UNIX and C Programming (COMP1000)

Lecture 7: Shell Scripting

Updated: 29th July, 2015

Department of Computing
Curtin University

Reading

For more information, see the weekly reading list on Blackboard.

- Unfortunately, Hanly and Koffman do not cover shell scripting.
- However, the `bash` manual page is extremely detailed:

```
[user@pc]$ man bash
```

- Type the slash (/) character, followed by a search term, to find specific information in the page.

Outline

Shell Languages

- Each different shell provides an interpreted "scripting" language.
- You can write "scripts" — mini-programs that are read and executed by the shell.
- We *could* do everything with C. However...
- Scripts are able to perform certain tasks much more easily.
- One line of a script may translate to many lines of C code.

Scripting vs. Programming

Scripting languages are usually:

- Interpreted, not compiled;
  - (Scripts are read and executed by an "interpreter" program.)
- Based on strings and string substitution;
- Intertwined with external programs or "commands";
  - (Scripts use external programs as a C program uses functions.)
- Geared towards manipulating files and processes;
- Not suitable for complex mathematical problems;
- Not suitable for writing large-scale software.

(There are exceptions. Some languages do not fall neatly into one category.)

Bash

- Bash is the de facto standard shell on most UNIX systems.
- Bash was inspired by the older "Bourne shell", and stands for "Bourne Again SHell".
- Despite radical differences, C and Bash are distantly related via ALGOL (as are most modern languages).
- Most other shells bear strong resemblances to Bash anyway.

Bash scripts can use the following programming-like features:

- variables and command-line parameters;
- `if` and `case` statements;
- `while` and `for` loops.

However, these all look a little bit different from C.

Hello World

This script will print "Hello world!":

```
echo Hello world\!
```

To explain:

- The script contains a single "`echo`" command, which just outputs its arguments.
- Normally, "`!`" is a special character.
- We've escaped it (removed its special meaning) using a "\" (backslash).

We can run this script as follows:

```
[user@pc]$ bash helloworld.sh
```

(Assuming we named the script "helloworld.sh".)

Hash Bang

- We could *also* run the script like a normal program:

```
[user@pc]$ ./helloworld.sh
```

  . . . if we make a couple of tweaks *beforehand*.
- First, can specify the interpreter inside the script itself:

```
#!/bin/bash
echo Hello world\!
```

  - The very first characters must be "`#!`" (or "hash-bang").
  - In this case, the script is run using /bin/bash (the full path to bash).
- Second, we need to make the script file executable:

```
[user@pc]$ chmod 755 helloworld.sh
```

  (We'll return to this later.)

Comments

- Comments start with a "#" (hash), and are single-line.

```
#!/bin/bash

# Now, I'm going to ask you that question once
# more. And if you say no, I'm going to shoot
# you through the head. Do you have any cheese
# at all?
echo No, I was deliberately wasting your time.
```

(Most scripting languages use "#" to indicate comments.)

- Commenting serves the same purpose as in C, Java, etc.
- Use them liberally! (But make sure they're relevant. . .)

Variables — Revision From Lecture 2

- Recall that the shell (bash) has variables.
- These are strings, and are created through assignment:

```
name=Sam
age=65  # Still a string!
address="34 Green Street"
# Quotes are required if you need spaces.
```

- Recall that you access variables with a $ sign:

```
echo $name is $age, and lives at $address.
```

```
echo ${name} is ${age}, and lives at ${address}.
```

This works by substitution, like makefile variables and C preprocessor constants.

- Note: if you access a variable that doesn't exist, you will get an empty string (*not* an error!)

Example Script

A brief taste of scripting:

```
#!/bin/bash

for file in $*; do
    if [ -x $file ]; then
        ./$file &
    fi
done
```

- This script demonstrates the use of for-each loops, if statements, variables and parameters — some of the common components of scripts.
- We'll discuss all of these in more detail. . .

Users

First, some background on UNIX.

- On UNIX, you have a UID (user ID), a numeric code corresponding to your username.
  - At Curtin, your UID and username are probably equal.
  - In general, usernames can be non-numeric.
- Your UID is recorded against every file and directory you create.
- That is, UNIX remembers who "owns" what.
- A file's owner decides who else can access it (roughly).

Root

- Much of a UNIX system is owned by "root" — UID zero (a.k.a. the superuser, operator, administrator, BOFH[1]).
- The root user is a role, not an actual person.
- The root user has unlimited permission to access, modify and delete anything and everything.
- This power is only occasionally necessary.
- Overused, it can be very dangerous; mistakes are easily made, with catastrophic consequences.
- Even administrators have their own personal non-admin accounts, to limit the risk.

[1] See "Bastard Operator From Hell".

Processes

- A running program is called a "process".
- Processes have a PID (process ID).
- Processes also have an owner — a UID (user ID).
- Conceptually, every process runs *as a particular user* — normally the user who starts the process.

Process Listing

- The `ps`, `top`, `kill` and `killall` commands give you control over running processes.
- The "`ps`" command lists processes:

`[user@pc]$ ps`	(Current terminal only)
`[user@pc]$ ps -a`	(All your processess)
`[user@pc]$ ps -e`	(All processes)

- "`top`" provides an interactive interface, where you can see the CPU/memory resources taken by each process. (A text-based task manager.)

Process Killing

- You can send "signals" to processes, often to end them.
- The "`kill`" command does this, if you know the process ID:
  `[user@pc]$ kill 1375` (Use `ps` to find the PID.)
- By default, `kill` sends the "`SIGTERM`" signal.
  - The process can intercept this to facilitate a graceful exit (e.g. closing files).
- You can instead send the "`SIGKILL`" signal, which immediately ends a process (like a "force quit" option):

```
[user@pc]$ kill -KILL 1375
```

- "`killall`" lets you give a command name or username:

```
[user@pc]$ killall firefox
```

```
[user@pc]$ killall -u username
```

Daemons

- Some long-running system processes — called *daemons* — have their very own username and UID (user ID).
- FTP servers often run as a virtual user called "ftp".
- Other servers sometimes run as "nobody".
- This helps protect the rest of the system from security vulnerabilities in such software.
- If the FTP server has a bug, any damage is limited to things the "ftp" user can do.
- The alternative is for this software to run as "root", but only if the administrator *really, really* trusts it!

Groups

- Users can belong to groups.
  - Each user can belong to any number of groups.
  - Each group can have any number of users.
- Normally this is decided by the administrator.
- Groups allow complex file-ownership/access arrangements.
- Each group has its own name and a GID (group ID).

File Permissions

- Each file and directory is owned by both a user and a group.
- Each also has a set of 9 flags, saying who can do what to it.
- The flags are "Read", "Write" and "eXecute". They are repeated for the user, the group, and everyone else.
  - Read permission allows you to view a file or directory.
  - Write permission allows you to modify it.
  - To delete a file, you need write permission *to its containing directory*, not to the file itself!
  - Execute permission allows you to run a program or script, or change into a directory.
- The file's owner decides:
  - Their own RWX permissions.
  - The group's RWX permissions.
  - Everyone else's RWX permissions.

Changing File Permissions

- File permissions can be changed with the `chmod` command:

```
[user@pc]$ chmod 640 file.txt
```

- A 3-digit number gives the new permissions: the 1st digit for the owner, the 2nd for the group, and the 3rd everyone else.
- Each digit is made by adding a combination of:
  - 4 to allow reading,
  - 2 to allow writing, and/or
  - 1 to allow execution.
- Thus, 640 allows:
  - reading and writing by the owner ($6 = 4 + 2$);
  - reading-only by the group (4);
  - no access to anyone else (0).
- These are "octal" (base-8) numbers!

Octal File Permissions

- The complete set of permission combinations are:
  - 0 — no permission.
  - 1 — execute-only.
  - 2 — write-only.
  - 3 — write and execute $(2+1)$.
  - 4 — read-only.
  - 5 — read and execute $(4+1)$.
  - 6 — read and write $(4+2)$.
  - 7 — read, write and execute $(4+2+1)$.
- Thus:
  - 600 — only the owner can read and write.
  - 644 — the owner can read/write; everyone else can read.
  - 664 — the owner and group can read/write; everyone else can read.
  - 755 — the owner can do anything; everyone else can read/execute.
  - 777 — unlimited permission (don't use!)

Timestamps

- All operating systems keep track of the exact time each file was last modified.
- Some UNIX systems also keep track of:
  - The original creation time.
  - The last access time.
- Any two files can be compared to see which is "newer".
- `make` does this for its targets and pre-requisites (for instance).
- The `touch` command can alter a timestamp; by default, setting it to the current time:

```
[user@pc]$ touch program.c
```

(This would cause `make` to re-compile program.o.)

Symbolic Links (Symlinks)

- A symlink is a special file that behaves a bit like a pointer — it points to another file (or directory).
- You can create a symlink with the `ln` command:

```
[user@pc]$ ln -s originalfile newlink
```

This creates a symlink called "`newlink`" that references the file "originalfile".
- A symlink can be in a completely different directory to the file it references.
- A symlink takes its permissions from the file it references.
- You can access/modify the original file by referring to the symlink.

Hard Links

- A hard link is the connection between a file's data and its directory entry — its name and containing directory.
- Every file has at least one hard link — one name and location.
- In some cases, a file can have more than one.
- A file with two hard links looks like two separate files, which happen to share the same data.
- This has very specific applications (like incremental backups).
- However, symlinks are more flexible:
  - A file with multiple hard links still only has one owner, one group, and one set of permissions.
  - Hard links can only exist within a single filesystem. Symlinks can link to files on a completely separate drive.

Hard Links and File Deletion

- On UNIX, deletion is also called "unlinking".
- You're really just removing a hard link.
- If you remove the *last* hard link, the file is physically deleted.
- . . . except if it's currently open! If you delete a file while a program is trying to read from or write to it, the file's directory entry will vanish, but the file itself will remain — anonymously — until closed.

Devices

- UNIX has other special files that represent hardware devices, or parts of the operating system.
- These are conventionally located in the `/dev` directory.
- Some of these are "block" devices that store a block of data.
  - Hard drives and USB flash drives are block devices.
- Others are "character" devices that accept or generate a sequence of characters.
  - The terminal is a character device.
- Some are virtual devices that provide specialised services (and are useful in scripting).

Common Device Files

Filename	Description
`/dev/sda`	The first hard drive.
`/dev/sda1`	The first partition of the first hard drive.
`/dev/sdb`	The second hard drive (often a USB flash drive).
`/dev/cdrom`	The CD/DVD drive.
`/dev/tty`	The current terminal.
`/dev/null`	A "black hole" — swallows anything you write to it, but is always empty.
`/dev/zero`	Similar, but contains infinite `'\0'` bytes.
`/dev/urandom`	Similar, but contains infinite random bytes.

- This is a common way to discard the output of a command:

```
[user@pc]$ ./program >/dev/null
```


Permissions, Owners and Everything Else

The "`-l`" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Permissions, Owners and Everything Else

The "-l" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Filenames

The filenames — fairly self-explanatory!

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```


File Type

- Here, "`backup`" is a directory ("d").
- The other entries are ordinary files.
- File types include: "-" for ordinary files, "d" for directories, "l" for symbolic links, "b" for block devices, "c" for character devices, and others for various inter-process communication.

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```


Owner's Permissions

The owner has:

- Read ("r") and write ("w") permission for all entries.
- Execute ("x") permission for "`backup`" and "`prog`".
  - The owner can go into the "`backup`" directory.
  - The owner can run the "`prog`" file.

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```


Group's Permissions

The group has:

- Read ("r") permission for all entries.
- Execute ("x") permission for "`backup`" and "`prog`".

Permissions, Owners and Everything Else

The "-l" (for long) switch on ls gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Everyone Else's Permissions

- Nobody else can access "backup" or "prog" in any way.
- Everyone can else read "prog.c" and "prog.h".

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on ls gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Hard Links

- All three files have one hard link, which is normal.
- The backup directory has two. Like all directories, it contains a reference to itself — a special entry called ".". If it contained subdirectories, its link count would be higher still, due to the subdirectories' special ".." entries.

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on ls gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Owner

All files are owned by the username "12345678".

The Shell Script Files Files and Processes Control Data Exit statuses Regular Expressions

Permissions, Owners and Everything Else

The "-l" (for long) switch on ls gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Group

- All files are also owned by the group "student".
- Any users belonging to this group will have the group's level of access to these files.

Permissions, Owners and Everything Else

The "`-l`" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

File Size

The total number of bytes occupied by each file.

Permissions, Owners and Everything Else

The "`-l`" (for long) switch on `ls` gives a lot of information:

```
[user@pc]$ ls -l
```

```
drwxr-x--- 2 12345678 student 4096 Sep 10 10:07 backup
-rwxr-x--- 1 12345678 student 6719 Sep 10 09:59 prog
-rw-r--r-- 1 12345678 student  799 Sep 10 09:51 prog.c
-rw-r--r-- 1 12345678 student   98 Sep 10 09:52 prog.h
```

Timestamp

The time that each entry was last modified.

Back to Scripting

- The preceeding discussion will prepare you for the environment in which you write scripts.
- Scripts often need to deal with file permissions, timestamps and process control.

The "if" Statement

```
if command1; then
    ...
elif command2; then
    ...
elif command3; then
    ...
else
    ...
fi
```

- The "`elif`" (else-if) "`else`" parts are optional.
- The commands after "`if`" and "`elif`" are run to retrieve their exit statuses.
- These exit statuses act as boolean conditions.
- "`fi`" marks the end of an `if` statement.

Conditions

- An `if` statement often looks like this:

```
if [ -e $file ]; then
   ...   Technically, "[" is a command by itself.
fi
```

- The "[" command is commonly used with `if` statements.
- "[" takes a set of parameters that represent a condition to be tested.
  - e.g. `[ -e $file ]` tests whether the `$file` variable refers to an existing file.
- Like all commands, "[" returns an exit status, which the `if` statement checks.

File Tests

- The "[" command can check various file attributes:
  `-e thing` — does "`thing`" exist?
  `-f thing` — is "`thing`" an ordinary file?
  `-d thing` — is "`thing`" a directory?
  `-l thing` — is "`thing`" a symlink?
  `-r thing` — is "`thing`" readable?
  `-w thing` — is "`thing`" writable?
  `-x thing` — is "`thing`" executable?
- Example 1: Check whether "`logfile`" is writable (by you).

```
if [ -w logfile ]; then ...
```

- Example 2: Checks whether the value stored in `$dir` is a directory.

```
if [ -d ${dir} ]; then ...
```


More Tests

- The "[" command can perform a few binary tests as well:
  `file1 -nt file2` — is `file1` newer than `file2`?
  `file1 -ot file2` — is `file1` older than `file2`?
  `string1 = string2` — is `string1` equal to `string2`?
  `string1 != string2` — is `string1` not equal to `string2`?
- And it has separate test for strings containing integers:
  `$x -eq $y` — is $\$x = \$y$ (numerically)?
  `$x -ne $y` — is $\$x \neq \$y$?
  `$x -gt $y` — is $\$x > \$y$?
  `$x -ge $y` — is $\$x \geq \$y$?
  `$x -lt $y` — is $\$x < \$y$?
  `$x -le $y` — is $\$x \leq \$y$?
- When comparing integers, `bash` ignores leading zeroes and spaces.
- When comparing strings, `bash` compares them exactly.

Other Commands as Tests

- "[" is not the only way.
- Most commands report "success" or "failure".
- Example with `grep`:

```
if grep -q fail logfile; then
    echo Badness happened\!
fi
```

  - Runs the command "`grep -q fail logfile`".
  - This reports success if it finds "fail" inside `logfile`.
  - (The `-q` silences `grep`'s terminal output.)
- Example with `ping`:

```
if ping -c 1 ${server}; then
    echo I can access ${server}.
fi
```

Boolean Operators

- The `!`, `&&` and `||` operators ("not", "and" and "or") are used in bash as well.
- They operate on the exit statuses of commands.
- For instance: "`command1 && command2`" reports success if both commands report success.
- This is useful in `if` statements as well:

```
if [ -f $file ] && [ $file -nt $file2 ]; then...
```

If `$file` exists, AND it's newer than `$file2`, then run the commands in the `if` statement.

- Another example:

```
if ! [ -e $file ] || [ $val1 -le 15 ]; then...
```

If `$file` *does not* exist, OR `val1` is less than or equal to 15, then. . .

Short-Circuit Evaluation

- In "`cmd1 && cmd2`", `cmd2` will *only* run if `cmd1` succeeded.
- In "`cmd1 || cmd2`", `cmd2` will *only* run if `cmd1` failed.
- Most programming languages do something similar.
  - (i.e. they skip the remaining expressions if they already know the overall outcome.)
- Some script writers use this as a quick-and-dirty if statement:

```
[ -f $file ] && cat $file
```

"`cat`" will only run if "`[`" succeeds. This is equivalent to:

```
if [ -f $file ]; then
    cat $file
fi
```

(Note: "`cat $file`" displays the file referred to by `$file`.)

The case Statement

More flexible than Java's `switch` statement:

```
case value in
    (pattern1)
        ...
        ;;
    (pattern2)
        ...
        ;;
    (pattern3)
        ...
        ;;
    ...
esac
```

value is tested against each pattern. If a pattern matches, its commands are run (and we skip the other patterns).

Pattern Matching

Each pattern can contain "*", "?", "[]", "~", etc., with the same meaning as on the command-line (as in Chapter 2). For example:

```
case $thing in
    (abc)
        echo Is abc.
        echo Hurrah.
        ;;
    (abc*)  echo Starts with abc. ;;
    (*abc)  echo Ends with abc. ;;
    (*abc*) echo Contains abc. ;;
esac
```

- The value in `$thing` is checked against each pattern in turn.
- If one matches, the corresponding command(s) will be run.
- (Note: these are *not* regexes!)

The while Loop

```
while command; do
    ...
done
```

- Works largely the same way as bash's `if` statement, except of course as a loop.
- For example:

```
read filename
while ! [ -e $filename ]; do
    echo $filename does not exist.
    echo Enter another filename:
    read filename
done
```

- Note: there is no "do-while" loop in bash.

The for (each) Loop

- Unlike C, bash supports the "for-each"[2] loop.
  - It also supports traditional C-style `for` loops, slightly modified.
  - However, these are of limited use in a string-based language.
- For-each loops are simple: you have a list of values, and the loop gives you one at a time.
- On each iteration, the loop picks the next value in the list, and sets a variable to that value.
- The commands in the loop then deal with that variable.

```
for variable in list-of-values; do
    ...
done
```

[2]The word "each" doesn't actually appear in `bash` — that's just a generic term.

Input

- You can acquire user input with the `read` command, which reads a line of text:

```
read variable
```

- You can also read multiple values at once:

```
read variable1 variable2 variable3
```

  - The user is expected to enter at least three words.
  - The 1st will be assigned to **variable1**, the 2nd to **variable2** and the rest of the line to **variable3**.

Command-line parameters

- Script files can have their own command-line parameters.
- These are represented by special variables: $1 for the first parameter, $2 for the second, and so on.
- Say you create a file called `thescript`, containing this:

```
#!/bin/bash
echo $1 is $2 years old and lives at $3.
```

- You could run this as follows:

```
[user@pc]$ ./thescript Barry 102 "14 Grass St."
```

Based on this, $1 would be "Barry", $2 would be "102" and $3 would be "14 Grass St.".
- There are other special variables; e.g.
  - $# is the number of parameters (3, in the above example).
  - $* combines all parameters ("Barry 102 14 Mountain St.").

Arithmetic

- `bash` supports rudimentary maths expressions (integer-only).
- To evaluate an expression, you can enclose it in `$((` and `))`.
- This works by substitution — the whole expression is *replaced* by its result.
- For instance: "`$((5 + 6 * 4))`" is evaluated and replaced by "29".
- You can embed variables in these expressions as well.
- You can use the result (practically) anywhere.

Command Substitution

- You can form a command from the output of other commands.
- Enclose a command in `$(...)`, or `` `...` `` (backticks):
  - Don't confuse the 1st with `$((...))` for arithmetic.
  - Don't confuse the 2nd with single quotes, `'...'`.
- Bash runs the command and captures its output, which then replaces the command.
- For example:

```
thedate="$(date)"
```

  - Run by itself, "`date`" shows the current date and time.
  - Here, this is output is captured and assigned to the variable `thedate`.

Command Substitution — Examples

```
for word in $(cat file.txt); do
    echo $word
done
```

- `$(cat file.txt)` gets the entire contents of file.txt.
- This causes the `for` loop to iterate over each word in the file.
- Thus, each word in file.txt is printed on a separate line.

```
if [ $(date +%A) = Sunday ]; then
    echo "Free ice cream for everyone!"
fi
```

If the current day (as determined by "`date +%A`") is Sunday, output "Free ice cream for everyone!"

Exit Statuses — True or False

- Most decision making in Bash depends on "exit statuses".
- When a program/command ends, it returns a single integer to the operating system — its exit status.
- Zero indicates success. Anything else indicates failure.
  - (This is the opposite way around from boolean values in C!)
- We've been using exit statuses all along:

```
int main(void) {
    ...
    return 0;    /* Exit status == success! */
}
```

- We *could* instead return a variable, indicating either success or failure. Many programs do this.

C and Bash Interacting

myprogram.c

```
int main(void) {
    int status = 1; /* "Failure" by default */
    ...
    if(...) {        /* Error checking */
        ...          /* Do useful things... */
        status = 0; /* "Success"! */
    }
    return status;
}
```

Then in Bash:

```
if ./myprogram; then   # Run the above C program,
    echo "It worked!"  # and check if it succeeded.
fi
```


Regular Expressions, or "Regex"es

- We've already seen Bash's pattern matching.
- Regexes are the same idea, but with different notation.
- Used by various UNIX commands; e.g. `grep`, `sed`, `awk`, `vim`.
- *Wildcards* stand for a single unknown character:
  - "`.`" — any character.
  - "`[abcm-z]`" — any character within the brackets, where "-" gives a range of characters.
- *Quantifiers* may appear after something else, and say how many times it should occur:
  - "?" — zero or one times.
  - "*" — zero or more times.
  - "+" — one or more times.
- Thus:
  - "`.*`" means any sequence of characters.
  - "`abc?`" means "ab", optionally followed by "c".
  - "`[0-9]+`" means any non-empty digit sequence.

grep

- With `grep`, you can find patterns in files, streams and throughout entire directory hierarchies.
  - Use `-E` to allow "extended" regular expressions.
  - `-E` is not always required, but we'll use it for simplicity.
- Example 1: find all integers in file.txt:

```
grep -E '[0-9]+' file.txt
```

- Example 2: find all words ending in "ism" in the output of `somecprogram`:

```
./somecprogram | grep -E '[a-zA-Z]*ism'
```

- Example 3: find all occurrances of `fprintf` and `sprintf` throughout all files in the code directory:

```
grep -E -R '[fs]printf\(' code/
```


More Regex Notation

- Prefix a special character with "\" to take it literally:
  - "`\.\*`" means ".*".
  - "`\?+`" means a sequence of one or more "?"s.
  - "`\\?`" means an optional "\".
- You can group characters with "`(...)`":
  - "`a(bc)?`" means "a" optionally followed by "bc".
  - "`(silly )*me`" means any number of "silly "s, followed by "me"; e.g. "silly silly silly me".
- You can select between alternatives with "`|`":
  - "`abc|def|ghi`" means either "abc" or "def" or "ghi".
  - "`(heads |tails )+`" means one or more of either "heads" or "tails"; e.g. "heads heads tails heads tails".
- You can refer to the beginning/end of the line:
  - "`^abc`" means "abc", but only at the beginning of the line.
  - "`abc$`" means "abc", but only at the end of the line.

vim, less and man

- The `vim` editor and the `less` viewer both support regex searching.
- Type "/", enter a regex, and press Enter.
  - When a match is found, you can find the next one by typing "/" and Enter again.
- The `man` command usually uses `less`, so you can do regex searching while viewing man pages.
  - Try it with the bash man page!