Fundamentals of UNIX

Student Workbook

itcourseware
Training Materials For IT Professionals

FUNDAMENTALS OF UNIX

Jeff Howell

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: Brandon Caldwell, Denise Geller, Jim McNally, and Rob Roselius.

Editor: Rick Sussenbach

Assistant Editor: Jan Waleri

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Rich Bagley, Jimmy Ball, Robert Barr, Bruce Batky, Andrew Boardman, Mark Copley, Tait Cyrus, Rick Feeney, Todd Gibson, Bob Johnson, Roger Jones, Roger Kobylarczyk, Rod Kuhns, Channing Lovely, Russ Martin, Joseph McGlynn, Dennis Miller, Michael Naseef, Don Nichols, Bill Parrette, Mike Reese, John Roach, Robert Seitz, Kevin Smith, George Trujillo, Danielle Waleri, Bob Williams, and Todd Wright, and the many students who have offered comments, suggestions, criticisms, and insights.

CONTENTS

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Manage files and directories by creating, copying, renaming, removing, editing, and performing other operations on them.
- Create and modify files using the **vi** editor.
- Use many UNIX utilities for text file manipulation, personal workspace management, file system management, and communication with other users.
- Take advantage of UNIX security mechanisms to protect files and programs from unauthorized access, and to configure shared access among work groups.
- Use the UNIX shell (command interpreter) to control the flow and processing of data through pipelines.
- Increase your productivity by generating file names using "wild card" pattern matching.
- Look up commands and other information in the online UNIX reference manuals.
- Explain the purpose of shell programs.
- Recognize applications for shell programs.
- Design and write shell programs of moderate complexity.
- Manage multiple concurrent processes to achieve higher utilization of UNIX.

COURSE OVERVIEW

- **Audience**: This course is designed for users who are new to UNIX. Since this is intended to be a first course in UNIX, "users" means, in no particular order, application users, programmers, system administrators, help desk staff, documenters, testers, managers, and anyone else who will be using UNIX daily or occasionally.

- **Prerequisites**: None

- **Classroom Environment:**

  - UNIX system with one terminal per student.

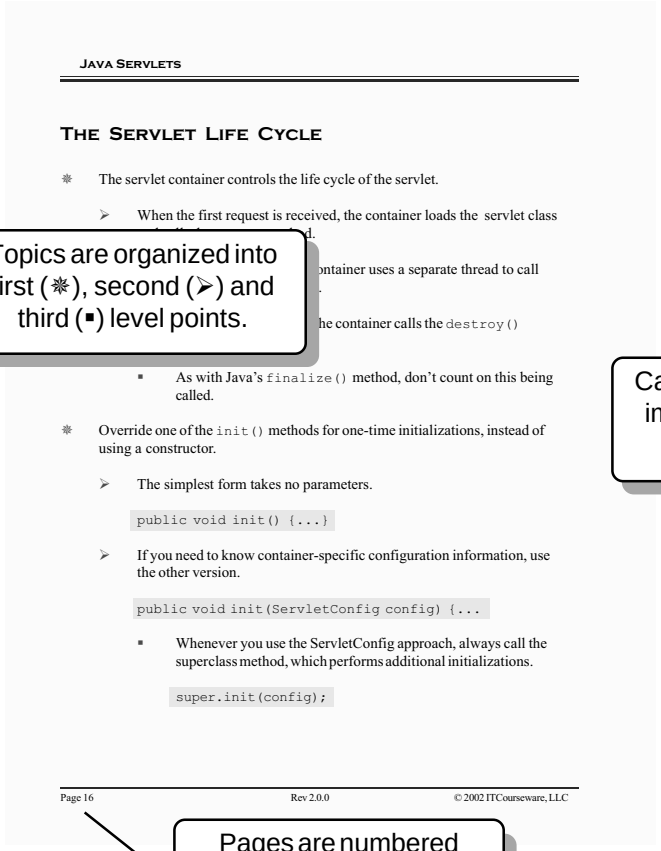
USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. *Hands On* sections provide opportunities for practical application of key concepts. *Try It* and *Investigate* sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

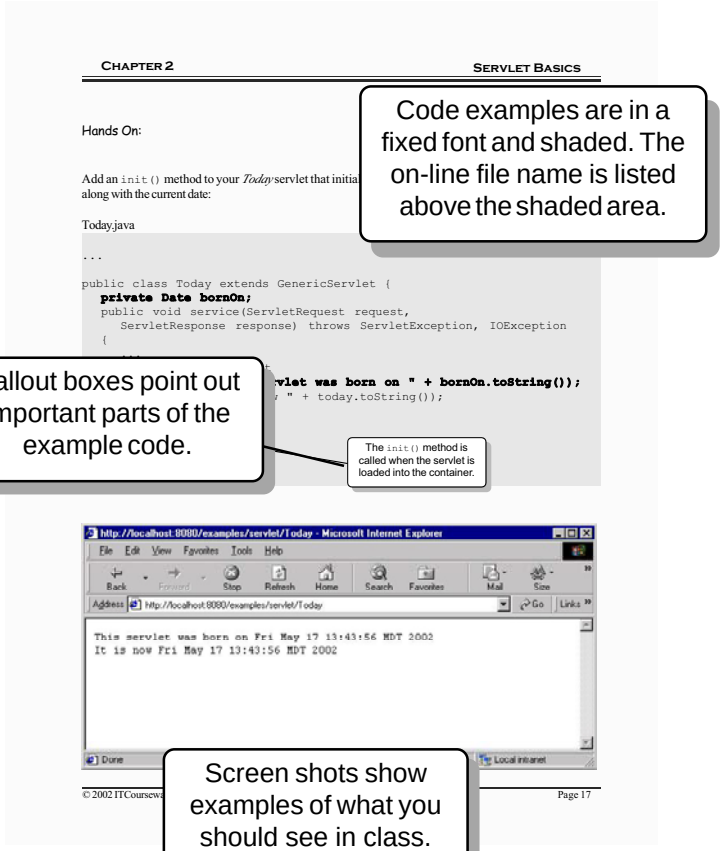
The Topic page provides the main topics for class-room discussion.

The Support page has additional information, examples and suggestions.



Topics are organized into first (❋), second (➢) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.



Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

SUGGESTED REFERENCES

Abrahams, Paul W. and Bruce Larson. 1995. ***UNIX for the Impatient***. Addison-Wesley, Reading, MA. ISBN 0201823764.

Anderson, Gail and Paul Anderson. 1986. ***UNIX C Shell Field Guide***. Prentice Hall, Upper Saddle River, NJ. ISBN 013937468X.

Blinn, Bruce. 1995. ***Portable Shell Programming***. Prentice Hall, Upper Saddle River, NJ. ISBN 0134514947.

Bolsky, Morris and David G. Korn. 1995. ***The New KornShell Command and Programming Language***. Prentice Hall, Upper Saddle River, NJ. ISBN 0131827006.

Hewlett-Packard Company. 1990. ***The Ultimate Guide to the VI and EX Text Editors***. Benjamin/ Cummings, Redwood City, CA. ISBN 0805344608.

Petersen, Richard. 1998. ***UNIX Clearly Explained***. Morgan Kaufman, San Francisco, CA. ISBN 0125521308.

Powers, Shelley, Tim O'Reilly, and Mike Loukides. 2002. ***UNIX Power Tools***. Third Edition. O'Reilly & Associates, Sebastapol, CA. ISBN 0596003307.

Robbins, Arnold. 2005. ***UNIX in a Nutshell***. Fourth Edition. O'Reilly & Associates, Sebastapol, CA. ISBN 0596100299.

Rosenblatt, Bill and Arnold Robbins. 2002. ***Learning the Korn Shell***. Second Edition. O'Reilly & Associates, Sebastapol, CA. ISBN 0596001959.

Sarwar, Syed Mansoor, Robert Koretsky and Syed Aqeel Sarwar. 2004. ***UNIX: The Textbook***. Addison-Wesley, Reading, MA. ISBN 032122731X.

Sobell, Mark G. 1994. ***A Practice Guide to the UNIX System***. Addison-Wesley, Reading, MA. ISBN 0805375651.

Vickery, Christopher. 1998. ***UNIX Shell Programmer's Interactive Workbook***. Prentice Hall, Upper Saddle River, NJ. ISBN 0130200646.

Waingrow, Kirk. 1999. ***UNIX Hints & Hacks***. Que, Indianapolis, IN. ISBN 0789719274.

CHAPTER 2 - GETTING STARTED

OBJECTIVES

- Provide a high-level overview of the background of UNIX.
- Log in and out of the system.
- Run a few commands.
- Change your password.
- Look up commands in the on-line UNIX reference manuals.

WHAT IS UNIX?

- UNIX is a general purpose, multi-tasking, multi-user, interactive, scalable computer operating system.
- HP-UX, SunOS, Solaris, AIX, BSD, SCO UNIX, System V.4, etc., are all versions of UNIX.
- This class teaches common features of all of the above.

WHY UNIX?

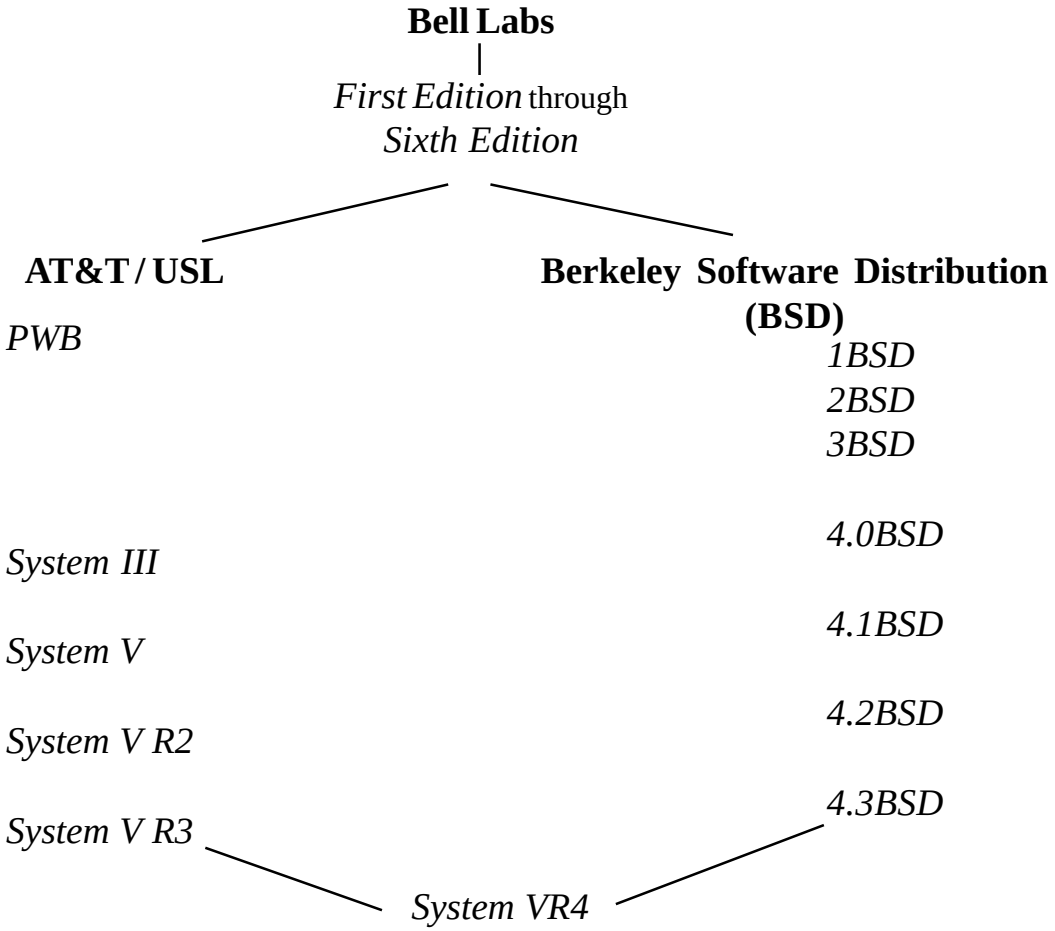
- In the 1980s, UNIX became commercially popular for several reasons:
  - Customer demand for the benefits of open systems:
    - Application portability
    - Vendor independence
    - Connectivity and interoperability in multi-vendor environments
    - User portability
  - New workstation hardware could be brought to market more quickly with an existing operating system.
  - Major vendors (such as Sun, DEC, HP, IBM) implemented UNIX-based product lines.
  - UNIX provides excellent networking capabilities.

UNIX is an *operating system* (OS), not a programming language. It supports software development in almost all languages, such as C, C++, Java, Perl, COBOL, Pascal, FORTRAN, BASIC, Smalltalk, LISP, etc. UNIX is also widely used in client/server configurations as the OS that supports server applications.

UNIX is the OS on which the Internet was developed and most World Wide Web sites are run on UNIX systems.

A BRIEF HISTORY OF UNIX

- UNIX was originally written at Bell Laboratories in 1969.
- In the mid-1970s, the University of California at Berkeley began making additions and enhancements to UNIX.
- In the early 1980s, AT&T began offering support for AT&T System III UNIX.



UNIX was originally designed and written by Ken Thompson for the purpose of aiding in computer science research. AT&T provided UNIX source code at a low cost to many universities, including the University of California At Berkeley (UCB). Berkeley UNIX, built upon the Sixth Edition, added over the years, many utilities, such as **vi** and **csh**; it was distributed under the name BSD (Berkley Software Distribution). Much research and development was done in the areas of file systems and networking. The history is well documented in several books, such as: McKusick, Marshall Kirk, et. al. 1996. ***The Design and Implementation of the 4.4BSD UNIX Operating System***. Addison-Wesley, Reading, MA. ISBN 0201549794.

AT&T, in 1982, merged several internal versions of UNIX and began licensing UNIX to vendors, such as Hewlett-Packard. In 1985, AT&T began shipping UNIX System V, and committed to support it and maintain backward compatibility in future versions of UNIX.

In 1989, AT&T's UNIX System Labs released System V, Release 4 (SVR4). SVR4 merged features of System V, BSD, and Microsoft's Xenix.

The diagram on the facing page is oversimplified with respect to the number of actual versions and variants of UNIX and its relatives, and with respect to the cross-influencing that the various versions have had on each other.

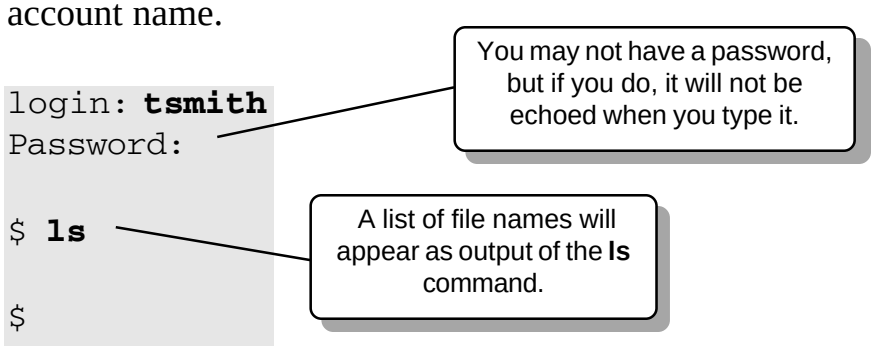
Major Versions

Hardware	Software	OS based on
Intel	SCO	Xenix
Intel	UnixWare	System V, R4.2
Hewlett Packard	HP-UX	System V
IBM	AIX	System V, R3.2
Pyramid	DC/OSX	System V
Sequent	Dynix	BSD and System V Derivatives
DEC	Ultrix	BSD
Sun	SunOS(4.x)	BSD
Sun	Solaris(2.x)	AT&T SVR4
MIPS	RISC/OS	System V

LOGGING IN

- To be able to use a UNIX system interactively, you must first have a user account that you can log into.

- UNIX will prompt by displaying **login:**, and you log in by typing your user account name.



- The dollar sign is your prompt.
  - The prompt you see may be different.
  - Prompts are configurable by the user.
- **ls** is a command that you type.
  - It means "**L**i**S**t the names of my files."
  - The **ls** command has nothing to do with logging in.
    - It is just a very common command that you can type after you log in.

Each user of a particular UNIX system has a unique user account name. This name is also known as a **User ID**. The user names on a system may follow a convention of "First initial - last name," or "Last name - first initial," or maybe even "First name - last initial." They may be employee numbers. Unless you are the System Administrator of a machine, your account name will be assigned to you by the System Administrator.

Who do you think **tsmith** is?

It is possible for an account to have no password. Most administrators require that all users put passwords on their accounts. We will see how to create and change a password shortly.

The prompt that UNIX displays is configurable by each user. Later in the class you will see how to change your prompt. In this workbook, we will always show the prompt as a dollar sign in courier font, like this: `$`. Commands that you should type will be in bold, like this:

```
$ ls
```

LOGGING OUT

- To log out, type **exit**.
- You can also log out by pressing <Ctrl>d**.**

```
$ exit

login:
```

- Now you are ready to log in again on your terminal.

The **exit** command will always log you out.

A *shell* is the program that runs on your terminal (or window) when you log in. It displays the prompt and waits for your command. Thus, a shell is sometimes called a *command interpreter*. The word "shell" comes from the idea that the program is acting as a shell around the UNIX operating system, i.e., acting as the interface between you and UNIX. The innermost part of the UNIX operating system is called the *kernel*, so the analogy is that we have a shell around a kernel.

There are several shell programs available on UNIX, and some provide additional ways to log out. The C shell (**csh**) lets you type **exit** or **logout**. All shells will let you log out with <Ctrl>d, but they may have an option set that ignores <Ctrl>d, so that you don't inadvertently log out by accidentally pressing <Ctrl>d.

We will study shells in much more detail later in the course. For now, just use **exit** to log out and be aware that there are other ways. (Of course, you might want to try **logout** and <Ctrl>d. Please experiment in this class!)

TRY A FEW MORE COMMANDS

- After logging in, try each of these commands to get some hands-on practice.
- Remember, the dollar sign ($) is the UNIX prompt – don't type the dollar sign!

`$ who`	Shows who is currently logged on (may not work properly on an X terminal).
`$ who am i`	Shows who you are.
`$ date`	Shows the date and time.
`$ cal`	Shows the calendar of the current month.
`$ cas`	A non-existent command that we use to demonstrate an error message.

CHANGING YOUR PASSWORD

- You can create a password if you don't have one, or change an existing one, with the **passwd** command.

- In the following example, the user selected the password **flower?**.

```
$ passwd
New password: flower?     (not echoed)
Re-enter new password: flower?     (not echoed)
```

- The new password must be used at the next login.

There are rules that apply to passwords. To read them, type:

```
$ man passwd
```

This will display the manual entry for the **passwd** command.

The next page describes how to use the online UNIX reference manuals.

If you forget your password, it must be reset by the System Administrator. Not even the S.A. can find out what your password is.

USING ON-LINE MANUALS

- Most UNIX systems keep electronic copies of the UNIX reference manuals in files on disk.
  - They are accessible with the **man** command (**man** is short for "manual").
- To read the manual entry for a particular command, for example **passwd**, type:

```
$ man passwd
```

- The entry for the **passwd** command will be displayed on your terminal, pausing when a full page has been displayed.
- Depending on what paging utility the **man** command is configured to use on your system, hit either the <Enter> key or the <Space> bar to display the next page.
- To quit without displaying the rest of the manual entry, use the **q** (for quit) key.
- Sometimes you see the names of commands written in the form ***command*(*N*)**, where ***N*** is a number.

```
passwd(1)
```

  - This means that **passwd** is documented in Section 1 of the UNIX reference manual.

Descriptions of commands in the on-line documentation are known for their terseness. It has been said that there are three rules to be aware of when reading UNIX manual entries:

Rule 1. Every word counts.
Rule 2. They only say it once.
Rule 3. You must know a fair amount of terminology before reading most entries.

We recommend that during your UNIX learning curve you have a more tutorially oriented book available, such as the Robbins or Sobell books listed under Suggested References in Chapter 1 of this workbook. Over time, the online manuals can become more efficient for you to look things up.

On another note, the **man** command works fine if you know the name of the command that you want to read about. But what if you want to change your password and you don't know the name of the command to use? Well, the **man** command has a useful option, the **-k** option, which will display the titles of all commands that contain the specified keyword. The **-k** option works only if **root** has run the **makewhatis** utility to create the keyword database. For example:

```
$ man -k passwd
   passwd (1)    - change login password and password attributes
   passwd (4)    - password file
   pwconv (1M)   - Installs and updates /etc/shadow with information from /etc/passwd
   rfpasswd (1M)  - change Remote File Sharing host password
   $
```

After perusing the titles of the commands, you can pick the one you need.

Although vendors vary on this, the UNIX manual typically is divided into topical sections as follows:

Section 1	Commands	Section 5	Misc. Facilities
Section 2	System Calls	Section 6	Games
Section 3	Library Calls	Section 7	Files and Devices
Section 4	File Formats	Section 8	System Admin.

LABS

These exercises are intended for you to get a little more practice with commands and with using system documentation.

❶ Study the manual entry for the **cal** command. What day of the week were you born on?

❷ Execute the **who** command with the command-line option that puts a header on the output. Options on UNIX commands are preceded by a hyphen.

Examples:

```
$ ls -l
$ man -k who
$ who -b
```

❸ Use the **spell** command to check the file *memo* for spelling errors. What is the output of **spell**? What are some options to **spell**?

Questions you need answers to in order to be successful on your system when you get back to work are:

1. What is my login name?
2. What is my password?
3. Which key backspaces?
4. How do I interrupt a command? (Usually <Break>, <Del>, or <Ctrl>C)

CHAPTER 7 - PERSONAL UTILITIES

OBJECTIVES

- Use the online calculator facilities to perform simple math.
- Display the current date and the calendar for any given month, for any given year.
- Read or write system news.
- Determine the current session's user id and which host you are on.
- Display user-defined "personal facts" for valid login ids.
- Create a file containing all input commands and resultant output.
- Get rid of the evidence (clear the screen).

THE DATE UTILITY

- **date** will display the current system time and date.

  Syntax:

  ```
  date [+format]
  ```

- The output may be formatted to meet your specific needs.
  - A plus sign precedes a user-specified format string, defining the appearance of the output.
  - Format specifiers are described on the facing page.
    - They must be preceded by a percent sign.
  - Use quotes if the format string contains spaces.

    Example:

    ```
    $ date +'Month: %B %t Day: %d Time: %T %p'
    Month: October   Day: 10          Time: 18:01:42 PM
    ```

- Formatted output from **date** is often used in shell programs to display the date and time in various formats, to create time-stamped filenames, and to store date/ time records in files.

Format specifiers are:

a	Use abbreviated weekday name.
A	Use full weekday name.
b	Use abbreviated month name.
B	Use full month name.
c	Country-specific date and time format.
d	Display day of month as 01 to 31.
D	Display date as %m/%d/%y.
e	Display day of month as 1 to 31.
H	Display hour as 00 to 23.
I	Display hour as 01 to 12.
j	Display day of year as 001 to 366.
m	Display month of year as 01 to 12.
M	Display minute as 00 to 59.
n	Insert a new-line character.
p	Display AM or PM indicator.
r	Time as %I:%M:%S %p (see above for field descriptions).
R	Time as %H:%M (see above for field descriptions).
S	Display second as 00 to 59.
t	Insert a tab character.
T	Display time as %H:%M:%S.
U	Display week number of year - 00 to 53.
w	Display numeric day of week (0 to 6) with Sunday = 0.
W	Display week number of year.
y	Display year within century as 00 to 99.
Y	Display year as ccyy.
Z	Display timezone name.

THE BC UTILITY

- Math is easily performed with the **bc** command.
- Operators include **+ - * / %** (remainder) & ^ (exponent).
- Enter an arithmetic expression, terminated by <Enter>.
- Terminate **bc** with **quit** or <Ctrl>D.

```
$ bc
(4+2)/10
0
quit or <Ctrl>D
$
```

- The last example illustrates an interesting point.
  - Floating point values must have their scale specified.
  - The scale (0 to 99) determines how many decimal places to display.
  - The default scale value is 0 (no decimal places will display).

```
$ bc
scale = 2
(4+2)/10
.60
quit
$
```


Like the **bc** utility, the **dc** utility (desk calculator) may be used to perform simple math (the **dc** utility is normally used by **bc**). The **dc** utility, however, uses Reverse-Polish Notation (RPN).

RPN requires that you enter the operands first, followed by the operator. You must explicitly request the result to be printed.

For example, to calculate the expression (2 + 4) / 10:

```
$ dc
2          enter the first operand
4          enter the next operand
+          enter the operator
10         enter the next operand
/          enter the next operand
p          p displays the current value
0          current value of (2+4)/10          -This is the result of integer division.
c          clear the stack
2k         display 2 numbers following the decimal   -This tells dc to display result in
                                                      floating point notation.
2          reenter the expression
4+10/p
.60        the result is displayed in floating point or decimal notation
q          quit the dc utility
```

Look at the manual entry for **dc** for more information.

Investigate:

How would you do the problem above such that it is entered on one line and the answer is displayed to 4 decimal places?

THE EXPR UTILITY

- The **expr** command may be used to perform integer math on the command line.
  - Operators include **+ - * / %**.
  - All results are integers.
  - **/** and ***** will be evaluated before **+** and **-.**
- To evaluate the expression 5 + 5 / 10:

```
$ expr 5 + 5 / 10
5
```

- Spaces are necessary between the arguments.
- The result is calculated as 5 + (5 / 10).
  - To return the result of the expression (5 + 5) / 10, you must apply parentheses, using the escape character \.

```
$ expr \( 5 + 5 \) / 10
1
```


The **expr** utility is commonly used in shell scripts to perform arithmetic.

Here is a short shell script example:

```
i=1
while [ $i -le 5 ]
do
   echo $i
   i=`expr $i + 1`    #Grave accents
done
```

THE cal UTILITY

- The **cal** command prints the calendar for a specified year.
  - You may specify a particular month of a particular year to display.
  - By default, **cal** prints the current month's calendar.
- The syntax for the command is:

```
cal [ [ month ] year ]
```

  - ***month*** must be a number between 1 and 12.
  - ***year*** must be a number between 1 and 9999.
- Note: **cal** understands the year **94** as year 94 in the first century A.D., not **1994**.

While the **cal** utility simply displays calendars, the somewhat related **calendar** utility provides a reminder service.

Use the **calendar** utility to display your "to-do" list for today and tomorrow. This command reads the *calendar* file in the current directory, and prints out any lines containing today's or tomorrow's date.

The date may appear anywhere on the line and **calendar** understands a variety of date formats. See the manual entry for **calendar(1)**.

If more than one date occurs on a line, that line will display on both dates.

Here is an example **calendar** file for October:

```
$ more calendar
Oct 6 Dentist appointment 2:30
10/7  Pay-day!!
10/21 Pay-day!!
12/20 DG's and TG's b-day
On Dec 1, make reservations for RM's 12/7 b-day

$ calendar
Oct 6 Dentist appointment 2:30
10/7  Pay-day!!
```

The *calendar* file is created with an editor.

By the way, what is today's date in the above example?

THE news UTILITY

- **news** is used to display items of interest to any who inquires.
- System news is deposited into files within the */var/news* directory.
  - This directory is usually accessible by everyone.
  - On many systems, news is in */usr/news*.
- Deposited news is accessed with the **news** command:

```
news [ -a|n|s ] [ items ]
```

- The **-n** option lists news items by name.
- The **-a** option displays all news items, even old ones.
- The **-s** option shows the number of items, or prints **No news**.
- By default, only new items are displayed to the user.
  - "new" is relative to the time stamp on the system-created *$HOME/.news_time* file.
  - Articles are displayed only one time, by default.
  - A specific item is displayed by requesting it by name on the command line.

The command **news -n** is commonly run in the system-wide login script */etc/profile*. If it isn't, you may want to include the **news** command in your *.profile*, so you will start each login session by displaying the news.

Also, *.profile* is a file that contains commands that are executed when you log in. It must be in your *home* directory.

THE ID UTILITY

- Use **id** to display the current user's login name.

```
$ id
uid=501(ringo) gid=1(other)
```

- **id** is sometimes used for security reasons in shell scripts to programmatically determine who ran the script.

Investigate:

The **su** command allows you to assume another user's identity. Ask your neighbor for his or her username and password (or ask the instructor for an account you may experiment with). Use the **su** command to assume that identity:

```
$ su new_login_name
Password:
$
```

Enter the **id** command. What displays?

Type **cd** to return to the *home* directory. Whose *home* directory are you in?

Read the **man** pages on **su**. What option to **su** must be supplied to simulate the user actually logging in?

THE UNAME UTILITY

- **uname** displays the system name of the UNIX system.
- It is used to determine which system you are on in a networked environment.

Some of the more commonly used options to **uname** are:

-m Print the machine hardware name.
-n Print the nodename.
-p Print the current host's processor type.
-r Print the operating system release.
-v Print the operating system version.
-a Print the **uname** information.

THE finger UTILITY

- Use the **finger** command to get information about each logged-in user.

```
$ finger
```

- You may specify **finger** on any valid login id.

```
$ finger ringo
Login name: ringo                In real life: ringo
Directory: /home/ringo           Shell: /bin/ksh
On since Oct 11 16:32:54 on pts/6 from ets4
50 minutes Idle Time
Unread mail since Tue Oct 11 16:39:16 1994
No Plan.
```


You may use the **finger** utility to inform your coworkers of your hectic schedule. The **finger** utility looks for the presence of two files in your *home* directory, *.plan* and *.project*. Use these files to assist those that need to know of your plans.

The *.plan* file usually contains information about your plans for the day. The *.project* file describes projects you are currently working on.

Assume the following *.plan* and *.project* are in Ringo's home directory:

```
$ cat .plan
I am in a meeting until noon today (Oct 11).
Afterward, I will be going to see the new Star Trek film
matinee.  I should be available during the previews and
credits.
```

```
$ cat .project
Currently working on the StarCon planning committee.
```

Then other users can:

```
$ finger ringo
Login name: ringo       In real life: ringo
Directory:  /home/ringo  Shell:  /bin/ksh
On since Oct 12 10:31:44 on pts/5         from ets4
Unread mail since Wed Oct 12 11:07:00 1994
Project: Currently working on the StarCon planning committee.
Plan:
I am in a meeting until noon today (Oct 11).
Afterward, I will be going to see the new Star Trek film
matinee.  I should be available during the previews and
credits.
```

Note:

Only the first line of the *.project* file is printed.

THE SCRIPT UTILITY

- To capture the input and output of a session, use the **script** command.

```
script [-a] [file]
```

- All text that displays on the screen will be captured in a file in the current directory.
  - The default filename is **typescript**.
  - You may specify a different filename:

```
$ script myscript
```

- To end the script session, type <Ctrl>D or **exit**.

```
$ script myscript
Script started, file is myscript
$ <enter some input>
<get some output>
$ exit
Script done, file is myscript
$
```

- The file will contain all input and output to the screen, including prompts.
- The file may be cleaned up using any editor.

What does the **-a** option to **script** do?

script is very useful for help desk customers when documenting bugs and other "features."

THE clear UTILITY

- Use **clear** to erase your screen.
- The prompt will redisplay after **clear**.

THE AT UTILITY

- The **at** command schedules a job to execute once at a specific time in the future.
  - Before you can use **at**, the administrator must place your user name in */usr/lib/cron/at.allow*.
- The execution time is specified on the **at** command line **(at** understands many time formats).
- **at** reads the scheduled commands from the keyboard, until <Ctrl>D is typed.

```
$ at 0515pm today
rm *.o
rm core
<Ctrl>D
$
```

- Each new **at** job is assigned a unique job ID, which is displayed after you press <Ctrl>D.
- The **atq** (or **at -l**) command lists your scheduled **at** job ids.
- The **atrm** (or **at -r**) command is used to remove, or unschedule, **at** jobs.

```
$ atrm id
```

- Any output of an **at** job is mailed to the user.

THE CRONTAB UTILITY

- The **crontab** command is used to schedule commands to run repeatedly at specified intervals.
  - Before you can use **crontab**, the administrator must place your user name in */usr/lib/cron/cron.allow*.

To schedule a command to run repeatedly, first create a **crontab** file, named whatever you like, with any editor, whose list of entries are in the following format, one entry per line:

```
minutes   hours   date_num month_num weekday_num command
```

minutes	is 0-59
hours	is 0-23
date_num	is 1-31
month	is 1-12
weekday_num	is 0 - 6 (0=Sunday)

A ***** in the field represents all valid values.

The first five fields of each entry specify the times the command will run, to a one-minute resolution. Blanks must separate the fields. Each field may have comma or dash separators. The last field (the command) may contain embedded blanks.

An example **crontab** file follows:

```
0  16  *  10-12  5  /home/rob/stuff/myscript
0,15,30,45 * * * * /home/denise/eod/cleanup
```

myscript will run at 4:00 pm every Friday, October through December. When will **cleanup** run?

After creating the **crontab** file, it must be submitted for processing with the **crontab** command. Any previous list is destroyed. Assuming you named your **crontab** file *cronlist*:

```
$ crontab cronlist
```

Use **crontab -r** to remove your list of cron jobs. Use **crontab -l** to list your current cron jobs. Any output of a **cron** job is mailed to the user.

LABS

❶ Use the **date** command to display the full name of the month, the Julian date, (i.e., the day of the year), the century, and the year within the century, (i.e., 1994, the time as hours, minutes, seconds AM or PM, as shown below).

```
October 292 1993 Time: 08:41:50 AM
```

See if you can come up with the **date** options to duplicate **date**'s default format.

❷ Experiment with the **bc**, **dc** and **expr** commands. Use **dc** to calculate the amount of money you will earn per year on an investment of $1250, at a 7% annual percentage rate. Use **bc** to calculate the amount of interest earned.

❸ Create some news items and read those created by other users. Experiment with **news** options.

❹ Use the **cal** command to determine the day of the week you were born on.

❺ Set up *.project* and *.plan* files in your *$HOME* directory. Work with a partner(s) to determine her/his/their projects and plans for the day using the **finger** command.

❻ Experiment with the **script** command. While the **script** command is running, **vi** *bigfile* and make some changes. Now exit from the **script** command by typing **exit** on the command line. Where is the output from the **script** command written? What is the content of the file resulting from the **vi** *bigfile*? Can you explain why the content is what it is?

❼ Use the **at** command to print a message to yourself. Print the message 5 minutes after the present time. Note whether the message appeared on your terminal display. Where does the message go? Ask the instructor for help on this.

CHAPTER 18 - SPECIAL VARIABLES

OBJECTIVES

- Parameterize the runtime behavior of scripts by passing in command line arguments.
- Perform usage error checking on passed parameters.
- Use loops to examine and process passed parameters.
- Create unique file names using the PID of the currently executing script.

COMMAND LINE ARGUMENTS

- When a shell script gets executed, user-provided arguments to the script are stored in special variables that are accessible to the script at runtime.

- The variables are automatically created and set by the shell, and are named **1**, **2**, **3**, etc.

Example:

echoem

```
# File: echoem
# This script prints its first three arguments.
echo $1 $2 $3
```

```
$ echoem hello
hello
$ echoem hello there
hello there
$ echoem hello there my friend
```

What did that last one **echo**?

Another example:

bakit

```
cp $1 $1.bak
chmod 400 $1.bak
```

```
$ bakit nameit
$ ls
```


Command-line arguments (parameters) are also called *positional parameters*, because their positions on the command line determine their names.

The Bourne shell (**sh**) can directly access only nine command-line arguments, **$1** through **$9**. This is not a limitation, though. We will see shortly how scripts can process command line arguments left to right, one at a time, without concern for how many there are.

The Korn shell (**ksh**) can directly access positional parameters in positions 10 and higher, by using the notation **${10}**. However, this is not commonly done, because of the mechanism for one-at-time-processing mentioned above.

$# - NUMBER OF ARGUMENTS

- The variable # is set to the number of arguments specified on the command line by the user when the script is executed.

  Example:

  counter

```
echo That is $# words.
```

```
$ counter This is a sentence.
That is 4 words.
$ counter apples grapes bananas
That is 3 words.
$ counter
That is 0 words.
$
```


The **$#** variable reference has several uses in a script. As we will see later, one frequent use is as a loop counter, to process each positional parameter one at a time, without knowing (or caring) how many there are.

Another use is when a script requires a certain number of arguments to fulfill its purpose in life, then it can check right at the start if it has enough. Look at *mycp*:

mycp
```
# This script will use cp to copy one file to another, but it will
# first check to see if the destination file exists.  If so, it will
# ask the user if overwriting is OK.

# Check for exactly two arguments
if [ $# -ne 2 ] ; then
   echo "Usage: $0 f1 f2"
   exit 1
fi

# Check for existence of $2
if [ -f $2 ] ; then
   echo "$2 already exists: overwrite? (y/n) \c"
   read a
   if [ "$a" = y ] ; then
      cp $1 $2
   fi
else
   cp $1 $2   # $2 does not exist, so just do the copy
fi
```

Note the reference to **$0** in the usage message. The name of the file by which the script is invoked is stored in **$0**.

THE shift COMMAND

- Many programmers design scripts that process command-line arguments so that the processing is done in a loop, accessing each variable one at a time, left to right.

- The key to this is the *shift command*, which promotes the positional parameter variables.

- When **shift** is executed in a script, here is what happens:

  $1 is shifted out and lost.

  $2 is renamed **$1**

  $3 is renamed **$2**

  . . . and so on

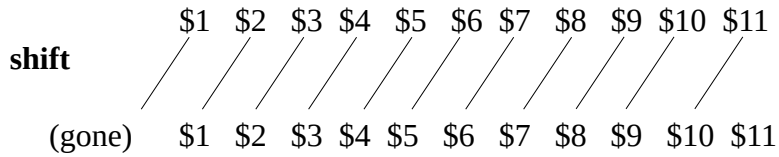
  $10 is renamed **$9**

  . . . and so on

- If you need **$1** later, you must save it before **shift**ing.

- Also, each time **shift** is executed, **$#** is decremented by 1.

You can think of **shift** as an operation that shifts all of the positional parameters to the left, throwing away $1.



$0 does not participate in the shift, and is not affected by it. The **#** variable is updated.

Actually, **shift** can take an integer parameter that will cause the shift to be repeated that many times. The parameter must be less than the current value of **$#**. Example: **shift 5** gets rid of **$1** through **$5**, and **$6** is placed in **$1**.

Here is an example that demonstrates the use of **shift**.

shifty

```
if [ $# -eq 0 ] ; then
   echo "Usage: shifty arg1 [ arg2 ... argn ]"
   exit 1                          # Premature exit
fi
echo Number of arguments supplied: $#.
echo $1 $2
shift
echo $1 $2
if [ $# -gt 3 ] ; then                # If more than 3 remaining
   shift 3
   echo $1 $2
else
   echo "That's enough shifting for now."
fi
```

Try running *shifty* several times with a different number of arguments each time. Try 1, 2, 3, 4, 5, 6, and 7 arguments.

$* - ALL ARGUMENTS

- The variable reference **$*** expands to all the arguments specified on the command line.

- **$*** is equivalent to:

```
$1 $2 $3 $4 ...
```

Example:

save_old

```
echo The following files will be saved in the old
directory:
echo $*
echo
for p in $*
do
     echo $p
     mv $p old/$p
     chmod 400 old/$p
done
```


The special variable **@** is similar to the ***** special variable.

$$ - PID OF SHELL

- The variable **$** contains the PID (Process ID) of the currently executing shell script.

- **$$** is commonly used to create uniquely named temporary files, because each PID is guaranteed to be unique by UNIX.

- The following example uses many features we have learned. (This script is not as simple as it looks.)

cleaner

```
# Process files containing pattern $1
#
grep -l $1 * > /tmp/cleaner$$
num_found=`wc -l < /tmp/cleaner$$`
echo Found $num_found files with pattern $1
for file in `cat /tmp/cleaner$$`
do
  echo $file
done
rm /tmp/cleaner$$
```


LABS

❶ Write a one-line script that **echo**es all of its arguments, no matter how many there are. Use **$***.

❷ Write a one-line script called *reverse* that will **echo** its arguments in reverse. Assume there will be no more than five arguments.

It should do this when run:

```
$ reverse tic tac toe
toe tac tic
$
```

Note: the solution is NOT: `echo toe tac tic`

❸ Modify the script in Exercise ❷ to require exactly five arguments. Print a usage message and exit if there aren't five arguments.

❹ Make a directory in your home directory named *Deleted*. Write a script called *purge* that accepts one command line argument as a file name, copies that file to the *Deleted* directory, then removes the file. Make sure that there is exactly one argument.

❺ Write a script to create a file in the current directory whose name ends with the PID of the script. Run your script a few times, doing an **ls** after each run. Remove the files when you are done.

