Scripting

Normally prefix bash programs with `#!/bin/bash`.

Simple Commands

- `date` – prints today's date
- `echo` – prints to screen
- `who` – prints who is currently on

You can put commands all on one line:

```
echo -e " `date` \n \t\tWe're all here now!\n `who` \n\n"
```

Here the backticks around date and who cause them to be interpreted as commands.

PATH

The PATH variable holds the directories searched to find commands. Earlier folders are checked first.

Running a Script

1. `bash myscript`
2. `./myscript`
3. Put the script in the path, or in your own folder in the path.

Putting scripts in the path has some risk.

Echo

Echo prints its arguments separated by single spaces, followed by a newline.

If an echoed variable contains characters listed in $IFS, it'll be split along those variables. If you quote it when calling it (`echo "$list"`), it'll prevent that from happening.

`-n`

The `-n` option will prevent a terminating newline, as will putting `\c` at the end of the string. Putting `\c` before the end of the line will also drop any other characters after it.

Inconsistency

Echo is implemented differently from platform to platform, which can cause problems – it's better to use `printf` instead, as it's derived from C. This still differs between Linux and BSD though.

Shell Variables

To store values in a shell variable, write the name of the variable followed by an `=` followed by the value (no spaces).

The shell treats all variables as strings, and adding spaces around the `=` sign causes problems because the spaces are included in the strings.

To refer to the value of a variable, use a `$` before the variable name:

```
echo count      #gives "count"
echo $count     #gives the value stored in count
```

File Name Substitution and Variables

You can define a variable as `*`:

```
x=*
```

Now `ls $x` will produce a directory list. The shell stores the `*` in x, as the shell doesn't perform filename substitution when assigning values to variables.

When you use the command, the shell scans `ls $x` and substitutes `*` for `$x`.

It then rescans the line and replaces the `*` with the list of directories.

Quotes

Backtick

Backticks execute a command and insert standard output at that point.

Single Quotes

Single quotes remove the special meaning of all enclosed characters, including backticks.

Double Quotes

Double quotes remove the special meaning of all enclosed characters, except `$`, backtick, and `\`.

Backquoting

```
echo Your current directory is `pwd`
```

This outputs your current directory by using the pwd command and putting the result inside the echo output.

```
echo "You have `ls | wc -l` files in your directory"
```

Environment

If you create a variable in the shell and then run a script which uses it, it won't pick up the value by default. You have to use `$export x` before running `bash`

`myscript`.

Passing Arguments

Arguments passed are automatically assigned to variables `$0` (initially the name of the function) to `$9`, and you can access the first 10 variables like that.

To access further variables, you need to use the `shift` command, which performs a left shift of the variables, so `$9` now contains what was the 11th variable, and `$0` now contains what was in `$1`.

`$*` is expanded to `$0 $1 $2…`, separated by the Internal Field Separator.

`$@` does the same but is always separated by spaces – it doesn't use the IFS.

If you use `"$@"`, then it's equivalent to `"$0" "$1" "$2"…`

Example Scripts

```
#!/bin/bash
for arg in *
do
    echo $arg
done
exit 0
```

This script echoes the arguments to the script. Note `exit 0` is a successful exit – exit with any other value is unsuccessful, and you can use different value to mark different errors.

`set`

You can use the `set` command in a script to set the command-line arguments. This can be useful for testing programs, when you don't want to keep entering the arguments when you call the script.

```
set and baker charlie
```

This will set `$0` to "and", `$1` to "baker", and `$2` to "charlie".

`set -s` is also useful. This will cause the script to exit on any shell command that gives an error. Any command returning a non-zero value will cause it to exit.

To use `set -e` with loops, you need to use `set +e` before your loop and `set -e` afterwards. (He said this, but the bit above was written on his slides – they seem to contradict each other.)

Set is useful for turning debugging on and off when you need it with `set -v -x` and `set +v +x`.

If you use `sudo` and `set -e` causes the script to exit, you will retain root access when the script exits.

Debugging

You can use the `-x` flag with bash to give you a trace as bash runs the script. `+` shows that a line is a trace, `++` shows that it's one level deeper. One example:

```
filelist="`ls`"
```

gives

```
++ ls
```

in the trace.

User Input

Data is taken from the user using `read` or `line`. `line` will take an entire line fron STDIN and write it to STDOUT.

Testing Conditions

You can test success of shell commands using `while`, `until`, `if`, `&&`, `||`, etc.

`0` represents `true` – the program was successful. Any value besides `0` represents failure.

You can make a logical expression using `test` and similar commands. You can also use square brackets, but must have spaces between the brackets and anything else to be parsed correctly.

You can use `-a` and `-o` for AND and OR operators.

You can use `&&` and `||` as a substitute for `if/then/else/fi`. They are the short-circuit logical operators, so as soon as a series of `&&`s hits a false condition, it knows that the result is false, so it won't execute any more of the commands. As soon as a series of `||`s hits a true condition, it knows that the result is true, so it won't execute any more of the commands.

`test` Command

You can write as `test …`, or as `[ … ]`. The second is cryptic but traditional, so we need to know it. Reminder that there must be spaces between the brackets and any contents.

String Operators

- `-z` to check if a string is zero (empty string) with `test`.
- `-n` to check it a string is non-zero (not the empty string) with `test`.
- `string1 = string2` checks if they're the same (still with `test`).
- `string1 != string2` checks if they're different (still with `test`).
- `\<` and `\>` to check alphabetical ordering (still with `test`).

Note: You can use `$?` to check the exit status of the last command.

Integer Operators

- `-eq` – equal
- `-ne` – not equal
- `-le` – less than or equal
- `-ge` – greater than or equal
- …

File Tests

- `test ! -r "$times"` – returns true if the file `times` is not readable by the user.
- `test ! -f "$times"` – returns true if the file `times` doesn't exist or is not an ordinary file (e.g. is a directory).
- `test -f "$times" -a -r "$times"` – returns true if `times` exists, is an orinary file, and is readable by the user.

Shell Arithmetic

By default, variables are assumed to be strings.

```
number=2
number=$number + 4
```

This code will give number the string value `2 + 4` by concatenating it with the string `+ 4`.

Note that you need to use `=` without spaces around it.

`expr` [check this]

`expr` only works with integers, and you have to use spaces around the operands.

`[[ … ]]`

This is another form of `test` that has some non-standard enhancements (so you can't rely on these existing on any machine).

If the argument to the right of `=` or `!=` is unquoted, it's treated as a pattern, e.g. a regex.

`(( … ))`

This is a non-standard way of representing `expr`. However, the logic operates opposite to the shell – it returns false if the arithmetic expressions evaluates to zero, and true otherwise.

Avoid this if you can, because it's non-standard.

`if/then(/else)/fi`

```
if condition
    then
        do some stuff
    else
        something else
fi
```

`fi` marks the end of the if section.