
count=0

possibly_hanging_job & {
        while ((count < TIMEOUT )); do
                eval '[ ! -d "/proc/$!" ] && ((count = TIMEOUT))'
                # /proc is where information about running processes is found.
                # "-d" tests whether it exists (whether directory exists).
                # So, we're waiting for the job in question to show up.
                ((count++))
                sleep 1
        done
        eval '[ -d "/proc/$!" ] && kill -15 $!'
        # If the hanging job is running, kill it.
```

- **$_**
  Special variable set to final argument of previous command executed.
- **$?**
  Exit status of a command, function, or the script itself (see Example 24-7)
- **$$**
  Process ID (PID) of the script itself. The $$ variable often finds use in scripts to construct "unique" temp file names (seeExample 32-6, Example 16-31, and Example 15-27). This is usually simpler than invoking mktemp.

52. Typing variables: declare or typeset

The declare or typeset builtins, which are exact synonyms, permit modifying the properties of variables. This is a very weak form of the typing [1] available in certain programming languages. The declare command is specific to version 2 or later of Bash. Thetypeset command also works in ksh scripts.

- **-r readonly**
  This is the rough equivalent of the C const type qualifier. An attempt to change the value of a readonly variable fails with an error message.
- **-i integer**
  Certain arithmetic operations are permitted for declared integer variables **without the**

need for expr or let.

- **-a array**
  The variable indices will be treated as an array.
- **-f function(s)**
  A declare -f line with no arguments in a script causes a listing of all the functions previously defined in that script.
- **-x export**
  This declares a variable as available for exporting outside the environment of the script itself.

53. Using the declare builtin restricts the scope of a variable

```
foo ()
{
FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar   # Prints bar.


#However ...
foo (){
declare FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar  # Prints nothing.


# Thank you, Michael Iatrou, for pointing this out.
```

The declare command can be helpful in **identifying variables,** environmental or otherwise. This can be especially useful with array

```
bash$ Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
bash$ echo ${Colors[@]}
purple reddish-orange light green
bash$ declare | grep Colors
Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
```

54. $RANDOM: generate random integer

$RANDOM is an internal Bash function (not a constant) that returns a pseudorandom integer in the range **0 - 32767**. It should notbe used to generate an encryption key.

```
number=$RANDOM

let "number %= $RANGE"
echo "Random number less than $RANGE  ---  $number"
```

```
Suites=" 1
2
3
4"
suite=($Suites)              # Read into array variable.
denomination=($Denominations)

num_suites=${#suite[*]}       # Count how many elements.
num_denominations=${#denomination[*]}
```

55. generating random numbers within a range

```
#  Generate random number between 6 and 30.
  rnumber=$((RANDOM%25+6))

#  Generate random number in the same 6 - 30 range,
#+ but the number must be evenly divisible by 3.
  rnumber=$(((RANDOM%30/3+1)*3))
```

```
#  Note that this will not work all the time.
#  It fails if $RANDOM%30 returns 0.

#  Frank Wang suggests the following alternative:
  rnumber=$(( RANDOM%27/3*3+6 ))

Bill Gradwohl came up with an improved formula that works for positive numbers.
$(((RANDOM%(max-min+divisibleBy))/divisibleBy*divisibleBy+min))
```

56. Reseeding RANDOM

As we have seen in the last example, it is best to reseed the RANDOM generator each time it is invoked. Using the same seed forRANDOM repeats the same series of numbers. (This mirrors the behavior of the random() function in C.)

```
# RANDOM=$$  seeds RANDOM from process id of script.
# It is also possible to seed RANDOM from 'time' or 'date' commands.

# Getting fancy...
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
#  Pseudo-random output fetched
#+ from /dev/urandom (system pseudo-random device-file),
#+ then converted to line of printable (octal) numbers by "od",
#+ finally "awk" retrieves just one number for SEED.
RANDOM=$SEED
random_numbers
```

The /dev/urandom pseudo-device file provides a method of generating much more "random" pseudorandom numbers (伪随机数, True "randomness," insofar as it exists at all, can only be found in certain incompletely understood natural phenomena, such as radioactive decay. Computers only simulate randomness, and computer-generated sequences of "random"numbers are therefore referred to as pseudorandom**.)than the $RANDOM variable**. dd if=/dev/urandom of=targetfile bs=1 count=XX creates a file of well-scattered pseudorandom numbers. **However, assigning these numbers to a variable in a script requires a workaround, such as filtering through od** (as in above example, Example 16-14, and Example A-36), or even piping to md5sum (see Example 36-14).

57. Seed

The seed of a computer-generated pseudorandom number series can be

considered an identification label. For example, think of the pseudorandom series with a seed of 23 as Series #23.

A property of a pseurandom number series is the length of the cycle before it starts repeating itself. A good pseurandom generator will produce series with very long cycles.

58. Pseudorandom numbers, using awk

```
# random2.sh: Returns a pseudorandom number in the range 0 - 1.
# Uses the awk rand() function.

AWKSCRIPT=' { srand(); print rand() } '
#           Command(s) / parameters passed to awk
# Note that srand() reseeds awk's random number generator.

echo -n "Random number between 0 and 1 = "

echo | awk "$AWKSCRIPT"
# What happens if you leave out the 'echo'?
```

Manipulating Strings

59. Manipulating Strings

Bash supports a surprising number of string manipulation operations. Unfortunately, these tools lack a unified focus. Some are a subset of parameter substitution, and others fall under the functionality of the UNIX expr command. This results in inconsistent command syntax and overlap of functionality, not to mention confusion.

60. String Length

- **${#string}**

  String length (number of characters in $var). For an array, ${#array} is the length of the

first element in the array.

Exceptions:

- **${#*} and ${#@} give the number of positional parameters.**
- **For an array, ${#array[*]} and ${#array[@]} give the number of elements in the array.**

- **expr length $string**
  These are the equivalent of strlen() in C.

- **expr "$string" : '.*'**

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

61. Length of Matching Substring at Beginning of String

- **expr match "$string" '$substring'**
  $substring is a regular expression.
- **expr "$string" : '$substring'**
  $substring is a regular expression.

```
stringZ=abcABC123ABCabc
#       |------|
#       12345678

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

62. Index

expr index $string $substring
Numerical position in $string of first character in $substring that matches.

```
stringZ=abcABC123ABCabc
#       123456 ...
echo `expr index "$stringZ" C12`            # 6
                                    # C position.

echo `expr index "$stringZ" 1c`             # 3
# 'c' (in #3 position) matches before '1'.

#This is the near equivalent of strchr() in C.
```

63. Substring Extraction

${string:position}

Extracts substring from $string at $position.

If the $string parameter is "*" or "@", then this extracts the positional parameters, starting at $position.

- **${string:position:length}**

  Extracts $length characters of substring from $string at $position.

```
# Is it possible to index from the right end of the string?

echo ${stringZ:-4}                    # abcABC123ABCabc
# Defaults to full string, as in ${parameter:-default}.
# However . . .

echo ${stringZ:(-4)}                  # Cabc
echo ${stringZ: -4}                   # Cabc
echo ${stringZ:2: -2}                 #cABC123ABCa
echo ${stringZ:1: -2}                 #bcABC123ABCa
# Now, it works.
# Parentheses or added space "escape" the position parameter.
```

If the $string parameter is "*****" **or** "**@**", then this extracts a maximum of $length positional parameters, starting at $position.

```
xzz@xzz-ubuntu:~$ set one two three
xzz@xzz-ubuntu:~$ echo ${*:2}
two three
xzz@xzz-ubuntu:~$ echo ${*:3}
three
xzz@xzz-ubuntu:~$ echo ${*:1}
one two three
xzz@xzz-ubuntu:~$ echo ${*:1:2}
one two
xzz@xzz-ubuntu:~$ echo ${*:1:3}
one two three
xzz@xzz-ubuntu:~$ echo ${*:2:2}
```

```
two three
#same as  “@”
```

- **expr substr $string $position $length**
  Extracts $length characters from $string starting at $position.

- **expr match "$string" '\($substring\)'**
  Extracts $substring at beginning of $string, **where $substring is a regular expression**.

- **expr "$string" : '\($substring\)'**
  Extracts $substring at beginning of $string, **where $substring is a regular expression.**

```
stringZ=abcABC123ABCabc
#       =======

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
# All of the above forms give an identical result.
```

- **expr match "$string" '.*\($substring\)'**
  Extracts $substring **at end of $string**, **where $substring is a regular expression.**

- **expr "$string" : '.*\($substring\)'**
  Extracts $substring **at end of $string, where $substring is a regular expression.**

```
stringZ=abcABC123ABCabc
#               ======

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

64. Substring Removal

- **${string#substring}**
  Deletes shortest match of $substring from front of $string.

- **${string##substring}**
  Deletes longest match of $substring from front of $string.

```
stringZ=abcABC123ABCabc
#       |----|          shortest
#       |----------|    longest
```

```
echo ${stringZ#a*C}      # 123ABCabc
# Strip out shortest match between 'a' and 'C'.

echo ${stringZ##a*C}     # abc
# Strip out longest match between 'a' and 'C'.
# You can parameterize the substrings.

X='a*C'

echo ${stringZ#$X}      # 123ABCabc
echo ${stringZ##$X}     # abc
                        # As above.
```

- **${string%substring}**
  Deletes shortest match of $substring from back of $string.

- **${string%%substring}**
  Deletes longest match of $substring from back of $string.

65. Substring Replacement

- **${string/substring/replacement}**
  Replace first match of $substring with $replacement.

- **${string//substring/replacement}**
  Replace all matches of $substring with $replacement.

```
#可以使用符*，会匹配最长的字符串，也可以使用[ ]

# Can the match and replacement strings be parameterized?
match=abc
repl=000
echo ${stringZ/$match/$repl}  # 000ABC123ABCabc
echo ${stringZ//$match/$repl} # 000ABC123ABC000
# Yes!

echo

# What happens if no $replacement string is supplied?
echo ${stringZ/abc}           # ABC123ABCabc
echo ${stringZ//abc}          # ABC123ABC
```

```
# A simple deletion takes place.
```

- **${string/#substring/replacement}**

  If **prefix of var matches Pattern**, then substitute Replacement for Pattern.

  If $substring matches front end of $string, substitute $replacement for $substring.

- **${string/%substring/replacement}**

  If **suffix of var matches Pattern**, then substitute Replacement for Pattern.

  If $substring matches back end of $string, substitute $replacement for $substring.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
                          # Replaces front-end match of 'abc' with 'XYZ'.

echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                          # Replaces back-end match of 'abc' with 'XYZ'.
```

66. Manipulating strings using awk

A Bash script may invoke the string manipulation facilities of awk as an alternative to using its built-in operations

```
String=23skidoo1
#      012345678    Bash
#      123456789    awk
# Note different string indexing system:
# Bash numbers first character of string as 0.
# Awk  numbers first character of string as 1.

echo ${String:2:4} # position 3 (0-1-2), 4 characters long
                                # skid

# The awk equivalent of ${string:pos:length} is substr(string,pos,length).
echo | awk '
{ print substr("'"${String}"'",3,4)      # skid
}
'
#  Piping an empty "echo" to awk gives it dummy input,
#+ and thus makes it unnecessary to supply a filename.
```

```
# And likewise:
echo | awk '
{ print index("'"${String}"'", "skid")      # 3
}                                   # (skid starts at position 3)
'   # The awk equivalent of "expr index" ...
```

67. Parameter Substitution

- **${parameter-default}, ${parameter:-default}**

  If parameter not set, use default.

  ${parameter-default} and ${parameter:-default} are almost equivalent. **The extra : makes a difference only whenparameter has been declared, but is null.**

  The default parameter construct finds use in providing "missing" command-line arguments in scripts.

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
```

- **${parameter=default}, ${parameter:=default}**

  If parameter not set, set it to default.

  Both forms nearly equivalent. The : makes a difference only when $parameter has been declared and is null, as above.

```
echo ${var=abc}   # abc
echo ${var=xyz}   # abc
# $var had already been set to abc, so it did not change.
```

- **${parameter+alt_value}, ${parameter:+alt_value}**

  If parameter set, use alt_value, else use null string.

  Both forms nearly equivalent. The : makes a difference only when parameter has been declared and is null.

- **${parameter?err_msg}, ${parameter:?err_msg}**

  If parameter set, use it, else print err_msg and abort the script with an exit status of 1.

  Both forms nearly equivalent. The : makes a difference only when parameter has been declared and is null, as above.

```
#  Check some of the system's environmental variables.
#  This is good preventative maintenance.
```

```
#  If, for example, $USER, the name of the person at the console, is not set,
#+ the machine will not recognize you.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}

:${NOTSET?“err_msg”} #script 将在这里退出，返回值为 1
```

68. Variable's Substitution

- **${!varprefix*}, ${!varprefix@}**
  Matches names of all previously declared variables beginning with varprefix.

```
xyz23=whatever
xyz24=

a=${!xyz*}       #  Expands to *names* of declared variables
#                + beginning with "xyz".
echo "a = $a"     #  a = xyz23 xyz24
a=${!xyz@}       #  Same as above.
echo "a = $a"     #  a = xyz23 xyz24

echo "---"

abc23=something_else
b=${!abc*}
echo "b = $b"     #  b = abc23
c=${!b}          #  Now, the more familiar type of indirect reference.
echo $c          #  something_else
```

Loops and Branches

69. for loop

```
for arg in [list]
do
```

```
 command(s)...
done
```

- **The argument list may contain wild cards**

```
for file in *
#               Bash performs filename expansion
#+              on expressions that globbing recognizes.
do
  ls -l "$file"  # Lists all files in $PWD (current directory).
  #  Recall that the wild card character "*" matches every filename,
  #+ however, in "globbing," it doesn't match dot-files.

  #  If the pattern matches no file, it is expanded to itself.
  #  To prevent this, set the nullglob option
  #+   (shopt -s nullglob).
  #  Thanks, S.C.
done


for file in [jx]*
do
  rm -f $file    # Removes only files beginning with "j" or "x" in $PWD.
  echo "Removed file \"$file\"".
done
```

- **Missing in [list] in a for loop**

```
for a
do
 echo -n "$a "
done
#  The 'in list' missing, therefore the loop operates on '$@'
#+ (command-line argument list, including whitespace).
```

- **The output of a for loop may be piped to a command or commands.**

```
for file in "$( find $directory -type l )"   # -type l = symbolic links
do
  echo "$file"
done | sort                              # Otherwise file list is unsorted.
#  Strictly speaking, a loop isn't really necessary here,
```

```
#+ since the output of the "find" command is expanded into a single word.
#  However, it's easy to understand and illustrative this way.

#  As Dominik 'Aeneas' Schnitzer points out,
#+ failing to quote  $( find $directory -type l )
#+ will choke on filenames with embedded whitespace.
#  containing whitespace.

OLDIFS=$IFS
IFS=:

for file in $(find $directory -type l -printf "%p$IFS")
do    #
      echo "$file"
      done|sort

# And, James "Mike" Conley suggests modifying Helou's code thusly:

OLDIFS=$IFS
IFS='' # Null IFS means no word breaks
for file in $( find $directory -type l )
do
 echo $file
 done | sort
```

- **A C-style for loop**

```
LIMIT=10
for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and naked "LIMIT"
do
 echo -n "$a "
done                          # A construct borrowed from ksh93.

# Let's use the C "comma operator" to increment two variables simultaneously.

for ((a=1, b=1; a <= LIMIT ; a++, b++))
do  # The comma concatenates operations.
 echo -n "$a-$b "
done
```

70. while loop

```
C-like syntax

((a = 1))      # a=1
# Double parentheses permit space when setting a variable, as in C.

while (( a <= LIMIT ))   #  Double parentheses,
do                       #+ and no "$" preceding variables.
 echo -n "$a "
 ((a += 1))            # let "a+=1"
 # Yes, indeed.
 # Double parentheses permit increment
```

By coupling the power of the read command with a while loop, we get the handy while read construct, useful for reading and parsing files.

```
cat $filename |   # Supply input from a file.
while read line   # As long as there is another line to read ...
do
 ...
done

######################
while read value   # Read one data point at a time.
  do
   rt=$(echo "scale=$SC; $rt + $value" | bc)
   (( ct++ ))
 done

 am=$(echo "scale=$SC; $rt / $ct" | bc)

 echo $am; return $ct   # This function "returns" TWO values!
 #  Caution: This little trick will not work if $ct > 255!
 #  To handle a larger number of data points,
 #+ simply comment out the "return $ct" above.
} <"$datafile"   # Feed in data file.
```

71. until loop

72. Nested Loops

A nested loop is a loop within a loop, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. Of course, a break within either the inner or outer loop would interrupt this process.

73. Loop Control

The **break** and **continue** loop control commands correspond exactly to their counterparts in other programming languages. The break command terminates the loop (breaks out of it), while continue causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

The **break** command may optionally take a parameter. A plain break terminates only the innermost loop in which it is embedded, but a **break N** breaks out of N levels of loop.

The **continue** command, similar to **break**, optionally takes a parameter. A plain continue cuts short the current iteration within its loop and begins the next. A **continue N** terminates all remaining iterations at its loop level and continues with the next iteration at the loop, N levels above.

```
for inner in 1 2 3 4 5 6 7 8 9 10  # inner loop
do

  if [[ "$inner" -eq 7 && "$outer" = "III" ]]
  then
    continue 2  # Continue at loop on 2nd level, that is "outer loop".
                # Replace above line with a simple "continue"
                # to see normal loop behavior.
  fi

  echo -n "$inner "  # 7 8 9 10 will not echo on "Group III."
done
```

74. Testing and Branching

case in .. esac

A case construct can filter strings for [globbing](#) patterns.

```
 -d|--debug)
*[!a-zA-Z]*|"")
```

Pattern-match lines may also start with a (left paren to give the layout a more structured appearance.

```
case $( arch ) in   # $( arch ) returns machine architecture.
  ( i386 ) echo "80386-based machine";;
# ^      ^
  ( i486 ) echo "80486-based machine";;
  ( i586 ) echo "Pentium-based machine";;
  ( i686 ) echo "Pentium2+-based machine";;
  (    * ) echo "Other type of machine";;
esac
```

75. select

The select construct, adopted from the Korn Shell, is yet another tool for building menus.

```
select variable [in list]
do
 command...
 break
done
```

This prompts the user to enter one of the choices presented in the variable list. Note that select uses the $PS3 prompt (#?) by default, but this may be changed.

If in list is omitted, then select uses the list of **command line arguments ($@)** passed to the script or the function containing the select construct.

```
PS3='Choose your favorite vegetable: ' # Sets the prompt string.
                                       # Otherwise it defaults to #? .

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break  # What happens if there is no 'break' here?
done
```

Command Substitution

76. Command substitute

Command substitution **reassigns the output of a command** or even multiple commands; it literally plugs the command output into another context.

Command substitution invokes a subshell.

77. `...` : backquotes

```
rm `cat filename`   # "filename" contains a list of files to delete.
# S. C. points out that "arg list too long" error might result.
# Better is              xargs rm -- < filename
# ( -- covers those cases where "filename" begins with a "-" )
```

78. (...) : alternative form of command substitution

```
textfile_listing2=$(ls *.txt)   # The alternative form of command substitution.
echo $textfile_listing2
# Same result.

# A possible problem with putting a list of files into a single string
# is that a newline may creep in.

# A safer way to assign a list of files to a parameter is with an array.
#      shopt -s nullglob    # If no match, filename expands to nothing.
#      textfile_listing=( *.txt )
```

79. attention

- **Command substitution may result in word splitting**

```
COMMAND `echo a b`     # 2 args: a and b

COMMAND "`echo a b`"   # 1 arg: "a b"

COMMAND `echo`         # no arg
```

```
COMMAND "`echo`"       # one empty arg
```

- **Even when there is no word splitting, command substitution can remove trailing newlines**

```
# cd "`pwd`"  # This should always work.
# However...

mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`"  # Error message:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # Works fine.
```

```
old_tty_setting=$(stty -g)   # Save old terminal setting.
echo "Hit a key "
stty -icanon -echo           # Disable "canonical" mode for terminal.
                             # Also, disable *local* echo.
key=$(dd bs=1 count=1 2> /dev/null)   # Using 'dd' to get a keypress.
stty "$old_tty_setting"      # Restore old setting.
echo "You hit ${#key} key."  # ${#variable} = number of characters in $variable
#
# Hit any key except RETURN, and the output is "You hit 1 key."
# Hit RETURN, and it's "You hit 0 key."
# The newline gets eaten in the command substitution.

#Code snippet by Stéphane Chazelas.
```

- **Using echo to output an unquoted variable set with command substitution removes trailing newlines characters from the output of the reassigned command(s). This can cause unpleasant surprises.**

```
dir_listing=`ls -l`
echo $dir_listing     # unquoted
```

```
# Expecting a nicely ordered directory listing.

# However, what you get is:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# The newlines disappeared.

echo "$dir_listing"   # quoted
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

- **Command substitution even permits setting a variable to the contents of a file, using either redirection or the cat command.**

```
variable1=`<file1`      #  Set "variable1" to contents of "file1".
variable2=`cat file2`   #  Set "variable2" to contents of "file2".
                        #  This, however, forks a new process,
                        #+ so the line of code executes slower than the above version.

#  Note that the variables may contain embedded whitespace,
#+ or even (horrors), control characters.

#  It is not necessary to explicitly assign a variable.
echo "` <$0`"           # Echoes the script itself to stdout.
```

Attention: Do not set a variable to the contents of a long text file unless you have a very good reason for doing so. Do not set a variable to the contents of a binary file, even as a joke.

- **Command substitution permits setting a variable to the output of a loop. The key to this is grabbing the output of an echocommand within the loop**

```
variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"              #  The 'echo' command is critical
done`                       #+ to command substitution here.

echo "variable1 = $variable1"  # variable1 = 12345
```

```
i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"               # Again, the necessary 'echo'.
  let "i += 1"               # Increment.
Done`
echo "variable2 = $variable2"  # variable2 = 0123456789
```

80. $(...)

The $(...) form of command substitution treats a double backslash in a different way than `...`.

```
bash$ echo `echo \\`

bash$ echo $(echo \\)
\
```

The $(...) form of command substitution permits nesting.

```
word_count=$( wc -w $(echo * | awk '{print $8}') )
```

In fact, **nesting with backticks is also possible**, but only by escaping the inner backticks, as John Default points out.

```
word_count=` wc -w \`echo * | awk '{print $8}'\` `
```

Arithmetic Expansion

81. Arithmetic Expansion

Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations in scripts. Translating a string into a numerical expression is relatively straightforward using backticks, double parentheses, or let.

- **Arithmetic expansion with backticks (often used in conjunction with expr)**

```
z=`expr $z + 3`          # The 'expr' command performs the expansion.
```

- **Arithmetic expansion with `double parentheses`, and using let**
  The use of **backticks** (backquotes) in arithmetic expansion has been superseded by double parentheses -- **((...))** and **$((...))** -- and also by the very convenient **let** construction.

Commands

Internal Commands and Builtins

82. Internal Commands and Builtins

A **builtin** is a command contained within the Bash tool set, literally **built in**. This is either for performance reasons -- builtins execute **faster** than external commands, which usually require forking off a separate process -- or because a particular builtin needs direct access to the **shell internals.**

Generally, a Bash builtin does not fork a subprocess when it executes within a script. An external system command or filter in a script usually will fork a subprocess.

83. echo

An echo requires the -e option to print escaped characters. See Example 5-2.

Normally, each echo command prints a terminal newline, but the -n option suppresses this.

84. printf

The printf, formatted print, command is an enhanced echo. It is a limited variant of the C language printf() library function, and its syntax is somewhat different.

```
printf format-string... parameter...
```

```
declare -r PI=3.14159265358979     # Read-only variable, i.e., a constant.
printf "Pi to 2 decimal places = %1.2f" $PI
```

```
printf "Pi to 9 decimal places = %1.9f" $PI  # It even rounds off correctly.
printf "\n"                                  # Prints a line feed,

error()
{
  printf "$@" >&2
  # Formats positional params passed, and sends them to stderr.
  echo
  exit $E_BADDIR
}

cd $var || error $"Can't cd to %s." "$var"
```

85. read

"Reads" the value of a variable from stdin, that is, interactively fetches input from the keyboard. The -a option lets read get array variables (see Example 27-6).

- A **read** without an associated variable assigns its input to the dedicated variable **$REPLY**.

```
In some instances, you might wish to discard the first value read.
In such cases, simply ignore the $REPLY variable.

{ # Code block.
read          # Line 1, to be discarded.
read line2    # Line 2, saved in variable.
 } <$0
```

```
# A single 'read' statement can set multiple variables.
echo -n "Enter the values of variables 'var2' and 'var3' "
echo =n "(separated by a space or tab): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
#  If you input only one value,
#+ the other variable(s) will remain unset (null).
```

- Normally, inputting **a \ suppresses a newline** during input to a read. The **-r** option causes an inputted \ to be interpreted **literally**.

```
read var1    # The "\" suppresses the newline, when reading $var1.
             #     first line \
```

```
        #     second line
read -r var2  # The -r option causes the "\" to be read literally.
        #     first line \
```

- The **read** command has some **interesting options** that permit echoing a **prompt** and even reading **keystrokes** without hitting **ENTER**.

```
# Read a keypress without hitting ENTER.

read -s -n1 -p "Hit a key " keypress
echo; echo "Keypress was "\"$keypress\""."

# -s option means do not echo input.
# -n N option means accept only N characters of input.
# -p option means echo the following prompt before reading input.

# Using these options is tricky, since they need to be in the correct order.
```

- The **-n** option to **read** also allows detection of the **arrow keys** and certain of the other unusual keys.

  The -n option to **read** will not detect the **ENTER** (newline) key.

  Example 15-6. Detecting the arrow keys

- The **-t** option to **read** permits timed input (see Example 9-4 and Example A-41).

- The **read** command may also "read" its variable value from a file redirected to stdin. If the file contains more than one line, only the first line is assigned to the variable. If **read** has more than one **parameter**, then each of these variables gets assigned a successive **whitespace-delineated** string. **Caution!**

```
IFS= " : "
read var1 var2 var3 < /etc/passwd
echo "$var1"  #root
echo "$var2" # x
echo "$var3"  # 0 0 root /root /bin/bash

# Note non-intuitive behavior of "read" here.
# 1) Rewinds back to the beginning of input file.
# 2) Each variable is now set to a corresponding string,
#    separated by whitespace, rather than to an entire line of text.
# 3) The final variable gets the remainder of the line.
# 4) If there are more variables to be set than whitespace-terminated strings
```

```
#    on the first line of the file, then the excess variables remain empty.


# How to resolve the above problem with a loop:
while read line
do
  echo "$line"
done <data-file


#  Setting the $IFS variable within the loop itself
#+ eliminates the need for storing the original $IFS
#+ in a temporary variable.
#  Thanks, Dim Segebart, for pointing this out.
echo "-----------------------------------------------"
echo "List of all users:"

while IFS=: read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O redirection.

#The variable IFS is set within the loop/subshell,
#+but its value does not persist outside the loop.
```

- Piping output to a read, using **echo** to set variables will fail
  Yet, piping the output of **cat** seems to work.

```
cat file1 file2 |
while read line
do
echo $line
done
```

Example 15-8. Problems reading from a pipe

```
find $1 \( -name "*$2" -o -name ".*$2" \) -print |
while read f; do
. . .
```

86. cd

The **-P** (**physical**) option to cd causes it to ignore **symbolic links**.

87. pushd, popd, dirs (用栈来存放工作目录)

This command set is a mechanism for **bookmarking** working directories, a means of moving back and forth through directories in an **orderly** manner. A pushdown **stac**k is used to keep track of directory names. Options allow various manipulations of the **directory stack.**

- **pushd dir-name：**pushes the path dir-name onto the directory **stack** and simultaneously changes the current working directory to **dir-name.**
- **popd ：**removes (pops) the top directory path name off the directory stack and simultaneously changes the current working directory to that directory popped from the stack.
- **dirs ：**lists the contents of the directory stack (compare this with the **$DIRSTACK** variable). A successful **pushd** or **popd** will automatically invoke **dirs.**
- Note that the implicit **$DIRSTACK array** variable, accessible from within a script, holds the contents of the directory stack.
  echo ${DIRSTACK[*]}

88. let

The **let** command carries out arithmetic operations on variables. In many cases, it functions as a less complex version of **expr.**

- The **let** command can, in certain contexts, return a surprising **exit status.**

```
var=0
echo $?    # 0
        # As expected.

let var++
echo $?    # 1
        # The command was successful, so why isn't $?=0 ???
        # Anomaly!

let var++
echo $?    # 0
        # As expected.
#This is part of the design spec for 'let' . . .
# "If the last ARG evaluates to 0, let returns 1;
#  let returns 0 otherwise." ['help let']
```

89. eval

- Combines the arguments in an expression or list of expressions and evaluates them. Any variables within the expression are **expanded**. The net result is to **convert a string into a command.**
- Each invocation of eval forces a **re-evaluation** of its arguments.

```
a='$b'
b='$c'
c=d

echo $a           # $b
                  # First level.
eval echo $a      # $c
                  # Second level.
eval eval echo $a   # d
                  # Third level.
```

```
# Now, showing how to do something useful with "eval" . . .
# (Thank you, E. Choroba!)

version=3.4     #  Can we split the version into major and minor
                #+ part in one command?
echo "version = $version"
eval major=${version/./;minor=}     #  Replaces '.' in version by ';minor='
                                    #  The substitution yields '3; minor=4'
                                    #+ so eval does minor=4, major=3
echo Major: $major, minor: $minor   #  Major: 3, minor: 4
```

```
killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
#                  -------- process ID of ppp -------

$killppp                  # This variable is now a command.
```

90. set

- The set command changes the value of internal script variables/options. One use for this is to toggle option flags which help determine the behavior of the script. Another application for it is to reset the positional parameters that a script sees as the result of a command (set `command`). The script can then parse the fields of the command output.

- Using **set** with the **--** option explicitly **assigns** the contents of a variable to the **positional parameters**. If no variable follows the **--** it **unsets** the **positional parameters.**

91. unset

The **unset** command deletes a shell variable, effectively setting it to **null.** Note that this command does not affect **positional parameters**.

92. export

The export command makes available variables to all **child processes** of the running script or shell. One important use of the export command is in `startup files`, to initialize and make accessible `environmental variables` to **subsequent** user processes.
Unfortunately, `there is no way to export variables back to the parent process`, to the process that called or invoked the script or shell.

93. declare, typeset

94. readonly

Same as **declare -r**, sets a variable as read-only, or, in effect, as a constant. Attempts to change the variable fail with an error message. This is the shell analog of the C language **const** type qualifier.

95. getopts

- This powerful tool parses command-line arguments passed to the script. This is the Bash analog of the **getopt** external command and the getopt library function familiar to C programmers. It permits passing and concatenating multiple options and associated arguments to a script (for example scriptname -abc -e /usr/local).

- The **getopts** construct uses two implicit variables. **$OPTIND** is the **argument pointer** (**OPT**ion **IN**Dex) and **$OPTARG** (**OPT**ion **ARG**ument) the (optional) argument attached to an option. A **colon**(分号) following the option name in the declaration tags that option as **having an associated argument.**

- A **getopts** construct usually comes packaged in a **while loop**, which processes the options and arguments one at a time, then increments the implicit **$OPTIND** variable to point to the next.

- The arguments passed from the command-line to the script must be preceded by a

dash (-). It is the **prefixed -** that lets getopts recognize c**ommand-line arguments** as options. In fact, **getopts** will not process arguments without the prefixed -, and will **terminate** option processing at the first argument encountered lacking them.

- The **getopts** template differs slightly from the standard **while loop**, in that it **lacks condition brackets**.
  The getopts construct is a highly functional replacement for the traditional **getopt** external command.

```
while getopts ":mnopq:rs" Option
do
 case $Option in
   m     ) echo "Scenario #1: option -m-   [OPTIND=${OPTIND}]";;
   n | o ) echo "Scenario #2: option -$Option-   [OPTIND=${OPTIND}]";;
   p     ) echo "Scenario #3: option -p-   [OPTIND=${OPTIND}]";;
   q     ) echo "Scenario #4: option -q-\
               with argument \"$OPTARG\"   [OPTIND=${OPTIND}]";;
   #  Note that option 'q' must have an associated argument,
   #+ otherwise it falls through to the default.
   r | s ) echo "Scenario #5: option -$Option-";;
   *     ) echo "Unimplemented option chosen.";;   # Default.
 esac
done
shift $(($OPTIND - 1))
#  Decrements the argument pointer so it points to next argument.
#  $1 now references the first non-option item supplied on the command-line
#+ if one exists.
```

96. source, . (dot command)

- This command, when invoked from the **command-line**, executes a script. **Within a script**, a source file-name **loads the file file-name**. Sourcing a file (dot-command) **imports code into the script**, appending to the script (same effect as the **#include** directive in a C program). The net result is the same as if the "sourced" lines of code were physically present in the body of the script. This is useful in situations when multiple scripts use a common data file or function library.

- If the sourced file is itself **an executable script**, then it will run, then **return control to the script that called it**. A sourced executable script may use a **return** for this purpose.

- Arguments may be (optionally) passed to the sourced file as `positional parameters`**.**

```
source $filename $arg1 arg2
```

- It is even possible for a script to **source itself**, though this does not seem to have any practical applications.

97. exit

If a script terminates with an exit **lacking an argumen**t, the exit status of the script is the exit status of the **last command** executed in the script, not counting the exit. This is equivalent to an **exit $?**.

98. exec

This shell builtin **replaces the current process** with a specified command. Normally, when the shell encounters a command, it **forks off** a **child process** to actually execute the command. Using the exec builtin, the shell does not fork, and the command exec'ed **replaces the shell**. **When used in a script, therefore, it forces an exit from the script when the exec'ed command terminates.** Unless the exec is used to **reassign file descriptors.**

```
#!/bin/bash

exec echo "Exiting \"$0\"."   # Exit from script here.

# ---------------------------------
# The following lines never execute.

echo "This echo will never echo."

exit 99                    #  This script will not exit here.
                           #  Check exit value after script terminates
                           #+ with an 'echo $?'.
                           #  It will *not* be 99.
```

99. shopt

This command permits **changing shell options** on the fly (see Example 25-1 and Example 25-2). It often appears in the Bash **startup files**, but also has its uses in scripts. Needs version 2 or later of Bash.

```
shopt -s cdspell
# Allows minor misspelling of directory names with 'cd' , 相当于纠错功能
# Option -s sets, -u unsets.

cd /hpme  # Oops! Mistyped '/home'.
pwd       # /home
```

```
        # The shell corrected the misspelling.
shopt -u cdspell
```

100. caller

Putting a **caller** command **inside a function** echoes to stdout information about the caller of that function.

```
#!/bin/bash

function1 ()
{
  # Inside function1 ().
  caller 0   # Tell me about it.
}

function1    # Line 9 of script.

# 9 main test.sh
#     Line number that the function was called from.
#   Invoked from "main" part of script.
#      Name of calling script.

caller 0     # Has no effect because it's not inside a function.
```

A **caller** command can also **return caller information** from a script **sourced** within another script. Analogous to a function, this is a "**subroutine call**."

You may find this command useful in debugging.

101. true

A command that returns a **successful** (**zero**) **exit status**, but **does nothing** else.

102. false

A command that returns an **unsuccessful exit status**, but does nothing else.

103. type

Similar to the **which** external command, **type** cmd identifies "cmd." Unlike which, type is a Bash builtin. The useful **-a** option to type identifies **keywords** and **builtins**, and also locates **system commands** with identical names.

104. hash [cmds]

Records the **path name** of specified commands -- in the shell **hash table** -- so the

shell or script will not need to search the **$PATH** on subsequent calls to those commands. When hash is called with no arguments, it simply lists the commands that have been hashed. The **-r** option **resets** the **hash table**.

- Hashing is a method of creating **lookup keys** for data stored in a **table**. The data items themselves are "scrambled" to create **keys**, using one of a number of simple mathematical algorithms (methods, or recipes).
- An advantage of hashing is that it is **fast**. A disadvantage is that **collisions** -- where a single key maps to more than one data item -- are possible.

105. bind

The **bind** builtin displays or modifies readline key **bindings.**

106. help

Gets a short **usage** summary of a shell builtin. This is the **counterpart** to **whatis**, but for builtins. The display of helpinformation got a much-needed update in the version 4 release of Bash.

```
help exit
```

Job Control Commands

107. jobs

Lists the **jobs** running in the **background**, giving the job number. Not as useful as **ps**.

Attention: It is all too easy to confuse **jobs** and **processes**. Certain builtins, such as **kill**, **disown**, and **wait** accept either a job number or a process number as an argument. The **fg**, **bg** and **jobs** commands accept only a **job number**.

108. disown

Remove job(s) from the shell's table of active jobs.

109. fg, bg

- The **fg** command switches a job running in the background into the foreground.
- The **bg** command restarts a suspended job, and runs it in the background. If no job number is specified, then the fg or bg command acts upon the **currently running job.**

110. wait

Suspend script execution until **all jobs** running in background have **terminated**, or **until** the job number or process ID specified **as an option terminates**. **Returns** the exit status of **waited-for command.**

- You may use the wait command to prevent a script from exiting before a background job finishes executing (this would create a dreaded **orphan process**).

```
updatedb /usr &     # Must be run as root.

wait
# Don't run the rest of the script until 'updatedb' finished.
# You want the the database updated before looking up the file name.
locate $1

#  Without the 'wait' command, in the worse case scenario,
#+ the script would exit while 'updatedb' was still running,
#+ leaving it as an orphan process.

exit 0
```

- Optionally, wait can take a job identifier as an argument, for example, **wait %1** or **wait $PPID**. See the **job id table**.

- **Job id table**

Notation	Meaning
%N	Job number [N]
%S	Invocation (command-line) of job begins with string *S*
%?S	Invocation (command-line) of job contains within it string *S*
%%	"current" job (last job stopped in foreground or started in background)
%+	"current" job (last job stopped in foreground or started in background)
%-	Last job
$!	Last background process

- Attention: Within a script, running a command in the background with an ampersand (&) may cause the script to **hang until ENTER is hit**. This seems to occur with **commands that write to stdout.** It can be a major annoyance.

```
#!/bin/bash
# test.sh
```

```
ls -l &
echo "Done."

#wait
```

Placing a wait after the background command seems to remedy this.

Redirecting the output of the command to a file or even to /dev/null also takes care of this problem.

111. suspend

This has a similar effect to Control-Z, but it **suspends the shell** (**the shell's parent process should resume it at an appropriate time**).

112. logout

Exit a login shell, optionally specifying an exit status.

113. times

Gives statistics on the system time elapsed when executing commands, in the following form:

```
0m0.020s 0m0.020s
```

This **capability** is of relatively limited value, since it is not common to profile and benchmark shell scripts.

114. kill

Forcibly terminate a process by sending it an appropriate terminate signal (see Example 17-6).

- **kill -l** lists all the **signals** (as does the file /usr/include/asm/signal.h). A **kill -9** is a sure kill, which will usually terminate a process that **stubbornly** refuses to die with a plain kill. Sometimes, a **kill -15** works. A **zombie proces**s, that is, a child process that has terminated, but that the parent process has not (yet) killed, cannot be killed by a logged-on user -- you can't kill something that is already dead -- but init will generally clean it up sooner or later.

115. killall

The killall command kills a running process **by name**, **rather than by process ID**. If there are multiple instances of a particular command running, then doing a killall on that command will terminate them all.

116. command

The command directive **disables aliases and functions** for the command immediately

following it.

```
command ls
```

117. builtin

Invoking **builtin BUILTIN_COMMAND** runs the command BUILTIN_COMMAND as a **shell builtin**, **temporarily** disabling both functions and external system commands with the same name.

118. enable

- **This either enables or disables a shell builtin command.** As an example, **enable -n kill** disables the shell builtin kill, so that when Bash subsequently encounters kill, it invokes the external command **/bin/kill**. **enable kill** enable it.
- The **-a** option to enable lists all the shell builtins, indicating whether or not they are enabled. The **-f filename** option lets enableload a builtin as a shared library (DLL) module from a properly compiled object file.

119. autoload

This is a port to Bash of the ksh autoloader. With **autoload** in place, a function with an autoload declaration will load from an external file at its first invocation. This saves system resources.

- **Note that** autoload is not a part of the core Bash installation. It needs to be loaded in with enable -f (see above).

External Filters, Programs and Commands

120. ls、cat、tac

```
ls -lRF
cat -sv
```

121. rev: 翻转每行字符

reverses each line of a file, and outputs to **stdou**t. This does not have the same effect as tac, as **it preserves the order of the lines**, but **flips** each one around (mirror image).

122. cp、mv、rm

cp: Particularly useful are the **-a** archive flag (for copying an entire directory tree), the **-u update flag** (which prevents overwriting identically-named newer files), and the -r and -R recursive flags.

123. rmdir、mkdir

124. chmod、chattr

125. ln

Which type of link to use?

As John Macdonald explains it:

Both of these [types of links] provide a certain measure of dual reference -- if you edit the contents of the file using any name, your changes will affect both the original name and either a hard or soft new name. The differences between them occurs when you work at a higher level. The advantage of a hard link is that the new name is totally independent of the old name -- if you remove or rename the old name, that does not affect the hard link, which continues to point to the data while it would leave a soft link hanging pointing to the old name which is no longer there. The advantage of a soft link is that it can refer to a different file system (since it is just a reference to a file name, not to actual data). And, unlike a hard link, a symbolic link can refer to a directory.

Links give the ability to **invoke a script** (or any other type of executable) with **multiple names**, and having that script behave according to **how it was invoked. $0 is different.**

126. find

-exec COMMAND \;

-maxdepth 搜索目录深度

-mindepth

127. xargs

A filter for feeding arguments to a command, and also a tool for assembling the commands themselves. It **breaks a data stream** into small enough **chunks** for filters and

commands to process. Consider it as a powerful replacement for backquotes. In situations where command substitution fails with a too many arguments error, substituting **xargs** often works.

- Normally, **xargs** reads from **stdin** or from a **pipe**, but it can also be given the output of a file.
- The default command for **xargs** is **echo**. This means that input piped to **xargs** may have **linefeeds** and other **whitespace** characters stripped out.
- -p
  -l
  -n: limits to NN the number of arguments passed.
  -0: in combination with **find -print0** or **grep -lZ**. This allows handling arguments containing **whitespace or quotes.**
  -P: permits running **processes** in **parallel**. This speeds up execution in a machine with a multicore CPU.
- As in **find,** a **curly bracket** pair serves as a placeholder for **replacement text.**

128. expr

expr 5 * 3 #符号两边要有空格

- The multiplication operator must be **escaped** when used in an arithmetic expression with **expr**.
  Logical Operators (Returns 1 if true, 0 if false **)**
- a=3
  b=`expr $a **\>** 10` # b=0

129. date

generate random integers.

```
date +%N | sed -e 's/000$//' -e 's/^0//'
#  Strip off leading and trailing zeroes, if present.
#  Length of generated integer depends on
#+ how many zeroes stripped off.
```

130. zdump

Time zone dump: echoes the time in a specified time zone.

131. time COMMAND

Outputs verbose timing statistics for executing a command.

132. touch

Utility for **updating** access/modification times of a file to current system time or other specified time, but also useful for **creating a new file**. The command touch zzz will create a new file of zero length, named zzz, assuming that zzz did not previously exist. Time-stamping empty files in this way is useful for storing date information, for example in keeping track of modification times on a project.

- Before doing a **cp -u** (copy/update), use **touch** to update the time stamp of files you don't wish overwritten.

133. at、batch

The at job control command executes a given set of commands at a specified time. Superficially, it resembles **cron**, however,at is chiefly useful for one-time execution of a command set.

at 2pm January 15 prompts for a set of commands to execute at that time. These commands should be **shell-script compatible**, since, for all practical purposes, the user is typing in an executable shell script a line at a time. Input terminates with a **Ctl-D**. Using either the **-f** option or **input redirection (<)**, at reads a command list **from a file.** This file is an executable shell script, though it should, of course, be **non-interactive**. Particularly clever is including the **run-parts** command in the file to execute a different set of scripts.

134. cal

135. sleep

This is the shell equivalent of a wait loop. It **pauses** for a specified number of seconds, doing nothing. It can be useful for timing or in processes running in the background, checking for a specific event every so often (polling), as in Example 32-6.

```
sleep 3     # Pauses 3 seconds.
sleep 3 h   # Pauses 3 hours!
```

The **watch** command may be a better choice than sleep for running commands **at timed intervals.**

136. usleep (微秒)

Microsleep (the u may be read as the Greek mu, or micro- prefix). This is the same as sleep, above, but "sleeps" in microsecond intervals. It can be used for fine-grained timing, or for polling an ongoing process at very frequent intervals.

```
usleep 30     # Pauses 30 microseconds.
```

Attention: The usleep command does not provide particularly accurate timing, and is

therefore unsuitable for critical timing loops.

137. hwclock, clock

The **hwclock** command accesses or adjusts the machine's hardware clock. Some options require root privileges. The**/etc/rc.d/rc.sysinit** startup file uses hwclock to set the system time from the hardware clock at bootup.

The **clock** command is a synonym for hwclock.

Text Processing Commands

138. sort、tsort

139. uniq

```
sort testfile | uniq -c | sort -nr
```

The **sort INPUTFILE | uniq -c | sort -nr** command string produces a **frequency** of occurrence listing on the INPUTFILE file (the -nroptions to sort cause a reverse numerical sort). This template finds use in analysis of log files and dictionary lists, and wherever the lexical structure of a document needs to be examined.

140. expand, unexpand

The **expand** filter converts **tabs to spaces.** It is often used in a pipe.

The **unexpand** filter converts s**paces to tabs**. This reverses the effect of expand.

141. cut

A tool for extracting fields from files. It is similar to the **print $N** command set in **awk**, but more limited. It may be simpler to use cut in a script than awk. Particularly important are the **-d (delimiter)** and **-f (field specifier)** options.

- **cut -d ' ' -f2,3 filename** is equivalent to **awk -F'[ ]' '{ print $2, $3 }' filename**
- It is even possible to specify a **linefeed** as a **delimiter**. The trick is to actually embed a linefeed (RETURN) in the command sequence.

```
bash$ cut -d'
```

```
' -f3,7,19 testfile
This is line 3 of testfile.
 This is line 7 of testfile.
 This is line 19 of testfile.
```

142. paste

Tool for **merging** together different files into a single, multi-column file. In combination with **cut**, useful for creating system log files.

143. join

- Consider this a special-purpose cousin of **paste**. This powerful utility allows merging two files in a **meaningful fashion**, which essentially creates a simple version of a **relational database.**
- The **join** command operates on exactly two files, but pastes together only those lines with a **common tagged field** (usually a numerical label), and writes the result to **stdout.** The files to be joined should be sorted according to the tagged field for the matchups to work properly.
- -t
  -1
  -2

144. head

Generating 10-digit random numbers

```
head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
```

145. tail

To list a specific line of a text file, pipe the output of head to tail -n 1. For example **head -n 8 database.txt | tail -n 1** lists the 8th line of the file database.txt.

146. grep

g/re/p -- global - regular expression – print.

- **-z**
  -**q:** suppress output
- To force grep to show the filename when searching only one target file, simply give /dev/null as the second file.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
```

```
osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

```
grep -q "$word" "$filename"    #  The "-q" option
                               #+ causes nothing to echo to stdout.
if [ $? -eq $SUCCESS ]
```

147. egrep、fgrep

- **egrep** -- **extended** grep -- is the same as grep -E. This uses a somewhat different, extended set of Regular Expressions, which can make the search a bit more flexible. It also allows the boolean | (or) operator.
- **fgrep** -- **fast** grep -- is the same as grep -F. It does a literal string search (no Regular Expressions), which generally speeds things up a bit.

148. agrep

agrep (approximate grep) extends the capabilities of grep to approximate matching. The search string may differ by a specified number of characters from the resulting matches. This utility is not part of the core Linux distribution.

- To search compressed files, use **zgrep**, **zegrep**, or **zfgrep**. These also work on non-compressed files, though slower than plain grep, egrep, fgrep. They are handy for searching through a mixed set of files, some compressed, some not.
- To search bzipped files, use **bzgrep**.

149. look

The command **look** works like **grep**, but does a lookup on a "dictionary," a sorted word list. By default, look searches for a match in /usr/dict/words, but a different dictionary file may be specified.

150. sed

Non-interactive "stream editor", permits using many ex commands in batch mode. It finds many uses in shell scripts.

151. awk

Programmable file extractor and formatter, good for manipulating and/or extracting fields (columns) in structured text files. Its syntax is similar to C.

152. wc

wc gives a "word count" on a file or I/O stream:

Using **wc** to count how many .txt files are in current working directory:

```
$ ls *.txt | wc -l
#  Will work as long as none of the "*.txt" files
#+ have a linefeed embedded in their name.

#  Alternative ways of doing this are:
#     find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#     (shopt -s nullglob; set -- *.txt; echo $#)
```

153. tr

character translation filter.

- Either tr "A-Z" "*" <filename or tr A-Z * <filename changes all the uppercase letters in filename to asterisks (writes to stdout). On some systems this may not work, but tr A-Z '[**]' will.
- The **-d** option **deletes** a range of characters.

```
echo "abcdef"                # abcdef
echo "abcdef" | tr -d b-d     # aef
```

- The **--squeeze-repeats** (or -s) option deletes all but the first instance of a string of consecutive characters. This option is useful for removing excess whitespace.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

- The **-c** "**complement**" option inverts the character set to match. With this option, tr acts only upon those characters notmatching the specified set.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

- Note that tr recognizes **POSIX character classes**.

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

```
cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" goes to "n", "b" to "o" ...
#  The   cat "$@"   construct
#+ permits input either from stdin or from files.
```

154. fold

A filter that **wraps lines** of input to a specified width. This is especially useful with the **-s** option, which breaks lines at word spaces (see Example 16-26 and Example A-1).

155. fmt

Simple-minded file **formatter**, used as a **filter** in a pipe to "**wrap**" long lines of text

output.

156. col

This deceptively named filter removes reverse line feeds from an input stream. It also attempts to replace whitespace with equivalent tabs. The chief use of col is in filtering the output from certain text processing utilities, such as **groff** and **tbl**.

157. column

Column formatter. This filter transforms list-type text output into a "pretty-printed" table by inserting tabs at appropriate places.

```
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n"
\
; ls -l | sed 1d) | column -t
```

158. colrm

Column **removal** filter. This removes columns (characters) from a file and writes the file, lacking the range of specified columns, back to stdout. **colrm 2 4 <filename** removes the second through fourth characters from each line of the text filefilename.

- Attention: If the file contains tabs or nonprintable characters, this may cause unpredictable behavior. In such cases, consider using expand and unexpand in a pipe preceding colrm.（一个字符就是一个 column)

159. nl

Line **numbering** filter: nl filename lists filename to stdout, but inserts consecutive numbers at the beginning of each non-blank line. If filename omitted, operates on stdin.

160. pr

- **Print** formatting filter. This will paginate files (or stdout) into sections suitable for hard copy printing or viewing on screen. Various options permit row and column manipulation, joining lines, setting margins, numbering lines, adding page headers, and merging files, among other things. The **pr** command combines much of the functionality of nl, paste, fold, column, andexpand.
- **pr -o 5 --width=65 fileZZZ | more** gives a nice paginated listing to screen of fileZZZ with margins set at 5 and 65.
- A particularly useful option is **-d**, forcing double-spacing (same effect as sed -G).

161. gettext

The GNU gettext package is a set of utilities for **localizing** and **translating** the text output of programs into **foreign languages.** While originally intended for C programs, it now supports quite a number of programming and scripting languages.

- The gettext program works on shell scripts. See the info page.

162. msgfmt

A program for generating binary message catalogs. It is used for **localization**.

163. iconv

A utility for converting file(s) to a different encoding (character set). Its chief use is for **localization**.

164. TeX、gs

- **TeX** and **Postscript** are text markup languages used for preparing copy for printing or formatted video display.
- **TeX** is Donald Knuth's elaborate typsetting system. It is often convenient to write a shell script encapsulating all the options and arguments passed to one of these markup languages.
- **Ghostscript** (**gs**) is a GPL-ed Postscript interpreter.

165. texexec

Utility for processing **TeX** and **pdf** files. Found in /usr/bin on many Linux distros, it is actually a **shell wrapper** that calls **Perl** to invoke **Tex**.

```
texexec --pdfarrange --result=Concatenated.pdf *pdf

#  Concatenates all the pdf files in the current working directory
#+ into the merged file, Concatenated.pdf . . .
#  (The --pdfarrange option repaginates a pdf file. See also --pdfcombine.)
#  The above command-line could be parameterized and put into a shell script.
```

166. ecscript

- Utility for converting **plain text** file to **PostScript**
- For example, **enscript filename.txt -p filename.ps** produces the PostScript output file filename.ps.

167. groff, tbl, eqn

- Yet another text markup and display formatting language is **groff.** This is the enhanced GNU version of the venerable UNIXroff/troff display and typesetting package. Manpages use **groff**.
- The **tbl** table processing utility is considered part of groff, as its function is to convert table markup into groff commands.
- The **eqn** equation processing utility is likewise part of groff, and its function is to convert equation markup into groffcommands.

168. lex、yacc

- The **lex** lexical analyzer produces programs for pattern matching. This has been replaced by the nonproprietary **flex** on Linux systems.
- The **yacc** utility creates a parser based on a set of specifications. This has been replaced by the nonproprietary **bison** on Linux systems.

File and Archiving Commands

169. tar

170. shar

171. ar

Creation and manipulation utility for archives, mainly used for binary object file libraries.

172. rpm

173. cpio

This specialized archiving copy command (**copy input and output**) is rarely seen any more, having been supplanted by **tar/gzi**p. It still has its uses, such as moving a directory tree. With an appropriate block size (for copying) specified, it can be appreciably faster than tar.

```
find "$source" -depth | cpio -admvp "$destination"
```

174. rpm2cpio

This command extracts a cpio archive from an rpm one.

```
rpm2cpio < $1 > $TEMPFILE              #  Converts rpm archive into
                                  #+ cpio archive.
cpio --make-directories -F $TEMPFILE -i  #  Unpacks cpio archive.
rm -f $TEMPFILE                        #  Deletes cpio archive.
```

175. pax

The pax portable archive exchange toolkit facilitates periodic file backups and is designed to be cross-compatible between various flavors of UNIX. It was designed to replace tar and cpio.

```
pax -wf daily_backup.pax ~/linux-server/files
#  Creates a tar archive of all files in the target directory.
#  Note that the options to pax must be in the correct order --
#+ pax -fw     has an entirely different effect.

pax -f daily_backup.pax
#  Lists the files in the archive.

pax -rf daily_backup.pax ~/bsd-server/files
#  Restores the backed-up files from the Linux machine
#+ onto a BSD one.
```

176. gzip、bzip2、compress、sq、zip

177. unarc, unarj, unrar

These Linux utilities permit unpacking archives compressed with the DOS arc.exe, arj.exe, and rar.exe programs.

178. lzma, unlzma, lzcat

Highly efficient Lempel-Ziv-Markov compression. The syntax of lzma is similar to that of gzip. The 7-zip Website has more information.

179. file

180. which

which command gives the full path to "command." This is useful for finding out

whether a particular command or utility is installed on the system.

181. whatis

whatis command looks up "command" in the whatis database. This is useful for identifying system commands and important configuration files. Consider it a simplified man command.

182. vdir

Show a detailed directory listing. The effect is similar to ls -lb.

183. locate, slocate

The **locate** command searches for files using a database stored for just that purpose. The **slocate** command is the secure version of locate (which may be aliased to slocate).

184. getfacl, setfacl

These commands retrieve or set the **file access control lis**t -- the owner, group, and file permissions.

185. readlink

Disclose the file that a **symbolic** link points to.

186. strings

Use the **strings** command to find **printable** strings in a **binary** or **data** file. It will list sequences of printable characters found in the target file. This might be handy for a quick 'n dirty examination of a core dump or for looking at an unknown graphic image file (**strings image-file** | more might show something like JFIF, which would identify the file as a jpeg graphic). In a script, you would probably parse the output of strings with grep or sed. See Example 11-7 and Example 11-9.

187. diff、patch

- **diff:** flexible file comparison utility. It compares the target files line-by-line sequentially. In some applications, such as comparing word dictionaries, it may be helpful to filter the files through sort and uniq before piping them to diff. **diff file-1 file-2** outputs the lines in the files that differ, with carets showing which file each particular line belongs to.
- Attention: The diff command returns an exit status of 0 if the compared files are

identical, and 1 if they differ. This permits use of diff in a **test construct** within a shell script .

- A common use for diff is generating difference files to be used with **patch** The -e option outputs files suitable for ed or exscripts.
- **patch:** flexible versioning utility. Given a difference file generated by **diff**, patch can upgrade a previous version of a package to a newer version. It is much more convenient to distribute a relatively small "diff" file than the entire body of a newly revised package. Kernel "patches" have become the preferred method of distributing the frequent releases of the Linux kernel.

188. diff3, merge

- An **extended** version of diff that compares **three files** at a time. This command returns an exit value of 0 upon successful execution, but unfortunately this gives no information about the results of the comparison.
- The **merge (**3-way file merge) command is an interesting adjunct to diff3. Its syntax is **merge Mergefile file1 file2**. The result is to output to Mergefile the changes that lead from file1 to file2. Consider this command a stripped-down version of patch.

189. sdiff

Compare and/or **edit** two files in order to merge them into an output file. Because of its **interactive** nature, this command would find little use in a script.

190. cmp

The **cmp** command is a simpler version of diff, above. Whereas diff reports the differences between two files, cmp merely shows at what point they differ.Use **zcmp** on gzipped files.

191. comm

Versatile file comparison utility. The files must be **sorted** for this to be useful.

192. basename

193. dirname

Strips the basename from a filename, printing only the path information.

194. split, csplit

These are utilities for **splitting a file into smaller chunks**. Their usual use is for splitting up large files in order to back them up on floppies or preparatory to e-mailing or uploading them.

The **csplit** command splits a file according to context, the split occuring where patterns are matched.

Encoding and Encryption1

195. sum, cksum, md5sum, sha1sum

These are utilities for generating **checksums**. A checksum is a number mathematically calculated from the contents of a file, for the purpose of checking its integrity. A script might refer to a list of checksums for security purposes, such as ensuring that the contents of key system files have not been altered or corrupted. For security applications, use the **md5sum**(message digest 5 checksum) command, or better yet, the newer **sha1sum** (Secure Hash Algorithm).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
8babc97a6f62a4649716f4df8d61728f  -
```

196. uuencode

This utility encodes **binary files** (images, sound files, compressed files, etc.) into **ASCII characters**, making them suitable for transmission in the body of an e-mail message or in a newsgroup posting. This is especially useful where MIME (multimedia) encoding is not available.

197. uudecode

This reverses the encoding, decoding uuencoded files back into the original binaries.

198. mimencode, mmencode

The **mimencode** and **mmencode** commands process multimedia-encoded e-mail attachments. Although mail user agents(such as pine or kmail) normally handle this automatically, these particular utilities permit manipulating such attachments manually from the command-line or in batch processing mode by means of a shell script.

199. crypt

At one time, this was the standard UNIX file encryption utility. Politically-motivated government regulations prohibiting the export of encryption software resulted in the disappearance of crypt from much of the UNIX world, and it is still missing from most Linux distributions. Fortunately, programmers have come up with a number of decent alternatives to it, among them the author's very own `cruft` (see Example A-4).

200. openssl

This is an Open Source implementation of Secure Sockets Layer encryption.

```
 #To encrypt a file:
openssl aes-128-ecb -salt -in file.txt -out file.encrypted \
-pass pass:my_password
#                              User-selected password.
#       aes-128-ecb      is the encryption method chosen.

# To decrypt an openssl-encrypted file:
openssl aes-128-ecb -d -salt -in file.encrypted -out file.txt \
-pass pass:my_password
#             User-selected password.
```

Piping openssl to/from tar makes it possible to encrypt an entire directory tree.

```
# To encrypt a directory:

sourcedir="/home/bozo/testfiles"
encrfile="encr-dir.tar.gz"
password=my_secret_password

tar czvf - "$sourcedir" |
openssl des3 -salt -out "$encrfile" -pass pass:"$password"
#             Uses des3 encryption.
# Writes encrypted file "encr-dir.tar.gz" in current working directory.

# To decrypt the resulting tarball:
openssl des3 -d -salt -in "$encrfile" -pass pass:"$password" |
```

```
tar -xzv
# Decrypts and unpacks into current working directory.
```

201. shred

Securely **erase** a file by overwriting it multiple times with random bit patterns before deleting it. This command has the same effect as Example 16-60, but does it in a more thorough and elegant manner.

Advanced forensic technology may still be able to recover the contents of a file, even after application of **shred**.

Miscellaneous

202. mktemp

Create a temporary file with a "unique" filename. When invoked from the command-line without additional arguments, it creates a zero-length file in the /tmp directory.

203. make

- Utility for **building and compiling binary packages.** This can also be used for any set of operations triggered by incremental changes in source files.
- The make command checks a **Makefile**, a list of file dependencies and operations to be carried out.
- The make utility is, in effect, a powerful scripting language similar in many ways to Bash, but with the capability of recognizing dependencies. For in-depth coverage of this useful tool set, see the GNU software documentation site.

204. install

Special purpose file copying command, similar to cp, but capable of setting permissions and attributes of the copied files. This command seems tailormade for installing software packages, and as such it shows up frequently in Makefiles (in the make install : section). It could likewise prove useful in installation scripts.

205. dos2unix

This utility, written by Benjamin Lin and collaborators, converts DOS-formatted text files (lines terated by CR-LF) to UNIX format (lines terminated by LF only), and vice-versa.

206. ptx

The **ptx [targetfile]** command outputs a permuted index (cross-reference list) of the targetfile. This may be further filtered and formatted in a pipe, if necessary.

207. more, less

Pagers that display a text file or stream to stdout, one screenful at a time. These may be used to filter the output of stdout . . . or of a script.

Communications Commands

Certain of the following commands find use in **network** data transfer and analysis, as well as in **chasing spammers**.

208. host

Searches for information about an Internet host by name or IP address, using DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

209. ipcalc

Displays IP information for a host. With the -h option, ipcalc does a reverse DNS lookup, finding the name of the host (server) from the IP address.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

210. nslookup

Do an Internet "**name server lookup**" on a host by IP address. This is essentially equivalent to ipcalc -h or dig -x . The command may be run either interactively or noninteractively, i.e., from within a script.

The **nslookup** command has allegedly been "deprecated," but it is still useful.

211. dig

Domain Information Groper. Similar to nslookup, dig does an Internet name server lookup on a host. May be run from the command-line or from within a script.

Some interesting options to **dig** are **+time=N** for setting a query timeout

to N seconds, +nofail for continuing to query servers until a reply is received, and -x for doing a reverse address lookup.

212. traceroute

Trace the route taken by packets sent to a remote host. This command works within a LAN, WAN, or over the Internet. The remote host may be specified by an IP address. The output of this command may be filtered by grep or sed in a pipe.

213. ping

Broadcast an ICMP ECHO_REQUEST packet to another machine, either on a local or remote network. This is a diagnostic tool for testing network connections, and it should be used with caution.

A successful ping returns an exit status of 0. This can be tested for in a script.

214. whois

Perform a DNS (Domain Name System) lookup. The -h option permits specifying which particular whois server to query. SeeExample 4-6 and Example 16-40.

215. finger

Retrieve information about **users** on a network. Optionally, this command can display a user's **~/.plan**, **~/.project**, and **~/.forwardfiles**, if present.

216. chfn

Change information disclosed by the finger command.

217. vrfy

Verify an Internet e-mail address.

This command seems to be missing from newer Linux distros.

Remote Host Access

218. sx、rx

The **sx** and **rx** command set serves to transfer files to and from a remote host using the **xmodem** protocol. These are generally part of a communications package, such as **minicom**.

219. sz, rz

The sz and rz command set serves to transfer files to and from a remote host using the **zmodem** protocol. Zmodem has certain advantages over xmodem, such as faster transmission rate and resumption of interrupted file transfers. Like sx and rx, these are generally part of a communications package.

220. ftp

Utility and protocol for **uploading / downloading** files to or from a remote host. An ftp session can be automated in a script (see Example 19-6 and Example A-4).

221. uucp, uux, cu

- **uucp:** UNIX to UNIX copy. This is a communications package for transferring files between UNIX servers. A shell script is an effective way to handle a uucp command sequence.

  Since the advent of the Internet and e-mail, uucp seems to have faded into obscurity, but it still exists and remains perfectly workable in situations where an Internet connection is not available or appropriate. The advantage of uucp is that it is fault-tolerant, so even if there is a service interruption the copy operation will resume where it left off when the connection is restored.

- **uux:** UNIX to UNIX execute. Execute a command on a remote system. This command is part of the uucp package.

- **cu:** Call Up a remote system and connect as a simple terminal. It is a sort of dumbed-down version of telnet. This command is part of the uucp package.

222. telnet

Utility and protocol for connecting to a remote host.

Attention: The telnet protocol contains security holes and should therefore probably be avoided. Its use within a shell script is not recommended.

223. wget

The **wget** utility noninteractively retrieves or downloads files from a Web or ftp site. It works well in a script.

```
wget -p http://www.xyz23.com/file01.html
#  The -p or --page-requisite option causes wget to fetch all files
```

```
#+ required to display the specified page.

wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
#  The -r option recursively follows and retrieves all links
#+ on the specified site.

wget -c ftp://ftp.xyz25.net/bozofiles/filename.tar.bz2
#  The -c option lets wget resume an interrupted download.
#  This works with ftp servers and many HTTP sites.
```

224. lynx

The **lynx** Web and file browser can be used inside a script (with the **-dump** option) to retrieve a file from a Web or ftp site noninteractively.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

With the **-traversal** option, lynx starts at the HTTP URL specified as an argument, then "crawls" through all links located on that particular server. Used together with the -crawl option, outputs page text to a log file.

225. rlogin

Remote login, initates a session on a remote host. This command has security issues, so use **ssh** instead.

226. rsh

Remote shell, executes command(s) on a remote host. This has security issues, so use ssh instead.

227. rcp

Remote copy, copies files between two different networked machines.

228. rsync

Remote synchronize, updates (synchronizes) files between two different networked machines.

229. ssh

Secure shell, logs onto a remote host and executes commands there. This secure replacement for telnet, rlogin, rcp, and rshuses identity authentication and encryption. See its manpage for details.

Within a loop, ssh may cause unexpected behavior. According to a Usenet post in the comp.unix shell archives, ssh inherits the loop's stdin. To remedy this, pass ssh either the **-n or -f** option.

230. scp

Secure copy, similar in function to rcp, copies files between two different networked machines, but does so using authentication, and with a security level similar to ssh.

Local Network

231. write

This is a utility for terminal-to-terminal communication. It allows sending lines from your terminal (console or xterm) to that of another user. The mesg command may, of course, be used to disable write access to a terminal

Since write is interactive, it would not normally find use in a script.

232. netconfig

A command-line utility for configuring a network adapter (using DHCP). This command is native to Red Hat centric Linux distros.

Mail

233. mail

Send or read e-mail messages.

This stripped-down command-line mail client works fine as a command embedded in a script.

234. mailto

Similar to the mail command, **mailto** sends e-mail messages from the command-line or in a script. However, mailto also permits sending **MIME** (multimedia) messages.

235. mailstats

Show mail statistics. This command may be invoked only by root.

236. vacation

This utility automatically replies to e-mails that the intended recipient is on vacation and temporarily unavailable. It runs on a network, in conjunction with sendmail, and is

not applicable to a dial-up POPmail account.

Terminal Control Commands

237. tput

Initialize terminal and/or fetch information about it from terminfo data. Various options permit certain terminal operations: **tput clear** is the equivalent of clear; **tput reset** is the equivalent of reset.

- Issuing a tput cup X Y moves the cursor to the (X,Y) coordinates in the current terminal. A clear to erase the terminal screen would normally precede this.
- Some interesting options to tput are:
    - **bold,** for high-intensity text
    - **smul,** to underline text in the terminal
    - **smso**, to render text in reverse
    - **sgr0,** to reset the terminal parameters (to normal), without clearing the screen

238. infocmp

This command prints out extensive information about the current terminal. It references the terminfo database.

239. reset

Reset terminal parameters and clear text screen. As with clear, the cursor and prompt reappear in the upper lefthand corner of the terminal.

240. clear

The clear command simply clears the text screen at the console or in an xterm. The prompt and cursor reappear at the upper lefthand corner of the screen or xterm window. This command may be used either at the command line or in a script. SeeExample 11-25.

241. resize

Echoes commands necessary to set $TERM and $TERMCAP to duplicate the size (dimensions) of the current terminal.

242. script

This utility records (saves to a file) all the user keystrokes at the command-line in a console or an xterm window. This, in effect, creates a record of a session.

Math Commands

243. factor (算因子)

Decompose an integer into prime factors.

244. bc

Bash can't handle floating point calculations, and it lacks operators for certain important mathematical functions. Fortunately, bc gallops to the rescue.

- Here is a simple template for using bc to calculate a **script** variable. This uses command substitution.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
echo "ibase=8; 10"|bc  #进制转换，转成十进制
bc <<!
        obase=16; "hex="; $dec
        obase=10; "dec="; $dec
        obase=8;  "oct="; $dec
        obase=2;  "bin="; $dec
!
```

- An alternate method of invoking bc involves using a here document embedded within a command substitution block. This is especially appropriate when a script needs to pass a list of options and commands to bc.

```
variable=`bc << LIMIT_STRING  #测试发现！，s之类都 可以
options
statements
operations
LIMIT_STRING
`
...or...
variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

245. dc

The **dc** (**desk calculator**) utility is **stack-oriented** and uses **RPN** (Reverse Polish Notation). Like **bc**, it has much of the power of a programming language.

Similar to the procedure with bc, echo a command-string to dc.

```
echo "[Printing a string ... ]P" | dc
# The P command prints the string between the preceding brackets.

# And now for some simple arithmetic.
echo "7 8 * p" | dc     # 56
#  Pushes 7, then 8 onto the stack,
#+ multiplies ("*" operator), then prints the result ("p" operator).
```

Most persons avoid dc, because of its non-intuitive input and rather cryptic operators. Yet, it has its uses.

246. awk

```
AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#             command(s) / parameters passed to awk
  echo $1 $2 | awk "$AWKSCRIPT"
```

Miscellaneous Commands

247. jot、seq

These utilities emit a **sequence of integers,** with a user-selectable increment.

The default **separator** character between each integer is a **newline**, but this can be changed with the -s option.

- Somewhat more capable than seq, jot is a classic UNIX utility that is not normally included in a standard Linux distro. However, the source rpm is available for download from the MIT repository.

Unlike seq, jot can generate a sequence of **random** numbers, using the **-r** option.

```
bash$ jot -r 3 999
1069
 1272
 1428
```

248. getopt

```
set -- `getopt "abcd:" "$@"`
# Sets positional parameters to command-line arguments.
# What happens if you use "$*" instead of "$@"?

while [ ! -z "$1" ]
do
 case "$1" in
   -a) echo "Option \"a\"";;
   -b) echo "Option \"b\"";;
   -c) echo "Option \"c\"";;
   -d) echo "Option \"d\" $2";;
    *) break;;
 esac

 shift
done

#  It is usually better to use the 'getopts' builtin in a script.
```

- It is often necessary to include an eval to correctly process whitespace and quotes.

```
args=$(getopt -o a:bc:d -- "$@")
eval set -- "$args"
```

249. run-parts

The run-parts command executes all the scripts in a target directory, sequentially in ASCII-sorted filename order. Of course, the scripts need to have execute permission.

250. yes

In its default behavior the yes command feeds a continuous string of the character **y** followed by a line feed to **stdout**. A **control-C** terminates the run. A different output string may be specified, as in **yes different string**, which would continually output different string to stdout.

- One might well ask the **purpose** of this. From the command-line or in a script, the output of yes can be redirected or piped into a program expecting user input. In effect, this becomes a sort of poor man's version of expect.
- **yes | fsck /dev/hda1** runs fsck non-interactively (careful!).
  yes | rm -r dirname has same effect as rm -rf dirname (careful!).
  The yes command **parses variables**, or more accurately, it echoes parsed variables. For example:

```
bash$ yes $BASH_VERSION
3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
. . .
```

This particular "feature" may be used to create a very large ASCII file on the fly:

251. banner

Prints arguments as a large vertical banner to stdout, using an ASCII character (default '#'). This may be redirected to a printer for hardcopy.

Note that banner has been dropped from many Linux distros.

252. printenv

Show all the environmental variables set for a particular user.

253. lp

The **lp** and **lpr** commands send file(s) to the print queue, to be printed as hard copy. These commands trace the origin of their names to the line printers of another era.

- bash$ **lp file1.txt** or bash **lp <file1.txt**
  It is often useful to pipe the formatted output from **pr** to **lp**.
  bash$ **pr -options file1.txt | lp**
- Formatting packages, such as **groff** and **Ghostscript** may send their output directly to **lp**.
  bash$ **groff -Tascii file.tr | lp**
  bash$ **gs -options | lp file.ps**

Related commands are **lpq,** for viewing the print queue, and **lprm**, for removing jobs from the print queue.

254. tee

[UNIX borrows an idea from the plumbing trade.]

This is a **redirection** operator, but with a difference. Like the plumber's tee, it permits "siphoning off" to a file the output of a command or commands within a pipe, but without affecting the result. This is useful for printing an ongoing process to a file or paper, perhaps to keep track of it for debugging purposes.

255. mkfifo

This obscure command creates a **named pipe**, a temporary **first-in-first-out** buffer for transferring data between processes. Typically, one process writes to the FIFO, and the other reads from it. See Example A-14.

```
mkfifo pipe1   # Yes, pipes can be given names.
mkfifo pipe2   # Hence the designation "named pipe."

(cut -d' ' -f1 | tr "a-z" "A-Z") >pipe2 <pipe1 &
ls -l | tr -s ' ' | cut -d' ' -f3,9- | tee pipe1 |
cut -d' ' -f2 | paste - pipe2

rm -f pipe1
rm -f pipe2
```

256. pathchk

This command checks the validity of a filename. If the filename exceeds the maximum allowable length (255 characters) or one or more of the directories in its path is not searchable, then an error message results.

Unfortunately, pathchk does not return a recognizable error code, and it is therefore pretty much useless in a script. Consider instead the file test operators.

257. dd

Though this somewhat obscure and much feared **data duplicator** command originated as a utility for exchanging data on magnetic tapes between UNIX minicomputers and IBM mainframes, it still has its uses. The **dd** command simply copies a file (or stdin/stdout), but with conversions. Possible conversions include **ASCII/EBCDIC**, **upper/lower case**, **swapping of byte pairs** between input and output, and **skipping** and/or **truncating** the head or tail of the input file.

```
# Converting a file to all uppercase:

dd if=$filename conv=ucase > $filename.uppercase
#                  lcase   # For lower case conversion
```

- Some basic options to dd are:

  if=INFILE

  INFILE is the source file.

  of=OUTFILE

OUTFILE is the target file, the file that will have the data written to it.

bs=BLOCKSIZE

This is the size of each block of data being read and written, usually a power of 2.

skip=BLOCKS

How many blocks of data to skip in INFILE before starting to copy. This is useful when the INFILE has "garbage" or garbled data in its header or when it is desirable to copy only a portion of the INFILE.

seek=BLOCKS

How many blocks of data to skip in OUTFILE before starting to copy, leaving blank data at beginning of OUTFILE.

count=BLOCKS

Copy only this many blocks of data, rather than the entire INFILE.

conv=CONVERSION

Type of conversion to be applied to INFILE data before copying operation.

- To demonstrate just how versatile dd is, let's use it to capture keystrokes.

```
keys=$(dd bs=1 count=1 2> /dev/null)
# 'dd' uses stdin, if "if" (input file) not specified.
```

- The dd command can do random access on a data stream.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
#  The "conv=notrunc" option means that the output file
#+ will not be truncated.
```

- The dd command can copy raw data and disk images to and from devices, such as floppies and tape drives (Example A-5). A common use is creating boot floppies.

  dd if=kernel-image of=/dev/fd0H1440

- Similarly, dd can copy the entire contents of a floppy, even one formatted with a "foreign" OS, to the hard drive as an image file.

  dd if=/dev/fd0 of=/home/bozo/projects/floppy.img

- Likewise, dd can create bootable flash drives. **dd if=image.iso of=/dev/sdb** See Marlow's Bootable USB Keys site.

- Other applications of dd include initializing temporary swap files (Example 31-2) and ramdisks (Example 31-3). It can even do a low-level copy of an entire hard drive partition, although this is not necessarily recommended.

258. od

The **od**, or **octal dump** filter converts input (or files) to octal (base-8) or other bases. This is useful for viewing or processing binary data files or otherwise unreadable system device files, such as **/dev/urandom**, and as a filter for binary data.

```
head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
```

```
# Sample output: 1324725719, 3918166450, 2989231420, etc.

SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
```

259. hexdump

Performs a hexadecimal, octal, decimal, or ASCII dump of a binary file. This command is the rough equivalent of od, above, but not nearly as useful. May be used to view the contents of a binary file, in combination with **dd** and less.

```
dd if=/bin/ls | hexdump -C | less
# The -C option nicely formats the output in tabular form.
```

260. objdump

Displays information about an object file or binary executable in either hexadecimal form or as a disassembled listing (with the **-d** option).

261. mcookie

This command generates a "**magic cookie**," a 128-bit (32-character) pseudorandom hexadecimal number, normally used as an authorization "signature" by the X server. This also available for use in a script as a "quick 'n dirty" random number.

262. units

This utility converts between different units of measure. While normally invoked in interactive mode, **units** may find use in a script.

263. m4

A hidden treasure, m4 is a powerful macro processing filter, virtually a complete language. Although originally written as a pre-processor for RatFor, m4 turned out to be useful as a stand-alone utility. In fact, m4 combines some of the functionality of eval, tr, and awk, in addition to its extensive macro expansion facilities.

```
#!/bin/bash
# m4.sh: Using the m4 macro processor

# Strings
string=abcdA01
echo "len($string)" | m4                    #  7
echo "substr($string,4)" | m4                 # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4        # 01Z
```

```
# Arithmetic
var=99
echo "incr($var)" | m4                    #  100
echo "eval($var / 3)" | m4                 #  33

exit
```

264. xmessage

This X-based variant of echo pops up **a message/query window** on the desktop.

```
xmessage Left click to continue -button okay
```

265. zenity

The **zenity** utility is adept at displaying GTK+ dialog widgets and very suitable for scripting purposes.

266. doexec

The **doexec** command enables passing an arbitrary list of arguments to a binary executable. In particular, passing **argv[0]**(which corresponds to **$0** in a script) lets the executable be invoked by various names, and it can then carry out different sets of actions, according to the name by which it was called. What this amounts to is roundabout way of passing options to an executable.

For example, the /usr/local/bin directory might contain a binary called "aaa". Invoking **doexec /usr/local/bin/aaa list** would listall those files in the current working directory beginning with an "a", while invoking (the same executable with) **doexec /usr/local/bin/aaa delete** would delete those files.

- The various behaviors of the executable must be defined within the code of the executable itself, analogous to something like the following in a shell script:

```
case `basename $0` in
"name1" ) do_something;;
"name2" ) do_something_else;;
"name3" ) do_yet_another_thing;;
*       ) bail_out;;
esac
```

267. dialog

The **dialog** family of tools provide a method of calling interactive "dialog" boxes from

a script. The more elaborate variations of dialog -- **gdialog**, **Xdialog**, and **kdialog** -- actually invoke X-Windows widgets.

268. sox

The **sox**, or "**sound exchange**" command plays and performs transformations on sound files. In fact, the /usr/bin/playexecutable (now deprecated) is nothing but a shell wrapper for sox.

For example, **sox soundfile.wav soundfile.au** changes a WAV sound file into a (Sun audio format) AU sound file.

Shell scripts are ideally suited for batch-processing sox operations on sound files. For examples, see the Linux Radio Timeshift HOWTO and the MP3do Project.

System and Administrative Commands

The startup and shutdown scripts in **/etc/rc.d** illustrate the uses (and usefulness) of many of these comands. These are usually invoked by root and used for system maintenance or emergency filesystem repairs. Use with caution, as some of these commands may damage your system if misused.

Users and Groups

269. users

Show all logged on users. This is the approximate equivalent of **who -q**.

270. groups

Lists the current user and the groups she belongs to. This corresponds to the **$GROUPS** internal variable, but gives the group names, rather than the numbers.

271. chown、chgrp

The **chown** command changes the ownership of a file or files. This command is a useful method that root can use to shift file ownership from one user to another. An ordinary user may not change the ownership of files, not even her own files.

The **chgrp** command changes the group ownership of a file or files. You must be

owner of the file(s) as well as a member of the destination group (or root) to use this operation.

272. useradd, userdel

The useradd administrative command adds a user account to the system and creates a home directory for that particular user, if so specified. The corresponding userdel command removes a user account from the system [2] and deletes associated files.

273. usermod

Modify a user account. Changes may be made to the password, group membership, expiration date, and other attributes of a given user's account. With this command, a user's password may be locked, which has the effect of disabling the account.

274. groupmod

Modify a given group. The group name and/or ID number may be changed using this command.

275. id

The id command lists the **real** and **effective** user IDs and the group IDs of the user associated with the current process. This is the counterpart to the **$UID**, **$EUID**, and **$GROUPS** internal Bash variables.

The id command shows the **effective** IDs only when they differ from the **real** ones.

276. lid

The lid (**list ID**) command shows the group(s) that a given user belongs to, or alternately, the users belonging to a given group. May be invoked only by root.

277. who

Show all users logged on to the system.

The **-m** gives detailed information about only the current user. Passing any two arguments to who is the equivalent of **who -m**, as in **who am i** or **who The Man**.

278. w

Show all logged on users and the processes belonging to them. This is an extended

version of **who**. The output of **w** may be piped to **grep** to find a specific user and/or process.

279. logname

Show **current user's login name** (as found in **/var/run/utmp**). This is a near-equivalent to whoami, above.

While **logname** prints the name of the logged in user, **whoami** gives the name of the user attached to the **current process**. As we have just seen, sometimes these are not the same.

280. su

Runs a program or script as a **substitute use**r. su rjones starts a shell as user rjones. A naked su defaults to root. See Example A-14.

281. sudo

Runs a command as root (or another user). This may be used in a script, thus permitting a regular user to run the script.

The file **/etc/sudoers** holds the names of users permitted to invoke sudo.

282. passwd

Sets, changes, or manages a user's password.

283. ac

Show users' **logged** in time, as read from /var/log/wtmp. This is one of the GNU accounting utilities.

284. last

List **last logged** in users, as read from **/var/log/wtmp**. This command can also show remote logins.

285. newgrp

Change user's group ID without logging out. This permits access to the new group's files. Since users may be members of multiple groups simultaneously, this command finds only limited use.

Terminals

286. tty

Echoes the name (filename) of the **current user's terminal**. Note that each separate xterm window counts as a different terminal.

287. stty

Shows and/or changes terminal settings. This complex command, used in a script, can **control terminal behavior and the way output displays**. See the info page, and **study it carefully.**

```
stty erase '#'                # Set "hashmark" (#) as erase character.
stty -echo    # Turns off screen echo.
stty echo     # Restores screen echo.
```

288. setterm

Set certain terminal attributes. This command writes to its terminal's stdout a string that changes the behavior of that terminal.

```
bash$ setterm -cursor off
setterm -bold on
echo bold hello

setterm -bold off
echo normal hell
```

289. tset

Show or initialize terminal settings. This is a less capable version of **stty**.

290. setserial

Set or display serial port parameters. This command must be run by root and is usually found in a system setup script.

291. getty, agetty

The initialization process for a terminal uses getty or agetty to set it up for login by a user. These commands are not used within user shell scripts. Their scripting counterpart is stty.

292. mesg

Enables or disables write access to the current user's terminal. Disabling access would prevent another user on the network to **write** to the terminal.

293. wall

This is an acronym for "**write all**," i.e., sending a message to all users at every terminal logged into the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem (see Example 19-1).

Information and Statistics

294. uname

Output system specifications (OS, kernel version, etc.) to stdout. Invoked with the -a option, gives verbose system info (seeExample 16-5). The -s option shows only the OS type.

295. arch

Show system architecture. Equivalent to uname -m. See Example 11-26.

296. lastcomm

Gives information about **previous commands**, as stored in the **/var/account/pacct** file. Command name and user name can be specified by options. This is one of the GNU accounting utilities.

297. lastlog

List the last login time of all system users. This references the **/var/log/lastlog** file.

298. lsof

List open files. This command outputs a detailed table of all currently open files and gives information about their owner, size, the processes associated with them, and more. Of course, lsof may be piped to grep and/or awk to parse and analyze its results.

The **lsof** command is a useful, if complex administrative tool. If you are unable to dismount a filesystem and get an error message that it is still in use, then running lsof helps determine which files are still open on that filesystem. The **-i** option

lists open network socket files, and this can help trace intrusion or hack attempts.

```
lsof -an -i tcp
```

299. strace

System trace: diagnostic and debugging tool for tracing system calls and signals. This command and **ltrace**, following, are useful for diagnosing why a given program or package fails to run . . . perhaps due to missing libraries or related causes.

300. ltrace

Library trace: diagnostic and debugging tool that traces library calls invoked by a given command.

301. nc

The **nc** (**netcat**) utility is a complete toolkit for connecting to and listening to TCP and UDP ports. It is useful as a diagnostic and testing tool and as a component in simple script-based HTTP clients and servers.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1;
Thu, 31 Mar 2005 15:41:35 -0700
```

302. free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using grep, awk or Perl. The **procinfo** command shows all the information that **free** does, and much more.

303. procinfo

Extract and list information and statistics from the **/proc** pseudo-filesystem. This gives a very extensive and detailed listing.

304. lsdev

List devices, that is, show installed hardware.

305. du

Show (disk) file usage, recursively. Defaults to current working directory, unless otherwise specified.

306. df

Shows filesystem usage in tabular form.

307. dmesg

Lists all **system bootup messages** to stdout. Handy for debugging and ascertaining which device drivers were installed and which system interrupts in use. The output of dmesg may, of course, be parsed with grep, sed, or awk from within a script.

308. stat

Gives detailed and verbose statistics on a given file (even a directory or device file) or set of files.

309. vmstat

Display virtual memory statistics.

310. uptime

Shows how long the system has been running, along with associated statistics.

311. hostname

Lists the **system's host name.** This command sets the host name in an /etc/rc.d setup script (/etc/rc.d/rc.sysinit or similar). It is equivalent to **uname -**n, and a counterpart to the **$HOSTNAME** internal variable.

Similar to the hostname command are the **domainname, dnsdomainname**, **nisdomainname**, and **ypdomainname** commands. Use these to display or set the system DNS or NIS/YP domain name. Various options to hostname also perform these functions.

312. hostid

Echo a 32-bit hexadecimal numerical identifier for the host machine.

313. sar

Invoking **sar** (**System Activity Reporter**) gives a very detailed rundown on system statistics. The Santa Cruz Operation ("Old"SCO) released sar as Open Source in June, 1999.

314. readelf

Show information and statistics about a designated elf binary. This is part of the binutils package.

```
bash$ readelf -h /bin/bash
```

315. size

The **size** [**/path/to/binary**] command gives the segment sizes of a binary executable or archive file. This is mainly of use to programmers.

```
bash$ size /bin/bash
```

System Logs

316. logger

Appends a user-generated message to the system log (**/var/log/messages** or **/var/log/syslog**). You do not have to be root to invoke logger.

```
logger Experiencing instability in network connection at 23:10, 05/21.
```

By embedding a logger command in a script, it is possible to write debugging information to /var/log/messages.

317. logrotate

This utility manages the system log files, rotating, compressing, deleting, and/or e-mailing them, as appropriate. This keeps the **/var/log** from getting cluttered with old log files. Usually **cron** runs logrotate on a daily basis.

Adding an appropriate entry to **/etc/logrotate.conf** makes it possible to manage personal log files, as well as system-wide ones.

Stefano Falsetto has created `rottlog`, which he considers to be an improved version of logrotate.

Job Control

318. ps

Process Statistics: lists currently executing processes by owner and PID (process ID). This is usually invoked with **ax** or **aux** options, and may be piped to grep or sed to

search for a specific process (see Example 15-14 and Example 29-3).

To display system processes in graphical "tree" format: ps afjx or ps ax --forest.

319. pgrep、pkill

Combining the ps command with grep or kill.

320. pstree

Lists currently executing processes in "**tree**" format. The -p option shows the PIDs, as well as the process names.

321. top

Continuously updated display of most cpu-intensive processes. The **-b** option displays in text mode, so that the output may be parsed or accessed from a script.

322. nice

Run a background job with an altered priority. Priorities run from 19 (lowest) to -20 (highest). Only root may set the negative (higher) priorities. Related commands are **renice** and **snice**, which change the priority of a running process or processes, and **skill**, which sends a **kill signal** to a process or processes.

323. nohup

Keeps a command running even after user **logs off.** The command will run as a **foreground** process unless followed by &. If you use nohup within a script, consider coupling it with a **wait** to avoid creating an **orphan** or **zombie** process.

324. pidof

Identifies process ID (**PID**) of a running job. Since job control commands, such as **kill** and **renice** act on the **PID** of a process (not its name), it is sometimes necessary to identify that PID. The pidof command is the approximate counterpart to the**$PPID** internal variable.

325. fuser

Identifies the processes (by PID) that are **accessing** a given file, set of files, or directory. May also be invoked with the **-k** option, which kills those processes. This has interesting implications for system security, especially in scripts preventing unauthorized

users from accessing system services.

- **One important application** for fuser is when physically inserting or removing storage media, such as CD ROM disks or USB flash drives. Sometimes trying a umount fails with a device is busy error message. This means that some user(s) and/or process(es) are accessing the device. An **fuser -um /dev/device_name** will clear up the mystery, so you can kill any relevant processes.
- The **fuser** command, invoked with the **-n** option identifies the processes accessing a port. This is especially useful in combination with **nmap.**

326. cron

Administrative program **scheduler**, performing such duties as cleaning up and deleting system log files and updating theslocate database. This is the superuser version of **at** (although each user may have their own crontab file which can be changed with the crontab command). It runs as a daemon and executes scheduled entries from **/etc/crontab**.

Process Control and Booting

327. init

The **init** command is **the parent of all processes.** Called in the final step of a **bootup**, init determines the runlevel of the system from **/etc/inittab**. Invoked by its alias **telinit**, and by root only.

328. telinit

Symlinked to init, this is a means of **changing the system runlevel**, usually done for system maintenance or emergency filesystem repairs. Invoked only by root. This command can be dangerous -- be certain you understand it well before using!

329. runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel 0), in single-user mode (1), in multi-user mode (2 or 3), in X Windows (5), or rebooting (6). This command accesses the /var/run/utmp file.

330. halt, shutdown, reboot

Command set to shut the system down, usually just prior to a power down.

Attention: On some Linux distros, the halt command has 755 permissions, so it can be invoked by a non-root user. A careless halt in a terminal or a script may shut down the system!

331. service

Starts or stops a system service. The startup scripts in **/etc/init.d** and **/etc/rc.d** use this command to start services at **bootup.**

Network

332. nmap

Network mapper and **port scanner**. This command scans a server to locate open ports and the services associated with those ports. It can also report information about packet filters and firewalls. This is an important security tool for locking down a network against hacking attempts.

333. ifconfig

Network interface configuration and tuning utility.

334. netstat

Show current **network statistics** and information, such as routing tables and active connections. This utility accesses information in /proc/net (Chapter 29). See Example 29-4.

- **netstat -r** is equivalent to **route**.
- A **netstat -lptu** shows sockets that are listening to ports, and the associated processes. This can be useful for determining whether a computer has been hacked or compromised.

335. iwconfig

This is the command set for configuring a **wireless network**. It is the wireless equivalent of ifconfig, above.

336. ip

General purpose utility for setting up, changing, and analyzing IP (Internet Protocol) networks and attached devices. This command is part of the **iproute2** package.

337. route

Show info about or make changes to the kernel routing table.

338. iptables

The **iptables** command set is a packet filtering tool used mainly for such security purposes as setting up network firewalls. This is a complex tool, and a detailed explanation of its use is beyond the scope of this document. Oskar Andreasson's tutorial is a reasonable starting point.

See also shutting down iptables and Example 30-2.

339. chkconfig

Check network and system configuration. This command lists and manages the network and system services started at bootup in the /etc/rc?.d directory.

Originally a port from IRIX to Red Hat Linux, chkconfig may not be part of the core installation of some Linux flavors.

340. tcpdump

Network packet "**sniffer.**" This is a tool for analyzing and troubleshooting traffic on a network by dumping packet headers that match specified criteria.

Dump ip packet traffic between hosts bozoville and caduceus:

```
bash$ tcpdump ip host bozoville and caduceus
```

Filesystem

341. mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file **/etc/fstab** provides a handy listing of available filesystems, partitions, and devices, including options, that may be automatically or manually mounted. The file**/etc/mtab** shows the currently mounted filesystems and partitions (including the virtual ones, such as **/proc**).

mount -a mounts all filesystems and partitions listed in **/etc/fstab**, except those with a noauto option. At bootup, a startup script in **/etc/rc.d** (rc.sysinit or something similar) invokes this to get everything mounted.

The versatile **mount** command can even mount **an ordinary file** on a block device, and the file will act as if it were a filesystem. **Mount** accomplishes that by associating the file with a **loopback** device. One application of this is to mount and examine an ISO9660 filesystem image before burning it onto a **CDR**.

```
mkdir /mnt/cdtest  # Prepare a mount point, if not already there.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Mount the image.
#                  "-o loop" option equivalent to "losetup /dev/loop0"
cd /mnt/cdtest     # Now, check the image.
ls -alR            # List the files in the directory tree there.
                   # And so forth.
```

342. umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be umounted, else filesystem corruption may result.

343. gnome-mount

The newer Linux distros have deprecated **mount** and **umount.** The successor, for command-line mounting of removable storage devices, is **gnome-mount**. It can take the **-d** option to mount a device file by its listing in **/dev**.

344. sync

Forces an **immediate write** of all updated data from buffers to hard drive (synchronize drive with buffers). While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure. In the olden days, a sync; sync (twice, just to make absolutely sure) was **a useful precautionary measure before a system reboot.**

At times, you may wish to force **an immediate buffer flush**, as when securely deleting a file (see Example 16-60) or when the lights begin to flicker.

345. losetup

Sets up and configures **loopback devices**.

- **Creating a filesystem in a file**

```
SIZE=1000000  # 1 meg

head -c $SIZE < /dev/zero > file  # Set up file of designated size.
losetup /dev/loop0 file           # Set it up as loopback device.
mke2fs /dev/loop0                 # Create filesystem.
mount -o loop /dev/loop0 /mnt     # Mount it.
```

346. mkswap

Creates a swap partition or file. The swap area must subsequently be enabled with **swapon**.

347. swapon, swapoff

Enable / disable swap partitition or file. These commands usually take effect at bootup and shutdown.

348. mke2fs

Create a Linux ext2 filesystem. This command must be invoked as root.

349. mkdosfs

Create a DOS FAT filesystem.

350. tune2fs

Tune ext2 filesystem. May be used to change filesystem parameters, such as maximum mount count. This must be invoked asroot.

Attention: This is an extremely dangerous command. Use it at your own risk, as you may inadvertently destroy your filesystem.

351. dumpe2fs

Dump (list to stdout) very verbose filesystem info. This must be invoked as root.

352. hdparm

List or change hard disk parameters. This command must be invoked as root, and it may be dangerous if misused.

353. fdisk

Create or change a partition table on a storage device, usually a hard drive. This command must be invoked as root.

354. fsck, e2fsck, debugfs

Filesystem **check**, **repair**, and **debug** command set.

- **fsck:** a front end for checking a UNIX filesystem (may invoke other utilities). The actual filesystem type generally defaults toext2.
- **e2fsck:** ext2 filesystem checker.
- **debugfs:** ext2 filesystem debugger. One of the uses of this versatile, but dangerous command is to (attempt to) recover deleted files. For advanced users only!

355. badblocks

Checks for bad blocks (physical media flaws) on a storage device. This command finds use when formatting a newly installed hard drive or testing the integrity of backup media. As an example, **badblocks /dev/fd0** tests a floppy disk.

The **badblocks** command may be invoked **destructively** (overwrite all data) or in **non-destructive** read-only mode. If root user owns the device to be tested, as is generally the case, then root must invoke this command.

356. lsusb, usbmodules

- The **lsusb** command lists all USB (Universal Serial Bus) buses and the devices hooked up to them.
- The **usbmodules** command outputs information about the driver modules for connected USB devices.

357. lspci

Lists pci busses present.

358. mkbootdisk

Creates a boot floppy which can be used to bring up the system if, for example, the MBR (master boot record) becomes corrupted. Of special interest is the **--iso** option, which uses mkisofs to create a bootable ISO9660 filesystem image suitable for burning a bootable CDR.

The **mkbootdisk** command is actually a Bash script, written by Erik Troan, in the **/sbin** directory.

359. mkisofs

Creates an **ISO9660** filesystem suitable for a CDR image.

360. chroot

CHange ROOT directory. Normally commands are fetched from **$PATH**, relative to /,

the default root directory. This changes the root directory to a different one (and also changes the working directory to there). This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those telnetting in, to a secured portion of the filesystem (this is sometimes referred to as confining a guest user to a "chroot jail"). Note that after a chroot, the execution path for system binaries is no longer valid.

A **chroot /opt** would cause references to /usr/bin to be translated to /opt/usr/bin. Likewise, chroot /aaa/bbb /bin/ls would redirect future instances of ls to /aaa/bbb as the base directory, rather than / as is normally the case. An alias XX 'chroot /aaa/bbb ls'in a user's `~/.bashrc` effectively restricts which portion of the filesystem she may run command "XX" on.

The chroot command is also handy when running from an emergency boot floppy (chroot to /dev/fd0), or as an option to lilowhen recovering from a system crash. Other uses include installation from a different filesystem (an rpm option) or running a readonly filesystem from a CD ROM. Invoke only as root, and use with care.

361. lockfile

This utility is part of the procmail package (`www.procmail.org`). It creates a lock file, a **semaphore** that controls access to a file, device, or resource.

(Definition: A semaphore is a flag or signal. (The usage originated in railroading, where a colored flag, lantern, or striped movable arm semaphore indicated whether a particular track was in use and therefore unavailable for another train.) A UNIX process can check the appropriate semaphore to determine whether a particular resource is available/accessible. **)**

- The lock file serves **as a flag** that this particular file, device, or resource is in use by a process (and is therefore "**busy**"). The presence of a lock file permits only restricted access (or no access) to other processes.

```
lockfile /home/bozo/lockfiles/$0.lock
# Creates a write-protected lockfile prefixed with the name of the script.

lockfile /home/bozo/lockfiles/${0##*/}.lock
# A safer version of the above, as pointed out by E. Choroba.
```

- Lock files are used in such applications as protecting system mail folders from **simultaneously being changed by multiple users**, indicating that a modem port is being accessed, and showing that an instance of Firefox is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts to create a lock file that already exists, the script will likely hang.

- Normally, applications create and check for lock files in the **/var/lock** directory. A script can test for the presence of a lock file by something like the following.

```
appname=xyzip
# Application "xyzip" created lock file "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock" ]
then   #+ Prevent other programs & scripts
       #  from accessing files/resources used by xyzip.
```

362. flock

Much less useful than the **lockfile** command is **flock**. It sets an "**advisory**" lock on a file and then **executes a command while the lock is on**. This is to prevent any other process from setting a lock on that file until completion of the specified command.

```
flock $0 cat $0 > lockfile__$0
#  Set a lock on the script the above line appears in,
#+ while listing the script to stdout.
```

Unlike lockfile, flock does not automatically create a lock file.

363. mknod

Creates block or character device files (may be necessary when installing new hardware on the system). The MAKEDEV utility has virtually all of the functionality of mknod, and is easier to use.

364. MAKEDEV

Utility for **creating device files**. It must be run as root, and in the /dev directory. It is a sort of advanced version of **mknod**.

365. tmpwatch

Automatically deletes files which have not been accessed within a specified period of time. Usually invoked by **cron** to remove **stale log files**.

Backup

366. dump, restore

The **dump** command is an elaborate filesystem backup utility, generally used on larger installations and networks. It reads raw disk partitions and writes a backup file in a

binary format. Files to be backed up may be saved to a variety of storage media, including disks and tape drives. The **restore** command restores backups made with dump.

367. fdformat

Perform a low-level format on a floppy disk (/dev/fd0*).

System Resources

368. ulimit

Sets an upper limit on use of system resources. Usually invoked with the -f option, which sets a limit on file size (**ulimit -f 1000** limits files to 1 meg maximum). The **-t** option limits the coredump size (**ulimit -c 0** eliminates coredumps). Normally, the value of ulimit would be set in **/etc/profile** and/or **~/.bash_profile** (see Appendix G).

- Judicious use of **ulimit** can protect a system against the **dreaded fork bomb.**

```
#!/bin/bash
# This script is for illustrative purposes only.
# Run it at your own peril -- it WILL freeze your system.

while true  #  Endless loop.
do
 $0 &     #  This script invokes itself . . .
         #+ forks an infinite number of times . . .
         #+ until the system freezes up because all resources exhausted.
done       #  This is the notorious "sorcerer's appentice" scenario.

exit 0     #  Will not exit here, because this script will never terminate.
```

A **ulimit -Hu XX** (where XX is the user process limit) in **/etc/profile** would abort this script when it exceeded the preset limit.

369. quota

Display user or group disk quotas.

370. setquota

Set user or group disk quotas from the command-line.

371. umask

User file creation permissions mask. Limit the default file attributes for a particular user. All files created by that user take on the attributes specified by **umask.** The (octal) value passed to umask defines the file permissions disabled. For example,umask 022 ensures that new files will have at most 755 permissions (777 NAND 022). Of course, the user may later change the attributes of particular files with chmod. The usual practice is to set the value of umask in **/etc/profile** and/or ~/.bash_profile(see Appendix G).

372. rdev

Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but rdev remains useful for setting up a ram disk. This is a dangerous command, if misused.

Modules

373. lsmod

List installed kernel modules.

Doing a **cat /proc/modules** gives the same information.

374. insmod

Force installation of a kernel module (use modprobe instead, when possible). Must be invoked as root.

375. rmmod

Force unloading of a kernel module. Must be invoked as root.

376. modprobe

Module loader that is normally invoked automatically in a startup script. Must be invoked as root.

377. depmod

Creates module dependency file. Usually invoked from a startup script.

378. modinfo

Output information about a loadable module.

Miscellaneous

379. env

Runs a program or script with certain **environmental variables** set or changed (without changing the overall system environment). The [varname=xxx] permits changing the environmental variable varname for the duration of the script. With no options specified, this command lists all the environmental variable settings.

- The first line of a script (the "sha-bang" line) may use **env** when the path to the shell or interpreter is unknown.

```
#! /usr/bin/env perl

print "This Perl script will run,\n";
print "even when I don't know where to find Perl.\n";

# Good for portable cross-platform scripts,
# where the Perl binaries may not be in the expected place.
```

380. ldd

Show shared lib dependencies for an **executable file.**

381. watch

Run a command repeatedly, at specified time intervals.

The default is two-second intervals, but this may be changed with the **-n** option.

```
watch -n 5 tail /var/log/messages
# Shows tail end of system log, /var/log/messages, every five seconds.
```

- Unfortunately, **piping** the output of **watch** command to **grep** does not work.

382. strip

Remove the debugging symbolic references from an executable binary. This decreases its size, but makes debugging it impossible.

This command often occurs in a **Makefile**, but rarely in a shell script.

383. nm

List symbols in an unstripped compiled binary.

384. xrandr

Command-line tool for manipulating the root window of the screen.

385. rdist

Remote distribution client: synchronizes, clones, or backs up a file system on a remote server.

386. Exercise

- **Exercise 1.** In /etc/rc.d/init.d, analyze the halt script. It is a bit longer than killall, but similar in concept. Make a copy of this script somewhere in your home directory and experiment with it (do not run it as root). Do a simulated run with the -vn flags (sh -vn scriptname). Add extensive comments. Change the commands to echos.
- **Exercise 2.** Look at some of the more complex scripts in /etc/rc.d/init.d. Try to understand at least portions of them. Follow the above procedure to analyze them. For some additional insight, you might also examine the file sysvinitfiles in /usr/share/doc/initscripts-?.??, which is part of the "initscripts" documentation.

Advanced Topics

Regular Expressions

387. introduction

- *** . ^ $ [ ] **
- **Escaped "angle brackets"** -- **\<...\>** -- mark **word** boundaries.

  The angle brackets must be escaped, since otherwise they have only their literal character meaning.

  "**\<the\>**" matches the word "the," but not the words "them," "there," "other," etc.

388. Extended REs

- **?:** matches zero or one of the previous RE
- **+:** matches one or more of the previous RE

```
# GNU versions of sed and awk can use "+",
# but it needs to be escaped.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# All of above are equivalent.
```

- **Escaped "curly brackets"** -- **\{ \}** -- indicate the number of occurrences of a preceding RE to match.

  It is necessary to escape the curly brackets since they have only their literal character meaning otherwise. This usage is technically not part of the basic RE set.

  "**[0-9]\{5\}**" matches exactly five digits (characters in the range of 0 to 9).

  Curly brackets are not available as an RE in the "classic" (non-POSIX compliant) version of **awk**. However, the GNU extended version of awk, **gawk**, has the **--re-interval** option that permits them (without being escaped).

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

  Perl and some **egrep** versions do not require escaping the curly brackets.

- **Parentheses** -- **( )** -- enclose a group of REs. They are useful with the following "|" operator and in substring extraction using **expr**.

- The -- | -- "**or**" RE operator matches any of a set of alternate characters.

```
egrep 're(a|e)d' misc.txt
```

389. POSIX Character Classes. [:class:]

This is an alternate method of specifying a range of characters to match.

- **[:alnum:]** matches alphabetic or numeric characters. This is equivalent to A-Za-z0-9.
- **[:alpha:]** matches alphabetic characters. This is equivalent to A-Za-z.
- **[:blank:]** matches a space or a tab.
- **[:cntrl:]** matches control characters.
- **[:digit:]** matches (decimal) digits. This is equivalent to 0-9.
- **[:graph:]** (graphic printable characters). Matches characters in the range of **ASCII 33 - 126**. This is the same as [:print:], below, but excluding the space character.
- **[:lower:]** matches lowercase alphabetic characters. This is equivalent to a-z.

- **[:print:]** (printable characters). Matches characters in the range of **ASCII 32 - 126**. This is the same as [:graph:], above, but adding the space character.
- **[:space:]** matches whitespace characters (space and horizontal tab).
- **[:upper:]** matches uppercase alphabetic characters. This is equivalent to A-Z.
- **[:xdigit:]** matches hexadecimal digits. This is equivalent to 0-9A-Fa-f.

Attention: POSIX character classes generally require quoting or double brackets ([[ ]]).

```
bash$ grep [[:digit:]] test.file
```

- These character classes may even be used with `globbing`, to a limited extent.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
```

Globbing

390. Introduction

Bash itself cannot recognize Regular Expressions. Inside scripts, it is commands and utilities -- such as sed and awk -- that interpret RE's.

Bash does carry out filename expansion -- a process known as **globbing** -- but this does not use the standard RE set. Instead, **globbing** recognizes and expands **wild cards**. Globbing interprets the standard wild card characters -- ***** and **?**, character lists in **square brackets**, and certain other special characters (such as **^ for negating the sense of a match**). There are important limitations on wild card characters in globbing, however. Strings containing * will **not match filenames that start with a dot**, Likewise, the ? has a different meaning in globbing than as part of an RE.

Bash performs **filename expansion** on **unquote**d command-line arguments.

```
echo *
a.1 b.1 c.1 t2.sh test1.txt
```

It is possible to modify the way Bash interprets special characters in **globbing**. A **set -f** command disables globbing, and the **nocaseglob** and **nullglob** options to **shopt** change globbing behavior.

Attention: Filename expansion can match dotfiles, but only if the pattern explicitly includes the dot as a literal character.

```
~/[.]bashrc    #  Will not expand to ~/.bashrc
~/?bashrc      #  Neither will this.
               #  Wild cards and metacharacters will NOT
```

```
            #+ expand to a dot in globbing.

~/.[b]ashrc    #  Will expand to ~/.bashrc
~/.ba?hrc      #  Likewise.
~/.bashr*      #  Likewise.

# Setting the "dotglob" option turns this off.
shopt dotglob
```

Here Documents

391. Here documents

A here document is a special-purpose code block. It uses a form of I/O redirection to feed a command list to an interactive program or a command, such as ftp, cat, or the ex text editor.

```
COMMAND <<InputComesFromHERE
...
InputComesFromHERE
```

```
#--------Begin here document-----------#
vi $TARGETFILE <<x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23
#----------End here document-----------#
#  Note that ^[ above is a literal escape
#+ typed by Control-V <Esc>.
#  Bram Moolenaar points out that this may not work with 'vim'
#+ because of possible problems with terminal interaction.
```

```
#!/bin/bash
```

```
#  Replace all instances of "Smith" with "Jones"
#+ in files with a ".txt" filename suffix.

ORIGINAL=Smith
REPLACEMENT=Jones

for word in $(fgrep -l $ORIGINAL *.txt)
do
 # ------------------------------------
 ex $word <<EOF
 :%s/$ORIGINAL/$REPLACEMENT/g
 :wq
EOF
 # :%s is the "ex" substitution command.
 # :wq is write-and-quit.
 # ------------------------------------
done
```

- The **-** option to mark a here document limit string (**<<-LimitString**) suppresses leading tabs (but not spaces) in the output. This may be useful in making a script more readable.

- A here document supports **parameter** and **command substitution**. It is therefore possible to pass different parameters to the body of the here document, changing its output accordingly.

```
cat <<Endofmessage

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.

# This comment shows up in the output (why?).

Endofmessage
```

- **Quoting or escaping** the "limit string" at the **head** of a here document disables parameter substitution within its body. The reason for this is that quoting/escaping the limit string effectively escapes the $, `, and \ special characters, and causes them to be interpreted literally. (Thank you, Allen Halsey, for pointing this out.)

```
cat <<'Endofmessage'

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.
```

```
Endofmessage
```

- It is possible to **set a variable** from the output of a here document. This is actually a devious form of **command substitution**.

```
variable=$(cat <<SETVAR
This variable
runs over multiple lines.
SETVAR)

echo "$variable"
```

- A here document can supply input to a function in the same script.

```
GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # This certainly looks like an interactive function, but...


# Supply input to the above function.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
MD
21226
RECORD001
```

- It is possible to use **:** as a **dummy command** accepting output from a here document. This, in effect, creates an "anonymous" here document.

```
: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # Print error message if one of the variables not set.
TESTVARIABLES
```

- A variation of the above technique permits **"commenting out" blocks of code.**

```
: <<COMMENTBLOCK
```

```
echo "This line will not echo."
This is a comment line missing the "#" prefix.
This is another comment line missing the "#" prefix.

&*@!!++=
The above line will cause no error message,
because the Bash interpreter will ignore it.
COMMENTBLOCK

echo "Exit value of above \"COMMENTBLOCK\" is $?."  # 0
# No error shown.
```

- **Attention:** The closing limit string, on the final line of a here document, must **start in the first character position**. There can be **no leading whitespac**e. Trailing whitespace after the limit string likewise causes unexpected behavior. The whitespace prevents the limit string from being recognized. **Except**, as Dennis Benzinger points out, if **using <<- to suppress tabs**.

- For those tasks too complex for a here document, consider using the **expect** scripting language, which was specifically designed for feeding input into interactive programs.

392. Here Strings

A **here strin**g can be considered as a stripped-down form of a here document. It consists of nothing more than **COMMAND <<< $WORD,** where **$WORD** is expanded and fed to the stdin of **COMMAN**D.

```
# Instead of:
if echo "$VAR" | grep -q txt  # if [[ $VAR = *txt* ]]
# etc.

# Try:
if grep -q "txt" <<< "$VAR"
then  #
  echo "$VAR contains the substring sequence \"txt\""
fi
```

```
String="This is a string of words."

read -r -a Words <<< "$String"
#  The -a option to "read"
#+ assigns the resulting values to successive members of an array.
```

- It is, of course, possible to feed the output of a here string into the **stdin** of a **loop.**

```
ArrayVar=( element0 element1 element2 {A..D} )

while read element ; do
  echo "$element" 1>&2
done <<< $(echo ${ArrayVar[*]})

# element0 element1 element2 A B C D
```

- **Prepending a line to a file**

```
read -p "Title: " title
cat - $file <<<$title > $file.new
```

I/O Redirection

393. Using exec

An **exec <filename** command redirects stdin to a file. From that point on, all **stdin** comes from that **file**, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using sedand/or awk.

Similarly, an **exec >filename** command redirects stdout to a designated file. This sends all command output that would normally go to stdout to that file.

- **exec N > filename** affects the entire script or current shell. Redirection in the PID of the script or shell from that point on has changed. However . . .
  N > filename affects only the newly-forked process, not the entire script or shell.

```
exec 6>&1           # Link file descriptor #6 with stdout.
                    # Saves stdout.
exec > $LOGFILE     # stdout replaced with file "logfile.txt".
.....
exec 1>&6 6>&-      # Restore stdout and close file descriptor #6.
```

- I/O redirection is a clever way of avoiding the dreaded **inaccessible variables within a subshell** problem.**Example 20-4. Avoiding a subshell**

```
cat myfile.txt | while read line;  #管道命令产生 subshell
```

```
            do {
              echo $line
              (( Lines++ ));  #  Incremented values of this variable
                              #+ inaccessible outside loop.
                              #  Subshell problem.
            }
            done

echo "Number of lines read = $Lines"     # 0
                                 # Wrong!
exec 3<> myfile.txt
while read line <&3
do {
  echo "$line"
  (( Lines++ ));                 #  Incremented values of this variable
                                 #+ accessible outside loop.
                                 #  No subshell, no problem.
}
done
exec 3>&-

echo "Number of lines read = $Lines"     # 8
```

394. Redirecting Code Blocks

Blocks of code, such as **while**, **until**, and **for** loops, even **if/then** test blocks can also incorporate **redirection** of stdin. Even a **function** may use this form of redirection (see Example 24-11). The < operator at the end of the code block accomplishes this.

Subshells

Running a shell script launches a new process, a **subshell**.

395. subshell

Definition: A subshell is a child process launched by a shell (or shell script).

- A subshell is a separate instance of the **command processor** -- the shell that gives you the prompt at the console or in an xterm window. Just as your commands are interpreted at the command-line prompt, similarly does a script **batch-process** a list of commands. Each shell script running is, in effect, a subprocess (child process) of the parent shell.
- A shell script can itself launch **subprocesses**. These subshells let the script do **parallel processing**, in effect executing multiple subtasks **simultaneously**.
- In general, an **external command** in a script forks off a **subprocess**, whereas a Bash **builtin** does not. For this reason, builtins execute more quickly and use fewer system resources than their external command equivalents.
- An external command invoked with an **exec** does not (usually) fork off a **subprocess / subshell.**

396. Command List within Parentheses

(command1; command2; command3; ...)

- A command list embedded between parentheses runs as a **subshell**.

- **Variables** in a **subshell** are not visible outside the block of code in the subshell. They are **not accessible** to the **parent process,** to the shell that launched the subshell. These are, in effect, variables **local** to the child process.

- **Directory changes** made in a **subshell** do not carry over to the **parent shell.**

- A subshell may be used to set up a "**dedicated environment**" for a command group. One **application** of such a "dedicated environment" is testing **whether a variable is defined.**

```
if (set -u; : $variable) 2> /dev/null
#-u  Treat unset variables as an error when substituting.
```

Another **application** is **checking for a lock file**:

```
if (set -C; : > lock_file) 2> /dev/null
#-C  If set, disallow existing regular files to be overwritten
#+       by redirection of output.
```

- Processes may execute in parallel within different subshells. This permits **breaking a complex task into subcomponents processed concurrently.**

```
(cat list1 list2 list3 | sort | uniq > list123) &
(cat list4 list5 list6 | sort | uniq > list456) &
      # Merges and sorts both sets of lists simultaneously.
```

```
# Running in background ensures parallel execution.
#
# Same effect as
#   cat list1 list2 list3 | sort | uniq > list123 &
#   cat list4 list5 list6 | sort | uniq > list456 &

wait   # Don't execute the next command until subshells finish.

   diff list123 list456
```

- A code block between **curly brackets** does not launch a subshell.
  { command1; command2; command3; . . . commandN; }

397. cmd1 | cmd2

cmd1、cmd2 产生 subshell.

Restricted Shells

Disabled commands in restricted shells

Running a script or portion of a script in restricted mode **disables certain commands** that would otherwise be available. This is a **security measure** intended to limit the privileges of the script user and to minimize possible damage from running the script.

The following commands and actions are disabled:

- **Using cd** to change the working directory.
- Changing the values of the **$PATH, $SHELL, $BASH_ENV,** or **$ENV** environmental variables.
- Reading or changing the **$SHELLOPTS**, shell environmental options.
- **Output redirection.**
- Invoking commands containing one or more **/'s**.
- **Invoking exec** to substitute a different process for the shell.
- Various other commands that would enable monkeying with or attempting to subvert the script for an unintended purpose.
- Getting out of **restricted** mode within the script.

```
#!/bin/bash

#  Starting the script with "#!/bin/bash -r"
#+ runs entire script in restricted mode.
set -r
# set --restricted    has same effect.
```

Process Substitution

- **Piping** the **stdout** of a command into the **stdin** of another is a powerful technique. But, what if you need to pipe the stdout of **multiple commands**? This is where **process substitution** comes in.
- **Process substitution** feeds the output of a process (or processes) into the stdin of another process.

398. Template

Command list enclosed within parentheses

>(command_list)

<(command_list)

- Process substitution uses **/dev/fd/<n>** files to send the results of the process(es) within parentheses to another process.
- This has the same effect as a **named pipe** (temp file), and, in fact, named pipes were at one time used in **process substitution**.

399. Instance

- Bash creates a pipe with two file descriptors, **--fIn** and **fOut**--.
  The stdin of true connects to fOut (dup2(fOut, 0)), then Bash passes a **/dev/fd/fIn** argument to **echo**. On systems lacking /dev/fd/<n> files, Bash may use temporary files.(括号内的标准输出与标准输入相连，标准输入就是/dev/fd/fIn 这个文件，所以**<(command_list) 命令执行后其实就是产生一个/dev/fd/fIn 文件，这个文件的内容就是命令的标准输出)**

```
xzz@xzz-ubuntu:~$ ls -l <(echo ii)
lr-x------ 1 xzz xzz 64 Aug 28 10:38 /dev/fd/63 -> pipe:[54557]
```

```
bash$ echo >(true) <(true)
/dev/fd/63 /dev/fd/62

bash$ wc <(cat /usr/share/dict/linux.words)
 483523  483523 4992010 /dev/fd/63
```

- Process substitution can compare the output of two different commands, or even the output of different options to the same command.

```
bash$ comm <(ls -l) <(ls -al) #相当于比较两个文件
```

- Some other usages and uses of process substitution:

```
read -a list < <( od -Ad -w24 -t u2 /dev/urandom ) #相当于从文件中读取数据，只读一行
#  Read a list of random numbers from /dev/urandom,
#+ process with "od"
#+ and feed into stdin of "read" . . .
```

```
tar cf >(bzip2 -c > file.tar.bz2) $directory_name
# Calls "tar cf /dev/fd/?? $directory_name", and "bzip2 -c > file.tar.bz2".
#
# Because of the /dev/fd/<n> system feature,
# the pipe between both commands does not need to be named.
#
# This can be emulated.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $directory_name
rm pipe
#       or
exec 3>&1
tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-
```

Functions

Like "real" programming languages, Bash has functions, though in a somewhat limited

implementation. A function is a **subroutine**, a code block that implements a set of operations, a "**black box**" that performs a specified task. Wherever there is **repetitive code**, when a task repeats with only slight variations in procedure, then consider using a function.

400. Template

function *function_name* {
command...
}
or
function_name () {
command...
}
This second form will cheer the hearts of C programmers (and is more `portable`).

A function may be "compacted" into a single line.In this case, however, a **semicolon** must follow the final command in the function.

```
fun2 () { echo "Even a single-command function? Yes!"; }
```

- Functions are called, triggered, simply by invoking their names. A function call is equivalent to a command.
- The function definition must precede the first call to it. There is no method of "declaring" the function, as, for example, in C.
- Functions may not be empty!
- It is even possible to nest a function within another function, although this is not very useful.
- Function declarations can appear in unlikely places, even where a command would otherwise go.
- Function names can take strange forms. It's a devious way to obfuscate the code in a script.
- What happens when different versions of the same function appear in a script? when a function is defined multiple times, **the final version is what is invoked**. This is not, however, particularly useful.

Complex Functions and Function Complexities

Functions may process arguments passed to them and return an **exit status** to the script for further processing.

401. Exit and Return

- **exit status**

  Functions **return a value**, called an **exit status**. This is analogous to the exit status returned by a command. The exit status may be explicitly specified by a return statement, otherwise it is the exit status of the last command in the function (0 if successful, and a non-zero error code if not). This exit status may be used in the script by referencing it as $?. This mechanism effectively permits script functions to have a "return value" similar to C functions.

- **return**

  Terminates a function. A return command optionally takes an **integer argument**, which is returned to the calling script as the "exit status" of the function, and this exit status is assigned to the variable **$?**.

  The largest positive integer a function can return is **255**. The return command is closely tied to the concept of exit status, which accounts for this particular limitation. Fortunately, there are various **workarounds** for those situations requiring a large integer return value from a function.

  A **workaround** for obtaining large integer "return values" is to simply **assign the "return value" to a global variable.**

```
alt_return_test ()
{
 fvar=$1
 Return_Val=$fvar
 return   # Returns 0 (success).
}
echo "return value = $Return_Val"    # 1
```

```
monthD="31 28 31 30 31 30 31 31 30 31 30 31"  # Declare as local?
echo "$monthD" | awk '{ print $'"${1}"' }'    # Tricky.
#Template for passing a parameter to embedded awk script:
#                              $'"${script_parameter}"'
#    Here's a slightly simpler awk construct:
#    echo $monthD | awk -v month=$1 '{print $(month)}'
#    Uses the -v awk option, which assigns a variable value
#+   prior to execution of the awk program block.
```

402. Redirection

A function is essentially a code block, which means its stdin can be **redirected** (as in Example 3-1).

```
file_excerpt ()    #  Scan file for pattern,
{                  #+ then print relevant portion of line.
 while read line  # "while" does not necessarily need [ condition ]
 do
  echo "$line" | grep $1 | awk -F":" '{ print $5 }'
  # Have awk use ":" delimiter.
 done
} <$file  # Redirect into function's stdin.

file_excerpt $pattern
```

There is an alternate, and perhaps less confusing method of redirecting a **function's stdin.** This involves redirecting the stdin to an **embedded bracketed** code block **within the function.**

```
# Instead of:
Function ()
{
 ...
} < file

# Try this:
Function ()
{
 {
   ...
  } < file
}

# Similarly,

Function ()  # This works.
{
 {
  echo $*
 } | tr a b
}

Function ()  # This doesn't work.
{
```

```
 echo $*
} | tr a b   # A nested code block is mandatory here.
```

403. Local Variables

A variable declared as local is one that is visible **only within the block of code** in which it appears. It has **local scope**. In a function, a local variable has meaning only within that function block.

Before a function is called, all variables declared within the function are invisible outside the body of the function, not just those explicitly declared as local.

A local variable declared in a function is also visible to functions called by the parent function.

- When declaring and setting a local variable in a single command, apparently the order of operations is to **first set the variable, and only afterwards restrict it to local scope**. This is reflected in the **return value**.

```
t=$(exit 1)
echo $?     # 1
          # As expected.
echo

function0 ()
{

echo "==INSIDE Function=="
echo "Global"
t0=$(exit 1)
echo $?     # 1
          # As expected.

echo
echo "Local declared & assigned in same command."
local t1=$(exit 1)
echo $?     # 0
          # Unexpected!
#  Apparently, the variable assignment takes place before
#+ the local declaration.
#+ The return value is for the latter.

echo
```

```
echo "Local declared, then assigned (separate commands)."
local t2
t2=$(exit 1)
echo $?      # 1
           # As expected.
}
```

404. Local variables and recursion.

Recursion is an interesting and sometimes useful form of **self-reference**. Consider a definition defined in terms of itself, an expression implicit in its own expression, a snake swallowing its own tail, or . . . a function that calls itself.

Be aware that recursion is resource-intensive and executes slowly, and is therefore generally **not appropriate in a script.**

```
fact ()  #求阶乘
{
 local number=$1
 #  Variable "number" must be declared as local,
 #+ otherwise this doesn't work.
 if [ "$number" -eq 0 ]
 then
  factorial=1    # Factorial of 0 = 1.
 else
  let "decrnum = number - 1"
  fact $decrnum  # Recursive function call (the function calls itself).
  let "factorial = $number * $?"
 fi

 return $factorial
}
```

405. Recursion Without Local Variables

Example 24-16. The Fibonacci Sequence

Example 24-17. The Towers of Hanoi

Aliases

A Bash alias is essentially nothing more than a keyboard shortcut, an abbreviation, a means of avoiding typing a long command sequence. If, for example, we include alias lm="ls -l | more" in the `~/.bashrc file`, then each lm typed at the command-line will automatically be replaced by a ls -l | more. This can save a great deal of typing at the command-line and avoid having to remember complex combinations of commands and options. Setting alias rm="rm -i" (interactive mode delete) may save a good deal of grief, since it can prevent inadvertently deleting important files.

In a script, aliases have very limited usefulness. It would be nice if aliases could assume some of the functionality of the Cpreprocessor, such as macro expansion, but unfortunately Bash does not expand arguments within the alias body. Moreover, a script fails to expand an alias itself within "compound constructs," such as if/then statements, loops, and functions. An added limitation is that an alias will not expand recursively. Almost invariably, whatever we would like an alias to do could be accomplished much more effectively with a function.

- **unalias**: Setting and unsetting an alias

List Constructs

The **and list** and **or list** constructs provide a means of processing a number of commands consecutively. These can effectively replace complex nested if/then or even case statements.

406. Chaining together commands

- **and list**

command-1 && command-2 && command-3 && ... command-n

- **or list**

command-1 || command-2 || command-3 || ... command-n

- The **exit status** of an and list or an or list is the exit status of the last command executed.

Arrays

Newer versions of Bash support **one-dimensional arrays.** Array elements may be initialized with the **variable[xx]** notation. Alternatively, a script may introduce the entire array by an explicit **declare -a variable** statement. To dereference (retrieve the contents of) an array element, use curly bracket notation, that is, **${element[xx]}**.

407. Initialization

- array_name[n]=value
- array_name=(XXX YYY ZZZ ...)
- array_name=([xx]=XXX [yy]=YYY ...)

408. Various array operations

```
array=( zero one two three four five )
echo ${#array[0]}      #  4
                       #  Length of first element of array.
echo ${#array}         #  4
                       #  Length of first element of array.
                       #  (Alternate notation)
echo ${#array[*]}      #  6
                       #  Number of elements in array.
echo ${#array[@]}      #  6
                       #  Number of elements in array.

array2=( [0]="first element" [1]="second element" [3]="fourth element" )
echo ${#array2[*]}     # 3     (number of elements in array)
```

409. String operations on arrays

把数组的每个元素都当成字符串来逐个进行操作。The string operations are performed on each of the elements in var[@] in succession. If the result is a zero length string, that element disappears in the resulting assignment. However, if the expansion is in quotes, the null elements remain.

- Trailing Substring Extraction

```
arrayZ=( one two three four five five )
echo ${arrayZ[@]:1}     # two three four five five
```

```
#                ^        All elements following element[0].
```

- Substring Removal

```
# Removes shortest match from front of string(s).

echo ${arrayZ[@]#f*r}   # one two three five five
#                       # Applied to all elements of the array.
                        # Matches "four" and removes it.

# Longest match from back of string(s)
echo ${arrayZ[@]%%t*e}  # one two four five five
#                      # Applied to all elements of the array.
                        # Matches "three" and removes it.
```

- Substring Replacement

```
# Replace first occurrence of substring with replacement.
echo ${arrayZ[@]/fiv/XYZ}   # one two three four XYZe XYZe
#                           # Applied to all elements of the array.

# Replace all occurrences of substring.
echo ${arrayZ[@]//iv/YY}    # one two three four fYYe fYYe
                            # Applied to all elements of the array.

# Delete all occurrences of substring.
# Not specifing a replacement defaults to 'delete' ...
echo ${arrayZ[@]//fi/}      # one two three four ve ve
#                          # Applied to all elements of the array.

# Replace front-end occurrences of substring.
echo ${arrayZ[@]/#fi/XY}    # one two three four XYve XYve
#                           # Applied to all elements of the array.

# Replace back-end occurrences of substring.
echo ${arrayZ[@]/%ve/ZZ}    # one two three four fiZZ fiZZ
#                           # Applied to all elements of the array.

echo ${arrayZ[@]/%o/XX}     # one twXX three four five five
#                           # Why?

#  Before reaching for a Big Hammer -- Perl, Python, or all the rest --
#  recall:
#    $( ... ) is command substitution.
```

```
#    A function runs as a sub-process.
#    A function writes its output (if echo-ed) to stdout.
#    Assignment, in conjunction with "echo" and command substitution,
#+   can read a function's stdout.
#    The name[@] notation specifies (the equivalent of) a "for-each"
#+   operation.
#  Bash is more powerful than you think!
```

410. Loading the contents of a script into an array

```
script_contents=( $(cat "$0") )  #  Stores contents of this script ($0)
                                 #+ in an array.

for element in $(seq 0 $((${#script_contents[@]} - 1)))
 do               #  ${#script_contents[@]}
                  #+ gives number of elements in the array.
                  #
                  #  Question:
                  #  Why is  seq 0  necessary?
                  #  Try changing it to seq 1.
 echo -n "${script_contents[$element]}"
                  # List each field of this script on a single line.
# echo -n "${script_contents[element]}" also works because of ${ ... }.
 echo -n " -- "    # Use " -- " as a field separator.
done
```

- In an array context, some Bash builtins have a slightly altered meaning. For example, unset deletes array elements, or even an entire array.

411. Some special properties of arrays

```
declare -a colors

read -a colors    # Enter at least 3 colors to demonstrate features below.
#  Special option to 'read' command,
#+ allowing assignment of elements in an array.
```

```
element_count=${#colors[@]}
# Special syntax to extract number of elements in array.
#      element_count=${#colors[*]} works also.
#  The "@" variable allows word splitting within quotes
#+ (extracts variables separated by whitespace).
# Again, list all the elements in the array, but using a more elegant method.
  echo ${colors[@]}          # echo ${colors[*]} also works.

# The "unset" command deletes elements of an array, or entire array.
unset colors[1]              # Remove 2nd element of array.
                             # Same effect as   colors[1]=
echo  ${colors[@]}           # List array again, missing 2nd element.

unset colors                 # Delete entire array.
                             #  unset colors[*] and
                             #+ unset colors[@] also work.
```

412. Of empty arrays and empty elements

- **An empty array** is not the same as an array with **empty elements**.

```
array0=( first second third )
  array1=( '' )   # "array1" consists of one empty element.
  array2=( )      # No elements . . . "array2" is empty.
  array3=(  )     # What about this array?
echo "Number of elements in array0 = ${#array0[*]}"  # 3
echo "Number of elements in array1 = ${#array1[*]}"  # 1  (Surprise!)
echo "Number of elements in array2 = ${#array2[*]}"  # 0
echo "Number of elements in array3 = ${#array3[*]}"  # 0
```

- Adding a superfluous declare -a statement to an array declaration may speed up execution of subsequent operations on the array.

413. The Sieve of Eratosthenes

求素数的算法

```
sift () # Sift out the non-primes.
{

let i=$LOWER_LIMIT+1
# Let's start with 2.
```

```
until [ "$i" -gt "$UPPER_LIMIT" ]
do

if [ "${Primes[i]}" -eq "$PRIME" ]
# Don't bother sieving numbers already sieved (tagged as non-prime).
then

 t=$i

 while [ "$t" -le "$UPPER_LIMIT" ]
 do
   let "t += $i "
   Primes[t]=$NON_PRIME
   # Tag as non-prime all multiples.
 done

fi

 let "i += 1"
done
}
```

```
#  This improved version of the Sieve, by Stephane Chazelas,
#+ executes somewhat faster.

# Must invoke with command-line argument (limit of primes).

UPPER_LIMIT=$1                # From command-line.
let SPLIT=UPPER_LIMIT/2       # Halfway to max number.

Primes=( '' $(seq $UPPER_LIMIT) )

i=1
until (( ( i += 1 ) > SPLIT ))  # Need check only halfway.
do
 if [[ -n ${Primes[i]} ]]
 then
   t=$i
   until (( ( t += i ) > UPPER_LIMIT ))
   do
     Primes[t]=
   done
```

```
 fi
done
echo ${Primes[*]}
```

Optimized Sieve of Eratosthenes

```
Primes=( '' $(seq ${UPPER_LIMIT}) )

typeset -i i t
Primes[i=1]='' # 1 is not a prime.
until (( ( i += 1 ) > (${UPPER_LIMIT}/i) ))  # Need check only ith-way.
 do                                  # Why?
   if ((${Primes[t=i*(i-1), i]}))
   # Obscure, but instructive, use of arithmetic expansion in subscript.
   then
     until (( ( t += i ) > ${UPPER_LIMIT} ))
       do Primes[t]=; done
   fi
 done

# echo ${Primes[*]}
echo   # Change to original script for pretty-printing (80-col. display).
printf "%8d" ${Primes[*]}
```

414. Emulating a push-down stack

Example 27-15

415. Simulating a two-dimensional array

Bash supports only one-dimensional arrays, though a little trickery permits simulating multi-dimensional ones.

Indirect References

The actual notation is **\$$var**, usually preceded by an eval (and sometimes an echo). This is called an **indirect reference**.

Indirect referencing in Bash is a multi-step process. **First**, take the name of a variable: varname. **Then**, reference it: $varname. **Then**, reference the reference: $$varname. **Then**, escape the first $: \$$varname. **Finally**, force a **reevaluation** of the expression and assign it: **eval newvar=\$$varname**. A more straightforward method is the **${!varname}** notation.

Of what practical use is indirect referencing of variables? It gives Bash a little of the **functionality of pointers in C**, for instance, in table lookup. And, it also has some other very interesting applications. . . .

Bash does not support pointer arithmetic, and this severely limits the usefulness of indirect referencing. In fact, indirect referencing in a scripting language is, at best, something of an **afterthought.**

/dev and /proc

A Linux or UNIX filesystem typically has the /dev and /proc special-purpose directories.

416. /dev

The /dev directory contains entries for the physical devices that may or may not be present in the hardware. **(**The entries in /dev provide mount points for physical and virtual devices. These entries use very little drive space. Some devices, such as /dev/null, /dev/zero, and /dev/urandom are virtual. They are not actual physical devices and exist only in software.**)**Appropriately enough, these are called **device files**. As an example, the hard drive partitions containing the mounted filesystem(s) have entries in /dev, as df shows.

- The /dev directory contains **loopback devices**, such as /dev/loop0. A loopback device is a gimmick that allows an ordinary file to be accessed as if it were a **block device**. This permits **mounting an entire filesystem within a single large file.** See Example 17-8 and Example 17-7.

- A few of the pseudo-devices in /dev have other specialized uses, such as **/dev/null, /dev/zero, /dev/urandom, /dev/sda1** (hard drive partition),**/dev/udp** (User Datagram Packet port), and **/dev/tcp**.

- **NOTE:** A **block device** reads and/or writes data in chunks, or blocks, in contrast to a character device, which acesses data incharacter units. Examples of block devices are hard drives, CDROM drives, and flash drives. Examples of **character device**s are keyboards, modems, sound cards.

417. /proc

The **/proc** directory is actually a **pseudo-filesystem**. The files in /proc mirror currently running system and kernel processes and contain information and statistics about them.

- Shell scripts may extract data from certain of the files in /proc. Certain system commands, such as **procinfo, free, vmstat, lsdev,** and **uptime** do this as well.

- The /proc directory contains subdirectories with unusual **numerical names**. Every one of these names maps to the **process ID** of a currently running process. Within each of these subdirectories, there are a number of files that hold useful information about the corresponding process. The **stat** and **status files** keep running statistics on the process, the **cmdline file** holds the command-line arguments the process was invoked with, and the **exe file** is a symbolic link to the complete path name of the invoking process. There are a few more such files, but these seem to be the most interesting from a scripting standpoint.

- In general, **it is dangerous to write to the files in /proc,** as this can corrupt the filesystem or crash the machine.

Network Programming

A Linux system has quite a number of tools for accessing, manipulating, and troubleshooting network connections. We can incorporate some of these tools into scripts -- scripts that expand our knowledge of networking, useful scripts that can facilitate the administration of a network.

Example 30-1. Print the server environment

Example 30-2. IP addresses

Of Zeros and Nulls

418. /dev/null

Think of /dev/null as a **black hole**. It is essentially the equivalent of a **write-only file.** Everything written to it disappears. Attempts to read or output from it result in nothing. All the same, **/dev/null** can be quite useful from both the command-line and in scripts.

- **Suppressing output**
- **Deleting contents of a file**, but preserving the file itself, with all attendant permissions.

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   has same effect, but does not spawn a new process.

ln -s /dev/null ~/.netscape/cookies
# All cookies now get sent to a black hole, rather than saved to disk.
```

419. /dev/zero

Like /dev/null, /dev/zero is a **pseudo-device file**, but it actually **produces a stream of nulls** (**binary zeros, not the ASCII kind**). Output written to /dev/zero disappears, and it is fairly difficult to actually read the nulls emitted there, though it can be done with **od** or a **hex editor**. The **chief use** of /dev/zero is **creating an initialized dummy file of predetermined length intended as a temporary swap file.**

Debugging

The Bash shell contains no built-in debugger, and only bare-bones debugging-specific commands and constructs. Syntax errors or outright typos in the script generate cryptic error messages that are often of no help in debugging a non-functional script.

420. Tools for debugging

Tools for debugging non-working scripts include

1. **Inserting echo statements** at critical points in the script to trace the variables, and otherwise give a snapshot of what is going on.
   Even better is an echo that echoes only when debug is on.

```
debecho () {
 if [ ! -z "$DEBUG" ]; then
   echo "$1" >&2
   #         to stderr
 fi
}
```

2. Using the **tee** filter to check processes or data flows at critical points.

3. Setting option flags **-n -v -x**
   - **sh -n scriptname** checks for syntax errors without actually running the script. This is the equivalent of inserting **set -n** or **set -o noexec** into the script. Note that certain types of syntax errors can slip past this check.
   - **sh -v scriptname** echoes each command before executing it. This is the equivalent of inserting **set -v** or **set -o verbose** in the script.
   - The -n and -v flags work well together. **sh -nv scriptname** gives a verbose syntax check.
   - **sh -x scriptname** echoes the result each command, but in an abbreviated manner. This is the equivalent of inserting **set -x** or **set -o xtrace** in the script.
   - Inserting **set -u** or **set -o nounset** in the script runs it, but gives an unbound variable error message and aborts the script.

4. Using an **"assert" function** to test a variable or condition at critical points in a script. (This is an idea borrowed from C.)

```
#!/bin/bash
# assert.sh
##########################################################
assert ()                 #  If condition false,
{                         #+ exit from script
                          #+ with appropriate error message.
 E_PARAM_ERR=98
 E_ASSERT_FAILED=99


 if [ -z "$2" ]          #  Not enough parameters passed
```

```
 then                 #+ to assert() function.
  return $E_PARAM_ERR   #  No damage done.
 fi

 lineno=$2

 if [ ! $1 ]
 then
  echo "Assertion failed:  \"$1\""
  echo "File \"$0\", line $lineno"    # Give name of file and line number.
  exit $E_ASSERT_FAILED
 # else
 #   return
 #   and continue executing the script.
 fi
} # Insert a similar assert() function into a script you need to debug.
#################################################################
a=5
b=4
condition="$a -lt $b"     #  Error message and exit from script.
                          #  Try setting "condition" to something else
                          #+ and see what happens.

assert "$condition" $LINENO
# The remainder of the script executes only if the "assert" does not fail.

# Some commands.
# Some more commands . . .
echo "This statement echoes only if the \"assert\" does not fail."
# . . .
# More commands . . .

exit $?
```

5. Using the **$LINENO** variable and the **caller** builtin.

6. **Trapping at exit.**

   The exit command in a script triggers a signal 0, terminating the process, that is, the script itself. It is often useful to **trap the exit**, forcing a "printout" of variables, for example. **The trap must be the first command in the script.**

421. Trapping signals

- **trap**

  Specifies an action on receipt of a signal; also useful for debugging.

  NOTE: A signal is a message sent to a process, either by the kernel or another process, telling it to take some specified action (usually to **terminate**). For example, hitting a **Control-C** sends a user interrupt, an INT signal, to a running program.

  A simple instance:

```
trap '' 2
# Ignore interrupt 2 (Control-C), with no action specified.

trap 'echo "Control-C disabled."' 2
# Message when Control-C pressed.
```

  Trapping at exit

```
#!/bin/bash
# Hunting variables with a trap.

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
#  EXIT is the name of the signal generated upon exit from a script.
#
#  The command specified by the "trap" doesn't execute until
#+ the appropriate signal is sent.

echo "This prints before the \"trap\" --"
echo "even though the script sees the \"trap\" first."
echo

a=39

b=36

exit 0
#  Note that commenting out the 'exit' command makes no difference,
#+ since the script exits in any case after running out of commands.
```

  A Simple Implementation of a Progress Bar

```
#! /bin/bash
# Invoke this script with bash. It doesn't work with sh.
interval=1
```

```
long_interval=10

{
    trap "exit" SIGUSR1
    sleep $interval; sleep $interval
    while true
    do
      echo -n '.'     # Use dots.
      sleep $interval
    done; } &         # Start a progress bar as a background process.

pid=$!
trap "echo !; kill -USR1 $pid; wait $pid"  EXIT       # To handle ^C.

echo -n 'Long-running process '
sleep $long_interval
echo ' Finished!'

kill -USR1 $pid
wait $pid             # Stop the progress bar.
trap EXIT   # 接收自然退出的 exit 信号，由此可以看出信号的接收应该是就近原则

exit $?
```

Tracing a variable(每执行一行命令就会产生一个 DEBUG 信号)

```
#!/bin/bash

trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\""' DEBUG
# Echoes the value of $variable after every command.

variable=29

echo "  Just initialized \$variable to $variable."

let "variable *= 3"
echo "  Just multiplied \$variable by 3."

exit

#  The "trap 'command1 . . . command2 . . .' DEBUG" construct is
#+ more appropriate in the context of a complex script,
#+ where inserting multiple "echo $variable" statements might be
#+ awkward and time-consuming.
```

```
# Thanks, Stephane Chazelas for the pointer.


Output of script:

VARIABLE-TRACE> $variable = ""
VARIABLE-TRACE> $variable = "29"
 Just initialized $variable to 29.
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "87"
 Just multiplied $variable by 3.
VARIABLE-TRACE> $variable = "87"
```

trap '' SIGNAL (two adjacent apostrophes) **disables SIGNAL** for the remainder of the script. **trap SIGNAL restores the functioning of SIGNAL** once more. This is useful to protect a critical portion of a script from an undesirable interrupt.

```
trap '' 2  # Signal 2 is Control-C, now disabled.
      command
      ...
trap 2     # Reenables Control-C
```

Example 32-9.Running multiple processes (on an SMP box)

422. Internal variables for use by the debugger

- **$BASH_ARGC**

Number of command-line arguments passed to script, similar to $#.

- **$BASH_ARGV**

Final command-line parameter passed to script, equivalent ${!#}.

- **$BASH_COMMAND**

Command currently executing.

- **$BASH_EXECUTION_STRING**

The option string following the **-c** option to Bash.

- **$BASH_LINENO**

In a function, indicates the line number of the function call.

- **$BASH_REMATCH**

Array variable associated with **=~ conditional regex matching.**

- **$BASH_SOURCE**

This is the name of the script, usually the same as $0.

- **$BASH_SUBSHELL**

Options

Options are settings that change shell and/or script behavior.

- The **set** command enables options within a script. At the point in the script where you want the options to take effect, use **set -o option-name** or, in short form, **set -option-abbrev**. These two forms are equivalent.

```
#!/bin/bash

    set -o verbose
    # Echoes all commands before executing.
```

```
#!/bin/bash

    set -v
    # Exact same effect as above.
```

- To disable an option within a script, use **set +o option-name** or **set +option-abbrev**.

- An alternate method of enabling options in a script is to specify them immediately following the #! script header.

```
#!/bin/bash -x
    #
    # Body of script follows.
```

- It is also possible to enable script options from the command line. Some options that will not work with set are available this way. Among these are -i, force script to run interactive.

```
bash -v script-name
bash -o verbose script-name
```

- The following is a listing of some useful options. They may be specified in either abbreviated form (preceded by a single dash) or by complete name (preceded by a double dash or by -o).

Table 33-1. Bash options

Abbreviation	Name	Effect
-B	brace expansion	*Enable* brace expansion (default setting = *on*)
+B	brace expansion	*Disable* brace expansion

Abbreviation	Name	Effect
-C	noclobber	Prevent overwriting of files by redirection (may be overridden by >|)
-D	(none)	List double-quoted strings prefixed by $, but do not execute commands in script
-a	allexport	Export all defined variables
-b	notify	Notify when jobs running in background terminate (not of much use in a script)
-c ...	(none)	Read commands from **...**
checkjobs		Informs user of any open jobs upon shell exit. Introduced in version 4 of Bash, and still"experimental." *Usage:* shopt -s checkjobs (*Caution:* may hang!)
-e	errexit	Abort script at first error, when a command exits with non-zero status (except in until or while loops,if-tests, list constructs)
-f	noglob	Filename expansion (globbing) disabled
globstar	*globbing*star-match	Enables the ** globbing operator (version 4+ of Bash). *Usage:* shopt -s globstar
-i	interactive	Script runs in *interactive* mode
-n	noexec	Read commands in script, but do not execute them (syntax check)
-o Option-Name	(none)	Invoke the *Option-Name* option
-o posix	POSIX	Change the behavior of Bash, or invoked script, to conform to POSIX standard.
-o pipefail	pipe failure	Causes a pipeline to return the exit status of the last command in the pipe that returned a non-zero return value.
-p	privileged	Script runs as "suid" (caution!)
-r	restricted	Script runs in *restricted* mode (see Chapter 22).
-s	stdin	Read commands from stdin
-t	(none)	Exit after first command

Abbreviation	Name	Effect
-u	nounset	Attempt to use undefined variable outputs error message, and forces an exit
-v	verbose	Print each command to stdout before executing it
-x	xtrace	Similar to -v, but expands commands
-	(none)	End of options flag. All other arguments are positional parameters.
--	(none)	Unset positional parameters. If arguments given (*-- arg1 arg2*), positional parameters set to arguments.

Gotchas

- Assigning reserved words or characters to variable names.

```
case=value0       # Causes problems.
```

- Using a hyphen or other reserved characters in a variable name (or function name).

```
var-1=23
# Use 'var_1' instead.
```

- Using the same name for a variable and a function. This can make a script difficult to understand.

- Using whitespace inappropriately. In contrast to other programming languages, Bash can be quite finicky about whitespace.

```
var1 = 23   # 'var1=23' is correct.
# On line above, Bash attempts to execute command "var1"
# with the arguments "=" and "23".
```

- Not terminating with a semicolon the final command in a code block within curly brackets.

```
{ ls -l; df; echo "Done." }
# bash: syntax error: unexpected end of file

{ ls -l; df; echo "Done."; }
```

```
#                     ^     ### Final command needs semicolon.
```

- Assuming uninitialized variables (variables before a value is assigned to them) are "zeroed out". An uninitialized variable has a value of **null**, not zero.

- Mixing up **=** and **-eq** in a test. Remember, **= is for comparing literal variables and -eq for integers.**

- Misusing **string comparison operators.**

```
while [ "$number" < 5 ]    # Wrong! Should be:  while [ "$number" -lt 5 ]
hile [ "$number" \< 5 ]    #  1 2 3 4
do                         #
 echo -n "$number "        #  It *seems* to work, but . . .
 let "number += 1"         #+ it actually does an ASCII comparison,
done                       #+ rather than a numerical one.
```

- Attempting to use **let** to set string variables.

```
let "a = hello, you"
echo "$a"   # 0
```

- Sometimes variables within "test" brackets ([ ]) need to be quoted (double quotes). Failure to do so may cause unexpected behavior.

- Quoting a variable containing whitespace prevents splitting. Sometimes this produces unintended consequences.

- Commands issued from a script may fail to execute because the script owner lacks execute permission for them. If a user cannot invoke a command from the command-line, then putting it into a script will likewise fail. Try changing the attributes of the command in question, perhaps even setting the suid bit (as root, of course).

- Attempting to use - as a redirection operator (which it is not) will usually result in an unpleasant surprise.

```
command1 2> - | command2
# Trying to redirect error output of command1 into a pipe . . .
# . . . will not work.

command1 2>& - | command2  # Also futile.
```

- Using Bash-specific functionality in a Bourne shell script (**#!/bin/sh**) on a non-Linux machine may cause unexpected behavior. A Linux system usually aliases **sh** to **bash**, but this

does not necessarily hold true for a generic UNIX machine.

- Using **undocumented features** in Bash turns out to be a dangerous practice. In previous releases of this book there were several scripts that depended on the "feature" that, although the maximum value of an exit or return value was 255, that limit did not apply to negative integers. Unfortunately, in version 2.05b and later, that loophole disappeared.

- In certain contexts, a misleading exit status may be returned. This may occur when setting a local variable within a functionor when assigning an arithmetic value to a variable.
  The exit status of an arithmetic expression is not equivalent to an error code.

```
var=1 && ((--var)) && echo $var
#       ^^^^^^^^^^ Here the and-list terminates with exit status 1.
#                  $var doesn't echo!
echo $?   # 1
```

- A script with DOS-type newlines (\r\n) will fail to execute,
  since #!/bin/bash\r\n is not recognized, not the same as the expected#!/bin/bash\n. The fix is to convert the script to UNIX-style newlines.

```
#!/bin/bash

echo "Here"

unix2dos $0    # Script changes itself to DOS format.
chmod 755 $0   # Change back to execute permission.
            # The 'unix2dos' command removes execute permission.

./$0          # Script tries to run itself again.
            # But it won't work as a DOS file.

echo "There"

exit 0
```

- A shell script headed by **#!/bin/sh** will not run in full Bash-compatibility mode. **Some Bash-specific functions might be disabled.** Scripts that need complete access to all the Bash-specific extensions should start with **#!/bin/bash.**

- Putting whitespace in front of the **terminating limit string** of a **here document** will cause unexpected behavior in a script.

- A script may not **export** variables back to its parent process, the shell, or to the environment. Just as we learned in biology, a child process can inherit from a parent, but not vice versa.

- Setting and manipulating variables in a **subshell**, then attempting to use those same variables outside the scope of the subshell will result an unpleasant surprise.

- **Piping echo output to a read** may produce unexpected results. In this scenario, the read acts as if it were running in a **subshell**. Instead, use the **set** command (as in Example 15-18). **Note also that an echo to a 'read' works within a subshell. However, the value of the variable changes *only* within the subshell.**

```
#!/bin/bash
#  badread.sh:
#  Attempting to use 'echo and 'read'
#+ to assign variables non-interactively.

#   shopt -s lastpipe

a=aaa
b=bbb
c=ccc

echo "one two three" | read a b c
# Try to reassign a, b, and c.

echo
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
# Reassignment failed.

### However . . .
##  Uncommenting line 6:
#   shopt -s lastpipe
##+ fixes the problem!
### This is a new feature in Bash, version 4.2.

# ------------------------------

# Try the following alternative.

var=`echo "one two three"`
set -- $var
a=$1; b=$2; c=$3

echo "-------"
```

```
echo "a = $a"  # a = one
echo "b = $b"  # b = two
echo "c = $c"  # c = three
# Reassignment succeeded.

# ------------------------------

#  Note also that an echo to a 'read' works within a subshell.
#  However, the value of the variable changes *only* within the subshell.

a=aaa          # Starting all over again.
b=bbb
c=ccc
echo; echo
echo "one two three" | ( read a b c;
echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = one
# b = two
# c = three
echo "-----------------"
echo "Outside subshell: "
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
echo

exit 0
```

In fact, as Anthony Richardson points out, piping to any loop can cause a similar problem.

A lookalike problem occurs when **trying to write the stdout of a tail -f piped to grep.**

```
tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
#  The "error.log" file will not have anything written to it.
#  As Samuli Kaipiainen points out, this results from grep
#+ buffering its output.
#  The fix is to add the "--line-buffered" parameter to grep.
```

- Using "**suid**" commands within scripts is risky, as it may compromise system security.
- Using shell scripts for CGI programming may be problematic. Shell script variables are not "typesafe," and this can cause undesirable behavior as far as CGI is concerned. Moreover, it

is difficult to "cracker-proof" shell scripts.

- Bash does not handle the **double slash** (//) string correctly.
- Bash scripts written for Linux or BSD systems may need fixups to run on a commercial UNIX (or Apple OSX) machine. Such scripts often employ the GNU set of commands and filters, which have greater functionality than their generic UNIX counterparts. This is particularly true of such text processing utilites as **tr**.

Scripting With Style

Get into the habit of writing shell scripts in a structured and systematic manner. Even on-the-fly and "written on the back of an envelope" scripts will benefit if you take a few minutes to plan and organize your thoughts before sitting down and coding.

Herewith are a few stylistic guidelines. This is not (necessarily) intended as an Official Shell Scripting Stylesheet.

423. Unofficial Shell Scripting Stylesheet

- Comment your code. This makes it easier for others to understand (and appreciate), and easier for you to maintain.
- Add descriptive headers to your scripts and functions.

```
#!/bin/bash

#***********************************************#
#                 xyz.sh                        #
#          written by Bozo Bozeman              #
#              July 05, 2001                    #
#                                               #
#          Clean up project files.              #
#***********************************************#

E_BADDIR=85                   # No such directory.
projectdir=/home/bozo/projects    # Directory to clean up.

# ------------------------------------------------------ #
# cleanup_pfiles ()                                      #
```

```
# Removes all files in designated directory.              #
# Parameter: $target_directory                            #
# Returns: 0 on success, $E_BADDIR if something went wrong. #
# ------------------------------------------------------- #
cleanup_pfiles ()
{
```

- Avoid using "**magic numbers**", that is, "hard-wired" literal constants. Use meaningful variable names instead. This makes the script easier to understand and permits making changes and updates without breaking the application.

```
if [ -f /var/log/messages ]
then
  ...
fi
#  A year later, you decide to change the script to check /var/log/syslog.
#  It is now necessary to manually change the script, instance by instance,
#+ and hope nothing breaks.

# A better way:
LOGFILE=/var/log/messages  # Only line that needs to be changed.
if [ -f "$LOGFILE" ]
then
  ...
fi
```

- Choose **descriptive names** for variables and functions.

```
fl=`ls -al $dirname`                # Cryptic.
file_listing=`ls -al $dirname`      # Better.
```

- Use exit codes in a systematic and meaningful way.

```
E_WRONG_ARGS=95
...
...
exit $E_WRONG_ARGS
```

Ender suggests using the exit codes in **/usr/include/sysexits.h** in shell scripts, though these are primarily intended for C and C++ programming.

- Use standardized **parameter flags** for script invocation. Ender proposes the following set of flags.

```
-a      All: Return all information (including hidden file info).
```

```
-b      Brief: Short version, usually for other scripts.
-c      Copy, concatenate, etc.
-d      Daily: Use information from the whole day, and not merely
       information for a specific instance/user.
-e      Extended/Elaborate: (often does not include hidden file info).
-h      Help: Verbose usage w/descs, aux info, discussion, help.
       See also -V.
-l      Log output of script.
-m      Manual: Launch man-page for base command.
-n      Numbers: Numerical data only.
-r      Recursive: All files in a directory (and/or all sub-dirs).
-s      Setup & File Maintenance: Config files for this script.
-u      Usage: List of invocation flags for the script.
-v      Verbose: Human readable output, more or less formatted.
-V      Version / License / Copy(right|left) / Contribs (email too).
```

- Break complex scripts into simpler modules. Use functions where appropriate. See Example 37-4.

- Don't use a complex construct where a simpler one will do.

```
COMMAND
if [ $? -eq 0 ]
...
# Redundant and non-intuitive.

if COMMAND
...
# More concise (if perhaps not quite as legible).
```

Miscellany

Interactive and non-interactive shells and

scripts

- Non-interactive scripts can run in the background, but interactive ones hang, waiting for input that never comes. Handle that difficulty by having an expect script or embedded here document feed input to an interactive script running as a background job. In the simplest case, redirect a file to supply input to a read statement (read variable <file). These particular workarounds make possible general purpose scripts that run in either interactive or non-interactive modes.

- If a script needs to test whether it is running in an interactive shell, it is simply a matter of finding whether the prompt variable,**$PS1** is set. (If the user is being prompted for input, then the script needs to display a prompt.)

- Alternatively, the script can test for the presence of option "i" in the $- flag.

```
case $- in
*i*)    # interactive shell
;;
*)      # non-interactive shell
;;
```

- However, John Lange describes an alternative method, using the -t test operator.

```
fd=0   # stdin

#  As we recall, the -t test option checks whether the stdin, [ -t 0 ],
#+ or stdout, [ -t 1 ], in a given script is running in a terminal.
if [ -t "$fd" ]
then
  echo interactive
else
  echo non-interactive
fi
```

- Scripts may be forced to run in interactive mode with the **-i** option or with a **#!/bin/bash -i** header. Be aware that this can cause erratic script behavior or show error messages even when no error is present.

424. Shell Wrappers

A **wrapper** is a shell script that embeds a system command or utility, that accepts and

passes a set of parameters to that command. Wrapping a script around a complex command-line simplifies invoking it. This is expecially useful with **sed** and **awk.**

A **sed** or **awk** script would normally be invoked from the command-line by a **sed -e 'commands'** or **awk 'commands'**. Embedding such a script in a Bash script permits calling it more simply, and makes it reusable. This also enables combining the functionality of sed and awk, for example piping the output of a set of sed commands to awk. As a saved executable file, you can then repeatedly invoke it in its original form or modified, without the inconvenience of retyping it on the command-line.

425. Tests and Comparisons: Alternatives

For tests, the **[[ ]]** construct may be more appropriate than **[ ]**. Likewise, arithmetic comparisons might benefit from the **(( ))** construct.

426. Recursion: a script calling itself

"Colorizing" Scripts

The ANSI escape sequences set screen attributes, such as bold text, and color of foreground and background. DOS batch filescommonly used ANSI escape codes for color output, and so can Bash scripts.

Example 36-12. Drawing a box

- The simplest, and perhaps most useful ANSI escape sequence is **bold text**, **\033[1m ... \033[0m**. The **\033** represents an escape, the "**[1**" turns on the bold attribute, while the "[0" switches it off. **The "m" terminates each term of the escape sequence.**

```
bash$ echo -e "\033[1mThis is bold text.\033[0m"
```

- A similar escape sequence switches on the underline attribute (on an rxvt and an aterm).

```
bash$ echo -e "\033[4mThis is underlined text.\033[0m"
```

- The **tput sgr0** restores the terminal settings to normal.
- Since **tput sgr0** fails to restore terminal settings under certain circumstances, **echo -ne "\E[0m"** may be a better choice.
- Use the following template for writing colored text on a colored background.

echo -e '\E[COLOR1;COLOR2mSome text goes here.'

```
bash$ echo -e '\E[1;33;44m'"BOLD yellow text on blue background"; tput sgr0
```

Color	Foreground	Background
black	30	40
red	31	41
green	32	42
yellow	33	43
blue	34	44
magenta	35	45
cyan	36	46
white	37	47

Optimizations

Most shell scripts are quick 'n dirty solutions to non-complex problems. As such, optimizing them for speed is not much of an issue. Consider the case, though, where a script carries out an important task, does it well, but runs too slowly. Rewriting it in a compiled language may not be a palatable option. The simplest fix would be to rewrite the parts of the script that slow it down. Is it possible to apply principles of code optimization even to a lowly shell script?

- Check the loops in the script. Time consumed by repetitive operations adds up quickly. If at all possible, remove time-consuming operations from within loops.
- Use builtin commands in preference to system commands. Builtins execute faster and usually do not launch a subshell when invoked.
- Avoid unnecessary commands, particularly in a **pipe**.
- Use the **time** and **times** tools to profile computation-intensive commands. Consider rewriting time-critical code sections in C, or even in assembler.
- Try to minimize file I/O. Bash is not particularly efficient at handling files, so consider using more appropriate tools for this within the script, such as **awk** or **Perl**.

- Write your scripts in a modular and coherent form, so they can be reorganized and tightened up as necessary. Some of the optimization techniques applicable to high-level languages may work for scripts, but others, such as loop unrolling, are mostly irrelevant. Above all, use common sense.

Assorted Tips

427. Ideas for more powerful scripts

- You have a problem that you want to solve by writing a Bash script. Unfortunately, you don't know quite where to start. One method is to plunge right in and code those parts of the script that come easily, and write the hard parts as pseudo-code.

- To keep a record of which user scripts have run during a particular session or over a number of sessions, add the following lines to each script you want to keep track of. This will keep a continuing file record of the script names and invocation times.

```
# Append (>>) following to end of each script tracked.
whoami>> $SAVE_FILE    # User invoking the script.
echo $0>> $SAVE_FILE   # Script name.
date>> $SAVE_FILE      # Date and time.
echo>> $SAVE_FILE      # Blank line as separator.

#  Of course, SAVE_FILE defined and exported as environmental variable in ~/.bashrc
#+ (something like ~/.scripts-run)
```

- The >> operator appends lines to a file. What if you wish to prepend a line to an existing file, that is, to paste it in at the beginning?

```
file=data.txt
title="***This is the title line of data text file***"

echo $title | cat - $file >$file.new
```

  This is a simplified variant of the Example 19-13 script given earlier. And, of course, sed can also do this.

```
cat - $file <<<$title > $file.new
```

- A shell script may act as an embedded command inside another shell script, a Tcl or wish script, or even a Makefile. It can be invoked as an external shell command in a C program using the system() call, i.e., **system("script_name")**;.

- Setting a variable to the contents of an embedded sed or awk script increases the readability of the surrounding **shell wrapper.** See Example A-1 and Example 15-20.

```
awkscript='{ total += $ENVIRON["column_number"] }
END { print total }'
# Yes, a variable can hold an awk script.
```

- Put together files containing your favorite and **most useful definitions and functions**. As necessary, "**include**" one or more of these "library files" in scripts with either the **dot (.)** or **source** command.

- Use special-purpose comment headers to increase clarity and legibility in scripts.

```
## Caution.
rm -rf *.zzy   ##  The "-rf" options to "rm" are very dangerous,
               ##+ especially with wild cards.
```

- A particularly clever use of **if-test** constructs is for **comment block**s.

```
COMMENT_BLOCK=
#  Try setting the above variable to some value
#+ for an unpleasant surprise.

if [ $COMMENT_BLOCK ]; then

Comment block --
==================================
This is a comment line.
This is another comment line.
This is yet another comment line.
==================================

echo "This will not echo."

Comment blocks are error-free! Whee!

fi
```

- Using the **$**? exit status variable, a script may **test if a parameter contains only digits**, so it can be treated as an integer.

```
test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# An integer is either equal to 0 or not equal to 0.
# 2>/dev/null suppresses error message.
if [ $? -ne "$SUCCESS" ]
```

- The 0 - 255 range for function return values is a severe limitation. Global variables and other workarounds are often problematic. An alternative method for a function to communicate a value back to the main body of the script is to have the function write to stdout (usually with echo) the "return value," and assign this to a variable. This is actually a variant of **command substitution**.

- Using the `double-parentheses construct`, it is possible to use **C-style syntax** for setting and incrementing/decrementing variables and in **for** and **while** loops.
  See Example 11-12 and Example 11-17.

- Setting the path and umask at the beginning of a script makes it more `portable` -- more likely to run on a "foreign" machine whose user may have bollixed up the **$PATH** and **umask**.

- A useful scripting technique is to repeatedly feed the output of a filter (by piping) back to the same filter, but with a different set of arguments and/or options. Especially suitable for this are **tr** and **grep**.

```
# From "wstrings.sh" example.

wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

- Running a script on a machine that relies on a command that might not be installed is dangerous. Use **whatis** to avoid potential problems with this.

- An **if-grep test** may not return expected results in an error case, when text is output to stderr, rather that stdout.

```
if ls -l nonexistent_filename | grep -q 'No such file or directory'
  then echo "File \"nonexistent_filename\" does not exist."
fi
```

  Redirecting **stderr** to **stdout** fixes this.

```
if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'
#                               ^^^^
  then echo "File \"nonexistent_filename\" does not exist."
fi
```

- If you absolutely must access a subshell variable outside the subshell, here's a way to do it.

```
TMPFILE=tmpfile               # Create a temp file to store the variable.

(   # Inside the subshell ...
```

```
inner_variable=Inner
echo $inner_variable
echo $inner_variable >>$TMPFILE  # Append to temp file.
)

   # Outside the subshell ...

echo; echo "-----"; echo
echo $inner_variable            # Null, as expected.
echo "-----"; echo

# Now ...
read inner_variable <$TMPFILE    # Read back shell variable.
rm -f "$TMPFILE"                 # Get rid of temp file.
echo "$inner_variable"           # It's an ugly kludge, but it works.
```

- The run-parts command is handy for running a set of command scripts in a particular sequence, especially in combination with cron or at.

428. Widgets

It would be nice to be able to invoke X-Windows widgets from a shell script. There happen to exist several packages that purport to do so, namely Xscript, Xmenu, and widtools. The first two of these no longer seem to be maintained. Fortunately, it is still possible to obtain widtools here.

- The **xmessage** command is a simple method of popping up a message/query window. For example:

```
xmessage Fatal error in script! -button exit
```

- The latest entry in the widget sweepstakes is **zenity**. This utility pops up GTK+ dialog widgets-and-windows, and it works very nicely within a script.

```
get_info ()
{
 zenity --entry      #  Pops up query window . . .
                     #+ and prints user entry to stdout.

                     #  Also try the --calendar and --scale options.
}

answer=$( get_info )   #  Capture stdout in $answer variable.

echo "User entered: "$answer""
```

Security Issues

429. Infected Shell Scripts

A brief warning about script security is indicated. A shell script may contain a worm, trojan, or even a virus. For that reason, never run as root a script (or permit it to be inserted into the system startup scripts in /etc/rc.d) unless you have obtained said script from a trusted source or you have carefully analyzed it to make certain it does nothing harmful.

Various researchers at Bell Labs and other sites, including M. Douglas McIlroy, Tom Duff, and Fred Cohen have investigated the implications of shell script viruses. They conclude that it is all too easy for even a novice, a "script kiddie," to write one.

Here is yet another reason to learn scripting. Being able to look at and understand scripts may protect your system from being compromised by a rogue script.

430. Hiding Shell Script Source

For security purposes, it may be necessary to render a script unreadable. If only there were a utility to create a stripped binary executable from a script. Francisco Rosales' `shc -- generic shell script compiler` does exactly that.

Unfortunately, according to `an article` in the October, 2005 Linux Journal, the binary can, in at least some cases, be decrypted to recover the original script source. Still, this could be a useful method of keeping scripts secure from all but the most skilled hackers.

431. Writing Secure Shell Scripts

Dan Stromberg suggests the following guidelines for writing (relatively) secure shell scripts.

- Don't put secret data in ironment variables.
- Don't pass secret data in an external command's arguments (pass them in via a **pipe** or **redirection** instead).
- Set your **$PATH** carefully. Don't just trust whatever path you inherit from the caller if your script is running as root. In fact, whenever you use an environment variable inherited from the caller, think about what could happen if the caller put something misleading in the variable, e.g., **if the caller set $HOME to /etc.**

Portability Issues

This book deals specifically with Bash scripting on a GNU/Linux system. All the same, users of sh and ksh will find much of value here.

As it happens, many of the various shells and scripting languages seem to be converging toward the POSIX 1003.2 standard. Invoking Bash with the **--posix** option or inserting a **set -o posix** at the head of a script causes Bash to conform very closely to this standard. Another alternative is to use a **#!/bin/sh** sha-bang header in the script, rather than **#!/bin/bash.** Note that /bin/sh is alink to /bin/bash in Linux and certain other flavors of UNIX, and a script invoked this way disables extended Bash functionality. Most Bash scripts will run as-is under ksh, and vice-versa, since Chet Ramey has been busily porting ksh features to the latest versions of Bash.

On a commercial UNIX machine, scripts using GNU-specific features of standard commands may not work. This has become less of a problem in the last few years, as the GNU utilities have pretty much displaced their proprietary counterparts even on "big-iron" UNIX. `Caldera's release of the source` to many of the original UNIX utilities has accelerated the trend.

Bash, versions 2, 3, and 4

Bash version 4

432. Associative arrays

An associative array can be thought of as a set of **two linked arrays** -- one holding the data, and the other the keys that index the individual elements of the data array.

```
#!/bin/bash4
# fetch_address.sh

declare -A address
#       -A option declares associative array.

address[Charles]="414 W. 10th Ave., Baltimore, MD 21236"
address[John]="202 E. 3rd St., New York, NY 10009"
address[Wilma]="1854 Vermont Ave, Los Angeles, CA 90023"

echo "Charles's address is ${address[Charles]}."
```

```
# Charles's address is 414 W. 10th Ave., Baltimore, MD 21236.
echo "${!address[*]}"   # The array indices ...
# Charles John Wilma
```

- Elements of the index array may include embedded space characters, or even leading and/or trailing space characters. However, index array elements containing only whitespace are not permitted.

```
address[   ]="Blank"   # Error!
```

- Enhancements to the case construct: the **;;&** and **;&** terminators.

```
#!/bin/bash4

test_char ()
{
 case "$1" in
   [[:print:]] )  echo "$1 is a printable character.";;&       # |
   # The ;;& terminator continues to the next pattern test.      |
   [[:alnum:]] )  echo "$1 is an alpha/numeric character.";;&  # v
   [[:alpha:]] )  echo "$1 is an alphabetic character.";;&     # v
   [[:lower:]] )  echo "$1 is a lowercase alphabetic character.";;&
   [[:digit:]] )  echo "$1 is an numeric character.";&         # |
   # The ;& terminator executes the next statement ...         # |
   %%%@@@@@    )  echo "*********************************";;    # v
#   ^^^^^^^^  ... even with a dummy pattern.
 esac
}

echo

test_char 3
# 3 is a printable character.
# 3 is an alpha/numeric character.
# 3 is an numeric character.
# *********************************
echo

test_char m
# m is a printable character.
# m is an alpha/numeric character.
# m is an alphabetic character.
# m is a lowercase alphabetic character.
```

```
echo

test_char /
# / is a printable character.

echo

# The ;;& terminator can save complex if/then conditions.
# The ;& is somewhat less useful.
```

- The new **coproc** builtin enables two parallel processes to communicate and interact.
  参看 coproc.sh 和 coproc2.sh.

- The new **mapfile** builtin makes it possible to load an array with the contents of a text file without using a loop or command substitution.

```
mapfile Arr1 < $0
# Same result as     Arr1=( $(cat $0) )
echo "${Arr1[@]}"  # Copies this entire script out to stdout.

# But, not the same as   read -a   !!!
read -a Arr2 < $0
echo "${Arr2[@]}"  # Reads only first line of script into the array.
```

- Parameter substitution gets **case-modification** operators.

```
var=veryMixedUpVariable
echo ${var}           # veryMixedUpVariable
echo ${var^}          # VeryMixedUpVariable
#        *            First char --> uppercase.
echo ${var^^}         # VERYMIXEDUPVARIABLE
#        **           All chars  --> uppercase.
echo ${var,}          # veryMixedUpVariable
#        *            First char --> lowercase.
echo ${var,,}         # verymixedupvariable
#        **           All chars  --> lowercase.
```

- The declare builtin now accepts the -l lowercase and -c capitalize options.

```
declare -l var1           # Will change to lowercase
var1=MixedCaseVARIABLE
echo "$var1"              # mixedcasevariable
# Same effect as            echo $var1 | tr A-Z a-z

declare -c var2           # Changes only initial char to uppercase.
```

```
var2=originally_lowercase
echo "$var2"              # Originally_lowercase
# NOT the same effect as     echo $var2 | tr a-z A-Z
```

- Brace expansion has more options.

```
echo {40..60..2}
# 40 42 44 46 48 50 52 54 56 58 60
# All the even numbers, between 40 and 60.

echo {60..40..2}
# 60 58 56 54 52 50 48 46 44 42 40
# All the even numbers, between 40 and 60, counting backwards.
# In effect, a decrement.
echo {60..40..-2}
# The same output. The minus sign is not necessary.

# But, what about letters and symbols?
echo {X..d}
# X Y Z [ ] ^ _ ` a b c d
# Does not echo the \ which escapes a space.
```

Zero-padding, specified in the first term within braces, prefixes each term in the output with the same number of zeroes.

```
bash4$ echo {010..15}
010 011 012 013 014 015


bash4$ echo {000..10}
000 001 002 003 004 005 006 007 008 009 010
```

- Substring extraction on positional parameters now starts with $0 as the zero-index. (This corrects an inconsistency in the treatment of positional parameters.)

- The new ** globbing operator matches filenames and directories recursively.

```
shopt -s globstar  # Must enable globstar, otherwise ** doesn't work.
                   # The globstar shell option is new to version 4 of Bash.

echo "Using *"; echo
for filename in *
do
 echo "$filename"
done   # Lists only files in current directory ($PWD).
```

```
echo "Using **"
for filename in **
do
  echo "$filename"
done   # Lists complete file tree, recursively.
```

- The new $BASHPID internal variable.
- There is a new builtin error-handling function named **command_not_found_handle.**
#!/bin/bash4

```
command_not_found_handle ()
{ # Accepts implicit parameters.
  echo "The following command is not valid: \""$1\"""
  echo "With the following argument(s): \""$2\"" \""$3\"""   # $4, $5 ...
} # $1, $2, etc. are not explicitly passed to the function.

bad_command arg1 arg2

# The following command is not valid: "bad_command"
# With the following argument(s): "arg1" "arg2"
```

Bash, version 4.1

- The **printf** command now accepts a -v option for setting array indices.
- Within **double brackets**, the > and < string comparison operators now conform to the **locale**. Since the locale setting may affect the sorting order of string expressions, this has side-effects on comparison tests within [[...]] expressions.
- The **read** builtin now takes a -N option (read -N chars), which causes the read to terminate after chars characters.
- **Here documents** embedded in $(...) command substitution constructs may terminate with a simple).

```
multi_line_var=$( cat <<ENDxxx
------------------------------
This is line 1 of the variable
This is line 2 of the variable
This is line 3 of the variable
------------------------------
```

```
ENDxxx)

#  Rather than what Bash 4.0 requires:
#+ that the terminating limit string and
#+ the terminating close-parenthesis be on separate lines.

# ENDxxx
# )
```

Bash, version 4.2

- Bash now supports the the \u and \U Unicode escape.

```
echo -e '\u2630'   # Horizontal triple bar character.
# Equivalent to the more roundabout:
echo -e "\xE2\x98\xB0"
                # Recognized by earlier Bash versions
```

- When the **lastpipe** shell option is set, the last command in a pipe doesn't run in a subshell.

- Negative array indices permit counting backwards from the end of an array.

```
array=( zero one two three four five )   # Six-element array.
#        0    1   2    3    4    5
#       -6   -5  -4   -3   -2   -1

# Negative array indices now permitted.
echo ${array[-1]}   # five
echo ${array[-2]}   # four

# But, you cannot index past the beginning of the array.
echo ${array[-7]}   # array: bad array subscript

# So, what is this new feature good for?

echo "The last element in the array is "${array[-1]}""
# Which is quite a bit more straightforward than:
echo "The last element in the array is "${array[${#array[*]}-1]}""
# And ...

index=0
let "neg_element_count = 0 - ${#array[*]}"
# Number of elements, converted to a negative number.
```

```
while [ $index -gt $neg_element_count ]; do
  ((index--)); echo -n "${array[index]} "
done  # Lists the elements in the array, backwards.
      # We have just simulated the "tac" command on this array.
```

Endnotes

433. 1

434. 1

435. 1

436. 1

437. 1

438. 1

439. 1

440. 1

441. 1

442. 1

443. 1

444. 1

445. 1

446. 1

447. 1