Typical Shells

- Bourne Shell (bsh)
- C Shell (csh)
- Korn Shell (ksh)
- Bourne Again Shell (bash) – most common shell
- Tcsh Shell (tsch)
- Almquist Shell (ash) – used by BusyBox
- … and many others

All available shells are listed in /etc/shells.

Change default shell using the chsh command. Switch shells temporarily simply by executing the new shell.

Most shells act similarly because there is a POSIX specification defining much of how a shell should behave.

Shell Scripting

You can store a sequence of commands in a regular text file, which you can then invoke as a “program” (shell script).

Shell script can take input from user, file and output them on screen.

Anything you can do in a shell script, you can do on the shell command prompt.

Typically used for system administration tasks and anything needing to be easily modified without going through compilation.

Create a script

Shell scripts stored in plain text file, typically with one command per line.

If you want more than one command per line, separate with semicolon. Semicolons are *optional* on one line commands.

By convention, use the **.sh** file extension for each script. This is *not* required – Linux determines the executable-ness of a script via the execute bit.

Besides executable permission, script must have a read permission.

```
chmod +x script.sh
chmod 755 script.sh
```

Comments

```
# I am a comment!
```

A special comment, placed on the first line of the file, will tell the shell executing the script to use a different shell to continue execution:

```
#!/bin/bash
```

The `#!` is often called a “shebang.”

Variables

To define a variable use following syntax:

```
variable_name=value
no=10
```

Rules for Variable Names

- Variable name must consist of alphanumeric characters or underscore characters (_).
- Don't put spaces on either side of the equal sign when assigning value to variable.
- Variables names are case-sensitive.

Access value of variable

General case:

```
$variablename
```

If there might be ambiguity between the name of the variable and the rest of a string:

```
${variablename}
```

Executing Programs

Run a program as you would normally do on the shell prompt:

```
executable_name arg1 arg2 arg3 etc
```

You can also capture the output of a program and use it like a variable.

Modern syntax: *(can be nested inside another)*

```
var=$(executable_name arg1 arg2 arg3 etc)
```

Legacy syntax: *(cannot be nested)*

```
var=`executable_name arg1 arg2 arg3 etc`
```

Example:

```
echo "The computer hostname is $(hostname)"
```

Exit Codes

All executions have an exit code. Programs that have a success/failure behavior use exit code 0 for success and non-0 for failure.

In C, the exit code is the value returned from `int main`.

To use the exit code, execute the program and then immediately check the special variable `$?`

You can also use the exit code to create compound statements:

```
programname && echo "Program failed to run successfully."
```

In the above example, if `programname` is successful it will have exit code 0, causing short-circuit evaluation to skip the echo.

Shell Arithmetic

Legacy way requires program execution:
`expr` `op1 math-operator op2`

Examples:

```
var=`expr 1 + 3
var=$(expr 2 – 1)
```

Modern BASH supports inline math:

```
var=$((1+3))
var=$((2 - 1))
```

Arguments

Arguments to the shell script are done with positional variables:

`$n`

where n is the number of the argument. Note that the arguments will be the same as argv in a C-program – that is, $0 should be the name of the program being executed.

The read Statement

```
man read
```
⇨ “read a line from standard input”

```
read variable1, variable2,...variableN
```

Example:

```
echo "What is your name?"
read tehname
echo "Your name is ${tehname}!”
```

Shorthand

Shells support a variety of shorthand syntax to simplify common tasks:

`*`	All files
`Z*`	All files starting with letter Z
`*.z`	All files with extension .z
`Xy*.a`	All files beginning with Xy and with an extension of a
`Ab?`	All files starting with Ab and with one more character thereafter.
`{a..g}`	All characters in the range of a through g.

The test Statement

You can use the test statement to see if an expression is true or false.

```
test condition
```

The exit code of test is 0 if the condition is true (success) and 1 if the condition is false (failure to be true).

Example:

```
read value
test 5 == $value || echo "Not equal."
```

The sample program will display “Not Equal” if the numbers do not match.

There is a shorthand for *test* using brackets. You can write the above program as:

```
read value
[ 5 == $value ] || echo "Not equal."
```

You will sometimes see test used with POSIX-compliant argument operators such as -eq for == and -ne for !=, but BASH is provides familiar operators.

if statement

The if statement executes a command (the “condition”) and performs different logic depending on the result.

```
if condition
then
   # if condition is true
   # (exit status of condition is 0)
else
   # if condition is false
   # (exit status of condition is non-0)
fi
```

Often *then* is placed on the same line as the if with a semicolon separating them.

Inline Logic

Often *if* is paired with *test* to do inline logic in a shell script without a purpose-built program:

```
read choice
if [ $choice > 0 ]; then
   echo "number is positive"
else
   echo "number is negative"
fi
```

Loops

There are both for and while loops.

In each and every loop:

- The variable used in loop condition must be initialized, then execution of the loop begins.
- A test (condition) is made at the beginning of each iteration.
- The body of loop ends with a statement that modifies the value of the test (condition) variable.

For Loop

```
for variablename in list
do
   # code here will be repeated
   # for each item in list
done
```

Example:

```
for i in 1 2 3 4 5
do
   echo "Welcome $i times"
done
```

while Loop

while executes a command (the “condition”) until it is false (non-0).

```
while condition
do
   # runs while condition is false
done
```

Sample:

```
i=1
while [ $i <= 10 ]
do
   echo "$n * $i = $(($i * $n))"
   i=$(($i + 1))
done
```

The case Statement

```
case condition in
   pattern1) commands…;
   pattern2) commands…;
   patternN) commands…;
   *) default commands…;
esac
```

Sample:

```
read var
case $var in
   1) echo “One”;
   2) echo “Two”;
   3) echo “Three”;
   *) echo "Greater than four";;
esac
```

Functions

A function is a sequence of statements and commands.

Syntax:

```
function-name()
{
   statements and commands
}
```

Note that there are never any arguments put in the parenthesis.

Calling a function with arguments is done with:

```
function-name arg1 arg2 arg3 … argN
```

Receiving the parameters in the function are done the same way as parameters to the shell script – use positional variables.