Unit 12
Good Practices and Review

Unit Objectives

After completing this unit, you should be able to:

- Write any serious script you need to
- Plan the activity
- Produce **good code**
- In this unit:
  - Planning and design
  - Documentation
  - Debugging
  - Performance issues
  - Guidelines for scripting
  - Course summary

Planning and Design

- As well as your favorite design methodology (Flow Charts, Data-Flow, SSADM, and so forth) consider:
  - Functionality – clearly defined specification
  - Modular design – use of functions, separate programs
  - Environment – variables, directories
  - File naming convention – for temporary files, results
  - Testing – individual units, integration tests, boundary conditions
  - Debugging code – do not forget the next maintainer

Use of Comments

- A good programmer uses comments in a program to:
  - Explain the purpose and function of the code at key points
  - Describe the use of variables
  - Explain complicated syntax
  - Give yourself the credit (or the blame) for your work
  - Mark corrections or additions

- Remember to update the comments with the code

Commenting Out

- Lines can be commented out using the # comment character:
  - `# command arg1 arg2`
  - No Shell interpretation is performed to the right of #
  - Legal anywhere, except as the only statement in a flow-control construction (`if`, `while`, `until`)

- The "null" command can be used where commenting out would not work:
  - `: command arg1 arg2`
  - Arguments are ignored, but processed as usual
  - Always returns 0 (true)

Script Layout

- Some things must be done in a certain order other things can be arranged for **good code**:
  - Shell control line (first in script)
    `#!/usr/bin/ksh` or `#!/bin/bash`
  - Header comments
  - Validation of options
  - Testing of arguments
  - Initialization of variables
  - Function definitions
  - Main code

Debugging Code

Shell options can help with syntax checking:

- To check the syntax of a script without running it

```
set -o noexec          or          set -n
```

- For the shell to print its input as it reads it

```
set -o verbose         or          set -v
```

- An execution trace displays each command before it is run and after command line processing

```
set -o xtrace          or          set -x
```

- For functions, use

```
typeset -ft function ...
```

DEBUG Traps

After each simple command the shell issues the fake signals

- **DEBUG**
- **ERR**
- **EXIT**

The order is **DEBUG**, **ERR**, then any other traps, and lastly **EXIT**

To display the environment after each command set this trap

```
trap  "set"  DEBUG
```

When a command has a non-zero exit status, the shell sends the **ERR** signal

For example, to see what signals are causing error exits set this trap

```
trap  "kill -l $?"  ERR
```

Maintaining Code

- Documentation: Design and comments
- Clarity
  - Code
  - Documentation
- Modularity
  - Main script
  - Use "good" functions or separate programs

Good Functions

- To write functions that are reliable and easy to maintain:
  - Avoid altering global variables inside a function
  - Define and export functions only when necessary
  - Do not change the working directory inside a function
  - Tidy up local temporary files

Performance Issues for Shell Scripts

- If performance is an issue
  - Do not guess
  - Measure
- Performance of a script means two areas:
  - That of the shell
  - That of the script
- Remember that you should work in this order
  - Make it work
  - Make it robust
  - Make it more efficient/faster

Timing Commands

- To report the elapsed, user and system time for a command or pipeline, use **`time`** in the Korn or Bash shells:
  - A reserved word (not a command)
  - Output is to standard error
  - Input or output redirection applies to the commands under test only
  - Return value is that of the commands under test

```
$ time find / -name 'unix*' -print|sort
/unix

/usr/lib/unixtomh                output from find
real      0m25.51s               wall clock time
user      0m1.56s
sys       0m11.01s
$ _
```

Times for Shells

- The **`times`** command displays how much time your current shell and all its subshells have consumed:

```
$ times
0m0.99s  0m15.37s
0m8.61s  0m33.21s
```

  - User and system timings given in hundredths of a second
  - First line for the current shell
  - Second line for the subshells

Shell Performance

- To increase the startup speed of a new shell:
  - Keep your history file (**`.sh_history`**) small
  - Minimize the size of any **`$ENV`** file
  - Use **`autoload`** with your functions (ksh)
  - Use **`FPATH`** with your functions
  - Use **`set -o nolog`** to prevent function definitions being logged in your history (ksh)
  - Use **`tracked aliases`** or **`hash`**es
  - Try to use an **`alias`** in place of a simple function
  - Set **`MAILCHECK`** greater than the 600 second default
  - In bash, use brace expansion, for example:
    `mkdir ../release_{src,doc}`

Shell Script Performance

Tips for faster performance shell scripts:

- Shell built-in commands run faster than UNIX built-ins
- Avoid command substitution where you can use **`${ }`** parameter expansions, let or pattern matching
- Note **`$(< file)`** is faster than **`$(cat file)`**
- Use multiple arguments rather than separate commands – for example, **`typeset -i a=3 b=4`**
- Use **`set -f`** or **`set -o noglob`** if not using pathname metacharacters
- Use **`{ }`** grouping that is faster than **`( )`**
- Apply I/O re-directions to the whole of a loop syntax
- Set the **`integer`** attribute for suitable variables and don't use **`$`** for them with arithmetic expressions

Good Rules to Follow

1. Documentation
2. Make backups
3. Try three times
4. Do not overlook the obvious
5. Try it, it might work
6. Never say never, always avoid always
7. There is usually another way to do it

Checkpoint

1. What allows you to document your program for future reference?

2. Why is it a good idea to plan and design before you code?

3. Which statement is faster and why?
   `$(< data.file)` or `$(cat data.file)`

4. What set options can help in debugging a script?

Unit Summary

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary

Course Summary

- Basic concepts
- Shell variables and parameters
- Exit status, return codes and traps
- Programming constructs – flow control
- Shell commands and features
- Arithmetic in shells
- Shell types and functions
- Regular expressions and text selection
- Productivity using *sed* and *awk*
- Summary – good practice, debugging, performance