The Idea of a Link

When Unix creates a file, it does two things:

1. Set space on a disk to store data in the file.
2. Create a structure called an inode (index node) to hold the basic information about the file. The inode contains all information that Unix needs to make use of the file.

Unix keeps all the inodes in a large table. Within this table, each inode is known by a number called the inumber (index number).

The directory does not really contain the file. All the directory contains is the name of the file and its inumber. Thus, the contents of a directory are actually quite small.

The Contents of an Inode (Index Node)

- the name of the userid that owns the file
- the type of the file (ordinary, directory, special …)
- the size of the file
- where the data is stored
- file permissions
- the last time the file was modifies
- the last time the file was accessed
- the last time the inode was modified
- the number of links to the file

When Unix needs to use the file, it looks up the name in the directory, uses the corresponding inumber to find the inode, and then uses the information in the inode to access the file.

The connection between a file name and its inode is called a link. A link connects a file name with the file itself.

Multiple Links to the Same File

Unix allows multiple links to the same file. In other words, the same file can be known by more than one name.

The unique identifier of a file is its inumber, not its name.

Creating a New Link: ln

To create a new link to an ordinary file, use the ln command with the following syntax:

ln *file newname*

where *file* is the name of an existing ordinary file, and *newname* is the name you want to give the link.

Example:

```
mars% ls -l
total 4
-rw------- 1 sliao sliao 65 Jan 27 22:06 name
mars% ln name newfile
mars% ls -l
total 8
-rw------- 2 sliao sliao 65 Jan 27 22:06 name
-rw------- 2 sliao sliao 65 Jan 27 22:06 newfile
mars%
```

To make new links for one or more ordinary files and place them in a specified directory:

ln *file… directory*

Examples:

```
mars% ls -l
total 8
drwx------ 2 sliao sliao 4096 Jan 27 22:09 backups
-rw------- 1 sliao sliao   65 Jan 27 22:09 file
mars% ls -l backups
total 0
mars% ln file backups
mars% ls -l
total 8
drwx------ 2 sliao sliao 4096 Jan 27 22:10 backups
-rw------- 2 sliao sliao   65 Jan 27 22:09 file
mars% ls -l backups
total 4
-rw------- 2 sliao sliao 65 Jan 27 22:09 file
mars% rm file
mars% ls -l
total 4
drwx------ 2 sliao sliao 4096 Jan 27 22:10 backups
mars% ls -l backups
total 4
-rw------- 1 sliao sliao 65 Jan 27 22:09 file
mars%
```

How the Basic File Commands Work

1. COPY: cp

   Unix creates a new file with its own inumber. You end up with two files.

2. RENAME or MOVE: mv

   Unix changes the file name, but keeps the same inumber. You end up with one file.

3. CREATE A LINK: ln

   Unix makes a new directory entry using the file name you specify. You end up with one file and two names.

4. REMOVE:rm

   Unix deletes the link between the file name and the inumber by removing the directory entry. The file will not be deleted until the last link is removed.

Examples:

```
mars% ls -l
total 4
-rw------- 1 sliao sliao 65 Jan 27 22:09 file
mars% ln file file1
mars% ls -l
total 8
-rw------- 2 sliao sliao 65 Jan 27 22:09 file
-rw------- 2 sliao sliao 65 Jan 27 22:09 file1
mars% ln file file2
mars% ls -l
total 12
-rw------- 3 sliao sliao 65 Jan 27 22:09 file
-rw------- 3 sliao sliao 65 Jan 27 22:09 file1
-rw------- 3 sliao sliao 65 Jan 27 22:09 file2
mars% rm file file1
mars% ls -l
total 4
-rw------- 1 sliao sliao 65 Jan 27 22:09 file2
mars%
```

Symbolic Links: ln -s

There are two limitations about the links we discussed above:

1. You cannot create a link to a directory.
2. You cannot create a link to a file in a different file system.

The solution is to use ln with the -s option. Such a link, which is called symbolic link, does not contain the inumber of the original file. Rather, it contains the pathname of the original file.

Example:

```
% ls -l
total 7
-rw-r--r--  1 s_liao  wheel  1411 Jan 11 10:20
index.html
-rw-------  1 s_liao  wheel   496 Jan 20 18:50 mbox
drwxr-xr-x  2 s_liao  wheel   512 Feb 17 17:06 misc
drwxr-xr-x  7 s_liao  wheel  2560 Feb 15 00:07
public_html
% cd public_html/Courses/2941
% pwd
/usr/home/s_liao/public_html/Courses/2941
% cd
% ln -s public_html/Courses/2941
% ls -l
total 7
lrwxr-xr-x  1 s_liao  wheel    24 Feb 17 17:07 2941 ->
public_html/Courses/2941
-rw-r--r--  1 s_liao  wheel  1411 Jan 11 10:20
index.html
-rw-------  1 s_liao  wheel   496 Jan 20 18:50 mbox
drwxr-xr-x  2 s_liao  wheel   512 Feb 17 17:06 misc
drwxr-xr-x  7 s_liao  wheel  2560 Feb 15 00:07
public_html
% cd 2941
% pwd
/usr/home/s_liao/public_html/Courses/2941
%
```

Variables

Like all other programming languages, a Unix shell allows you to store values into variables.

The name of a shell variable is a sequence of letters, digits, and underscores (_) that begins with a letter or an underscore character followed by zero or more letters, digits, or underscore characters.

Unlike most other programming languages, however, there is only one data type for shell variables, type of string.

When a value is assigned to a shell variable, no matter what the value is, a number or a collection of letters, the shell interprets that value as a string of characters.

Environment Variables

Environment variables are variables that set up your working environment. A Unix shell will set some of these variables, but you can set or reset all of them.

User-Defined Variables

A Unix shell also recognizes variables that are assigned string values by users. In Bourne shell, a value is assigned to a variable as follows:

```
$ variable=my_value
```

Unlike some programming languages like C, where all variables must be declared before they can be used, shell variables do not need to be declared first. You can just simply assign a value to a variable when you want to use it.

Working with Variables

A Unix shell allows users to reference the values of several different variables at the same time:

```
mars% set my_count=1
mars% set my_bin=/home/sliao/bin
mars% set two=2
mars% echo $my_bin $my_count $two
/home/sliao/bin 1 2
mars%
```

The values of variables can be used anywhere on the command line, as the following examples show:

```
mars% set my_bin=/home/sliao/bin
mars% cd $my_bin
mars% pwd
/home/sliao/bin
mars% set num=36
mars% echo There are $num students in my class.
There are 36 students in my class.
mars%
```

Even the name of a command can be stored inside a variable, since a shell performs its substitution before determining the name of the program to be executed and its arguments. For example,

```
mars% set command=sort
mars% cat words
Brief
A
History
Time
Of
mars% $command words
A
Brief
History
Of
Time
mars% set command=wc
mars% set option=-l
mars% set file=words
mars% $command $option $file
5 words
mars%
```

Variables can be assigned to other variables as well:

```
mars% set v1=10
mars% set v2=v1
mars% echo $v2
v1
mars% set v2=$v1
mars% echo $v2
10
mars%
```

Remember that a dollar sign (**$**) must always be placed before the variable name whenever you want to use the value stored in that variable.

When we reference a variable, we can enclose the variable name in braces (**{ }**) to delimit the variable name from any following string. In particular, if the character following the variable name is a letter, a digit, or an underscore, the braces (**{ }**) are required.
Here is an example:

```
mars% set a=backup.
mars% echo Two backup files, ${a}1 and ${a}2
Two backup files, backup.1 and backup.2
mars%
```

The Null Value

A variable can contain a null value. To set a variable with a null value, you just assign no value to the variable

```
% set null_variable=
```

in the C shell, and

```
$ null_variable=
```

in Bourne shell.

Example:

```
mars% echo $NoSuchVar
NoSuchVar: Undefined variable.
mars% set NoSuchVar=
mars% echo $NoSuchVar

mars%
```

Alternatively, we can set null value to a variable by either

```
set null_variable=""
```

or

```
set null_variable=''
```

Filename Substitution

Command arguments are filenames for most of time. When you use a filename as an argument on a command line, a shell will search the specified pattern against all filenames in a directory.

Most characters in such a pattern march themselves, however, you can also use some special pattern-matching characters in your pattern.

Pattern Matches

- **`*`** Any string, including the null string
- **`?`** Any single character
- **`[...]`** Any one of the characters enclosed
- **`[!...]`** Any character ***other than*** those that follow the exclamation mark within brackets

Example:

```
mars% ls -a
.   ..   2941   .cshrc   .history   ll   .logout   mbox
.project
mars% ls .l*
.logout
mars%
```

The following example will bring some interesting results of the filename substitution.

```
mars% ls
2941   ll   mbox
mars% set x='*'
mars% echo $x
2941 ll mbox
mars%
```

How did that happen?

First, the shell assigns the single character **`*`** to variable x when it executes

```
mars% set x='*'
```

The following things will happen when

```
mars% echo $x
```

is executed:

1. The shell scans the command line, substitutes **`*`** as the value of x.
2. The shell then re-scans the line, encounters the **`*`** and then substitutes the names of all files in the current directory.
3. The shell then initiates execution of echo, pass it the filename list as arguments.

This order of evaluation is very important. First, the shell does the variable substitution, then performs the filename substitution.

Quotes

One of the unique features of the shell programming language is the way it interprets quote characters.

Basically, there are four different types of quote characters that a Unix shell recognizes, the single quote character ('), the double quote character ("), the backslash character (\), and the back quote character (`). All quotes must occur in pairs, while the backslash is unary in nature.

The Single Quotes (' ')

The single quotes mainly do two things:

1. Instruct a shell to treat all special characters as the characters themselves, other than the special meanings of them.
2. Instruct a shell to take the enclosed characters, which may contain whitespace, as a group.

Examples:

```
mars% echo *
2941 2947 ll mbox
mars% echo '*'
*

mars% echo '< > | ; ( ) { } >> " ` &'
< > | ; ( ) { } >> " ` &
mars%
```

All meanings of these special characters are ignored by the shell when those characters are enclosed by single quotes.

The second reason to use the single quotes is that we need to take a combination of words as a group at times. Assume that there is a file named phonebook containing some peoples' names and telephone numbers:

```
mars% cat phonebook
Perry Don 222-1234
Perryman Tony 222-4321
Pesce Tony 333-2345
Pestrak Stan 333-5432
Peter Brian 444-9876
Peter Bruno 555-6789
mars%
```

To look up someone in phonebook , we can use the Unix command grep:

```
mars% grep Don phonebook
Perry Don 222-1234
mars%
```

However, if we want to look up Perry, we could get some problems:

```
mars% grep Perry phonebook
Perry Don 222-1234
Perryman Tony 222-4321
mars%
```

It will display two lines that contain Perry. We try to overcome this problem by specifying the first name as well:

```
mars% grep Perry Don phonebook
grep: Don: No such file or directory
phonebook:Perry Don 222-1234
phonebook:Perryman Tony 222-4321
mars%
```

A shell uses one or more whitespace characters to separate the parameters on the command line, the above command line results in grep being passed three parameters: Perry, Don, and phonebook.

When grep is executed, it takes the first parameter as the pattern, and the remaining parameters as the names of the files to search for the pattern. However, when it tries to open the file Don, it fails and the system will issue the error message

```
grep: Don: No such file or directory
```

Then it goes to the next file, phonebook, opens it, searches for the pattern Perry and prints out the two matching lines.

To solve this problem, we can enclose the entire argument inside a pair of single quotes:

```
mars% grep 'Perry Don' phonebook
Perry Don 222-1234
mars%
```

In this case, the shell encounters the first ', and ignores any special characters until it finds the close '. The shell therefore divides the command line into two arguments, Perry Don, including the space character, and phonebook.

No matter how many space characters are enclosed between two single quotes, they will be preserved by the shell.

Examples:

```
mars% echo one      two   three  four
one two three four
mars% echo 'one      two   three  four'
one      two   three  four
mars%
```

When the single quotes are not used, the shell removes all extra whitespace characters from the command line and passes echo four parameters, one, two, three, and four.

In the second case, the space characters are preserved, and the shell treats the entire string of characters enclosed in the single quotes as a single parameter of echo.

The single quotes are also needed when assigning values containing whitespace or special characters to shell variables.

Example:

```
mars% set message='* are in this directory.'
mars% echo $message
2941 ll mbox are in this directory.
mars%
```

The above example shows that the shell still does filename substitution after variable name substitution, meaning that the * was replaced by the names of the all files in the current directory before the echo is executed.

The Backslash

Fundamentally, a backslash, (\), is equivalent to placing single quotes around a single character, with a few minor exceptions. A backslash quotes the single character that immediately follows it. The general format is:

```
\c
```

where c is the character to be quoted. Any special meaning normally attached to that character is removed. For example,

```
mars% set x=*
mars% echo \$x
$x
mars%
```

In this case, the shell ignores the $ that follows the backslash. Therefore, variable substitution is not performed.

The following commands perform the same task, quoting a backslash itself:

```
mars% echo '\'
\
mars% echo \\
\
mars%
```

The Double Quotes (" ")

A pair of double quotes works similarly to that of single quotes, but with one very important exception: they instruct the shell to take all special characters enclosed except:

1. Dollar signs
2. Back quotes

While the single quotes tell a Unix shell to ignore ***all*** of the enclosed characters, the double quotes ask a Unix shell to ignore ***most*** of them.

Here is an example to show the difference between single quotes and double quotes:

```
mars% echo '$SHELL'
$SHELL
mars% echo "$SHELL"
/bin/csh
mars%
```

The above example indicates that the shell interprets the dollar sign (**$**) as if it were not enclosed in double quotes. Another important exception between a pair of double quotes and single quotes is that they treat the back quotes differently.

Command Substitution

To capture the output of any command as an argument to another command, we can place that command line with a pair of back quotes (\`   \`). This is known as command substitution. A Unix shell will first execute the command(s) enclosed within the back quotes, then replace the entire quoted expression with their output.

Example:

```
mars% who
nischal  :0          2023-08-21 15:25 (:0)
nischal  pts/0       2023-08-21 15:27 (:0)
horvath-a pts/1       2023-10-13 21:42
(s0106f8790a336d19.wp.shawcable.net)
sliao    pts/2       2023-10-13 23:58 (wnpgmb0426w-ds02
202-50-88.dynamic.bellmts.net)
ng.tran  :1          2023-09-12 14:25 (:1)
ng.tran  pts/3       2023-09-12 14:27 (:1)
mars% echo `who | wc -l` users are logged on.
6 users are logged on.
mars%
```

The back quotes are often used to change the value stored in a shell variable.

Example:

```
mars% set name="Simon Liao"
mars% echo $name
Simon Liao
mars% set name=`echo $name | tr '[a-z]' '[A-Z]'`
mars% echo $name
SIMON LIAO
mars%
```

The technique of using echo in a pipeline to write data to the standard input of the following command is a very simple yet powerful technique.

We aforementioned that the double quotes would treat the command substitution differently as the single quotes do, here is an example to show the difference:

```
mars% echo " `who | wc -l` users are logged on."
 6 users are logged on.
mars% echo ' `who | wc -l` users are logged on.'
 `who | wc -l` users are logged on.
mars%
```

Arithmetic on Shell Variables

We have mentioned that a Unix shell has only one data type, string of characters. A shell also has no concept of performing arithmetic on values stored inside variables. For example,

```
mars% set i=1
mars% set i=$i+1
mars% echo $i
1+1
mars%
```

The shell only performs a literal substitution of the value of i, which is 1, and tacks on the characters +1.

However, a Unix program called expr can evaluate an expression on the command line:

```
mars% expr 1 + 1
2
mars%
```

Note that each operator and operand given to expr must be a separate argument. See the output from the following:

```
mars% expr 1+1
1+1
mars%
```

The usual arithmetic operators are recognized by expr:

- `+` for addition
- `-` for subtraction
- `/` for division
- `*` for multiplication
- `%` for modulus(remainder).

Multiplication, division, and modulus have higher precedence than addition and subtraction.

Example:

```
mars% expr 25 + 50 / 2
50
mars%
```

Since * is a special character, using it directly will cause some problems:

```
mars% expr 20 * 5
expr: syntax error
mars%
```

The reason is that the shell reads * and then substitutes the names of all files in the working directory. The correct way to perform the multiplication is to quote * with a \:

```
mars% expr 20 \* 5
100
mars%
```

Naturally, one or more of the arguments to expr can be the value stored inside a shell variable, since the shell takes care of the substitution first:

```
mars% set i=1
mars% expr $i + 1
2
mars%
```

We can do the same thing above by using the back quotes to assign the output from expr back to a variable:

```
mars% set i=1
mars% set i=`expr $i + 1`
mars% echo $i
2
mars%
```

Note that expr only evaluates integer arithmetic expressions.

Shell Programming

A pipeline can combine several Unix commands or programs together to perform a relatively complex task. However, if we want to solve a problem that contains a few sub-problems, we may have to use more than one pipeline to tackle the problem.

For example, to interchange the second and third columns of myfile, we can use the following commands:

```
cut -c1 myfile > temp1
cut -c2 myfile > temp2
cut -c3 myfile > temp3
cut -c4- myfile > temp4
paste -d"\0" temp1 temp3 temp2 temp4 > myfile
rm temp[1234]
```

If we need to do the same task often, typing the above commands every time is obviously the last thing we would like to do. Fortunately, a Unix shell allows us to solve this problem fair easily.

Shell Script

A Unix shell is an interactive command interpreter and command programming language. It allows users to put a collection of Unix commands in an executable file, which is often referenced as a shell script or shell program, and executes those commands accordingly. For example, we can write a shell script named **swap2col** to interchange the second and third columns of myfile:

```
mars% cat swap2col
cut -c1 myfile > temp1
cut -c2 myfile > temp2
cut -c3 myfile > temp3
cut -c4- myfile > temp4
paste -d"\0" temp1 temp3 temp2 temp4 > myfile
rm temp[1234]
mars% cat myfile
123456789
abcdefghi
```

```
ABCDEFGHI
mars% swap2col
mars% cat myfile
132456789
acbdefghi
ACBDEFGHI
mars%
```

Bourne Shell or C Shell?

Both Bourne shell and C shell families support the shell programming. However, it could be quite tricky for new shell programmers to select the right shell they are writing programs for. Here are some rules for choosing a shell:

1. If a script begins with **#!**, the Unix kernel executes it using whatever command follows the **#!**. Therefore, we can begin Bourne shell scripts with **#!/bin/sh** (**#!/usr/bin/sh**) or C shell scripts with **#!/bin/csh** (**#!/usr/bin/csh**).
2. If a script does not begin with **#!** followed by a shell name, the working shell will attempt to determine if the program is a Bourne shell script or a C shell script by looking at the first character of the program:
    a. If the first character is a comment, **#**, csh will execute the program as a C shell script.
    b. Otherwise, the program will be executed as a Bourne shell script.

Examples:

```
mars% cat home.1
#!/bin/csh
echo ~$1
mars% home.1 sliao
/home/sliao
mars% cat home.2
#!/bin/sh
echo ~$1
mars% home.2 sliao
~sliao
mars% cat home.3
# This is a shell script for C shell
echo ~$1
mars% home.3 sliao
/home/sliao
mars% cat home.4
echo ~$1
mars% home.4 sliao
~sliao
mars%
```

Note: All students can copy the examples from **/home/sliao/2941/PartII**

Working with Parameters

A Unix shell allows users to pass parameters to shell scripts. With processing the parameters passed to them, shell scripts become more powerful and useful.

Positional Parameters

When you run a shell script, the shell will create positional parameters that refer each word on the command line by its position.

The word in position **0** is the program name itself and is called **$0**, the next word is the first parameter and is called **$1**, and so on up to **$9**.

The following program named list.1 gives a long listing of all files in the current working directory.

```
mars% cat list.1
#!/bin/sh
# Program list.1
ls -l
mars% list.1
total 28
-rwxr-xr-x. 1 sliao sliao  33 Jan 15  2022 list.1
-rw-------. 1 sliao sliao  30 Feb 23 11:36 myfile
-rwxr-xr-x. 1 sliao sliao 153 Feb 23 11:33 swap2col
mars%
```

Then, we modify the program list.1 to list two files.

```
mars% cat list.2
#!/bin/sh
# Program list.2
ls -l $0 $1
mars% list.2 myfile
-rwxr-xr-x. 1 sliao sliao 40 Jan 15  2022 list.2
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

When you invoke shell script list.2, its name list.2 will be substituted into the program at the location **$0**, and whatever word follows it will be substituted into the location **$1**.

What will happen if we use more than one parameter to run the same program list.2?

```
mars% list.2 myfile list.1
-rwxr-xr-x. 1 sliao sliao 40 Jan 15  2022 list.2
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

The second parameter will be ignored by shell. We need to modify the program list.2 in order to hold more parameters.

```
mars% cat list.3
#!/bin/sh
# Program list.3
ls -l $1 $2 $3 $4
mars% list.3 myfile list.1
-rwxr-xr-x. 1 sliao sliao 33 Jan 15  2022 list.1
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

When a parameter for a position is not specified, its positional parameter will be assigned with the null value. The following examples will show more details related to multiple parameters.

```
mars% cat list.4
#!/bin/sh
# Program list.4
echo "Parameters: (1) $1 (2) $2 (3) $3 (4) $4"
ls -l $1 $2 $3 $4
mars% list.4 myfile list.1
Parameters: (1) myfile (2) list.1 (3)  (4)
-rwxr-xr-x. 1 sliao sliao 33 Jan 15  2022 list.1
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

In this example, the null value is assigned to the third and fourth positional parameters.

```
mars% list.4 myfile list.1 list.4 a_file
Parameters: (1) myfile (2) list.1 (3) list.4 (4) a_file
ls: cannot access a_file: No such file or directory
-rwxr-xr-x. 1 sliao sliao 33 Jan 15  2022 list.1
-rwxr-xr-x. 1 sliao sliao 92 Jan 15  2022 list.4
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

There is no file named a_file in the working directory. The message from the system shows so.

If we use more parameters than the positional parameters used in the program, all parameters that are not covered by the positional parameters will be ignored.

```
mars% list.4 myfile list.1 list.4 a_file last_file
Parameters: (1) myfile (2) list.1 (3) list.4 (4) a_file
ls: cannot access a_file: No such file or directory
-rwxr-xr-x. 1 sliao sliao 33 Jan 15  2022 list.1
-rwxr-xr-x. 1 sliao sliao 92 Jan 15  2022 list.4
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars%
```

Predefined Special Variables

Several predefined variables have special meanings and can be very helpful in shell programming.

$# Variable

The **$#** variable will be set to the number of positional parameters passed to the shell, not containing the name of the shell script itself. One of the primary usages of this variable is to present the number of parameters on the command line. The list program is modified to show the total number of the passed parameters.

```
mars% cat list.5
#!/bin/sh
# Program list.5
echo "The number of parameters passed is: $#"
echo "Parameters: (1) $1 (2) $2 (3) $3 (4) $4"
ls -l $1 $2 $3 $4
mars% list.5 myfile list.1
The number of parameters passed is: 2
Parameters: (1) myfile (2) list.1 (3)  (4)
-rwxr-xr-x. 1 sliao sliao 33 Jan 15  2022 list.1
-rw-------. 1 sliao sliao 30 Feb 23 11:36 myfile
mars% list.5 myfile list.1 list.5 f1 f2
The number of parameters passed is: 5
Parameters: (1) myfile (2) list.1 (3) list.5 (4) f1
-rw-r--r--. 1 sliao sliao  21 Mar 16  2022 f1
-rwxr-xr-x. 1 sliao sliao  33 Jan 15  2022 list.1
-rwxr-xr-x. 1 sliao sliao 138 Mar  6  2022 list.5
-rw-------. 1 sliao sliao  30 Feb 23 11:36 myfile
mars%
```

The correct number of parameters sent to list.5 is displayed, though the fifth parameter is ignored by the program because list.5 can only cover four parameters.

$* Variable

The special shell variable **$*** refers to all of the parameters passed to a program on the command line. A newer version of the program list is to show the utility of this variable.

```
mars% cat list.6
#!/bin/sh
# Program list.6
echo "The number of parameters passed is: $#"
echo "Parameters: $*"
ls -l $1 $2 $3 $4
mars% list.6 myfile list.1 list.5 f1 f2
The number of parameters passed is: 5
Parameters: myfile list.1 list.5 f1 f2
ls: cannot access f1: No such file or directory
-rwxr-xr-x. 1 sliao sliao  33 Jan 15  2022 list.1
-rwxr-xr-x. 1 sliao sliao 138 Mar  6  2022 list.5
-rw-------. 1 sliao sliao  30 Feb 23 11:36 myfile
mars%
```

Example: Add New Entries to Phonebook

Assuming that we use a file named phonebook to keep people's names and phone numbers.

```
mars% cat phonebook
Pesce Tony      333-2345
Pestrak Stan    333-5432
Peter Brian     444-9876
Peter Bruno     555-6789
mars%
```

We need to add new names to the file at times. For doing so, we write a program named addname that takes two parameters: the name of the person to be added and a telephone number.

```
mars% cat addname.1
#!/bin/sh
# Program addname, version 1
echo "$1	$2" >> phonebook
mars%
```

There is a tab between $1 and $2. The tab character needs to be quoted in order to be understood by the shell. Then, we try to add a new person to the list by using the program addname.1:

```
mars% cat phonebook
Pesce Tony	333-2345
Pestrak Stan	333-5432
Peter Brian	444-9876
Peter Bruno	555-6789
mars% addname.1 'Pester Larry' 555-4567
mars% cat phonebook
Pesce Tony	333-2345
Pestrak Stan	333-5432
Peter Brian	444-9876
Peter Bruno	555-6789
Pester Larry	555-4567
mars%
```

In this example, 'Pester Larry' is added to the file phonebook as the first positional parameter and his telephone number 555-4567 as the second. We may want to sort the names after we add a new entry to the phonebook. Add a sort command to the addname.1 program will make this possible.

```
mars% cat addname.2
#!/bin/sh
# Program addname, version 2
echo "$1	$2" >> phonebook
sort -o phonebook phonebook
mars%
```

Then, we add a new entry to the file phonebook:

```
mars% addname.2 'Petate James' 555-7654
mars% cat phonebook
Pesce Tony      333-2345
Pester Larry    555-4567
Pestrak Stan    333-5432
Petate James    555-7654
Peter Brian     444-9876
Peter Bruno     555-6789
mars%
```

Now, all names in phonebook are sorted. Every time a new name is added to the list, the file phonebook will be resorted.

Remove Entries from Phonebook

To keep the phonebook updated, we need to have a program to remove names from the list as well. We call this program rmname.

```
mars% cat rmname.1
#!/bin/sh
# Program rmname, version 1
grep -v "$1" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
mars%
```

The program rmname will take only one parameter, the name, and will remove the line that contains the specified name and associated telephone number. The -v option for grep is applied here to extract all lines that do not match the parameter and then write them into a file /tmp/phonebook.

After the mv is executed, the old phonebook file will be replaced by the updated file /tmp/phonebook.

Now, we run the program to remove Pestrak Stan from the list.

```
mars% cat phonebook
Pesce Tony       333-2345
Pester Larry     555-4567
Pestrak Stan     333-5432
Petate James     555-7654
Peter Brian      444-9876
Peter Bruno      555-6789
mars% rmname.1 'Pestrak Stan'
mars% cat phonebook
Pesce Tony       333-2345
Pester Larry     555-4567
Petate James     555-7654
Peter Brian      444-9876
Peter Bruno      555-6789
mars%
```

The line containing Pestrak Stan and his telephone number is removed from phonebook.

There is a potential problem for using this program. /tmp is an area all users can write their files to. However, if one user has a file in /tmp, other users cannot write their files to that directory under the same name. Therefore, there cannot be two users to run this program at the same time. We can solve this problem by using another special variable, **`$$`**, in the program rmname.

$$ Variable

The predefined special shell variable **$$** is set to the process number of the current process. Since the process numbers are unique among all existing processes, **$$** can be used to generate unique names for temporary files. The following is the modified program rmname:

```
mars% cat rmname.2
#!/bin/sh
# Program rmname, version 2
grep -v "$1" phonebook > /tmp/phonebook$$
mv /tmp/phonebook$$ phonebook
mars%
```

Different users who use this program at the same will get different names for their temporary files.

The shift Command

With the positional parameters, we can only refer up to 9 parameters. This is because the shell only accepts a single digit following the dollar sign. However, you can get extra parameters by using the **shift** command. If you execute the command

```
shift
```

then the shell will reassign the values of the positional parameters by discarding the current value of **$1** and reassigning the value of **$2** to **$1**, of **$3** to **$2**, and so on.

When **shift** is executed, the value of variable **$#** is also automatically decremented by one. Here is a program, testshift, to show how **shift** will reassign the values of the positional parameters:

```
mars% cat testshift.1
#!/usr/bin/sh
# Program testshift.1
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
mars%
```

To run the program with 3 parameters,

```
mars% testshift.1 a b c
3 a b c
2 b c
1 c
0
mars%
```

You can shift more than one position at once by using the format

```
shift n
```

For example,

```
shift 2
```

has the same effect as executing two **shift**s:

```
shift
shift
```

consecutively.

With the shell command **shift**, we can deal with more than 9 parameters on the command line. For example, if you have a program that needs to access the tenth positional parameter, you can first save the value of **$1** and then execute a **shift** and access the value of tenth parameter from **$9**:

```
par1=$1
shift
par10=$9
```

After executing the **shift** command, **$1** will now refer to the old **$2**, **$2** to **$3**, and so on.

Exit Status

When a program completes execution under the Unix system, it returns an exit status back to the system. The exit status is a number that indicates whether the program ran successfully or not.

An exit value of zero means that a program succeeded, while a nonzero value indicates that a program failed.

There are many reasons to cause a program's failure. For example, invalid options passed to the program (e.g., an option the program does not support), performing tasks which are not allowed by the system (e.g., using vi to create a file under a directory without writing permission), or wrong number of parameters.

The exit status of a pipeline will be the exit status of the last command in the pipeline.

The Exit Status Variable

There is a shell variable that will be set to the exit status of the last command executed.

$status

The C shell variable **status** is automatically set by the shell to exit status of the last command executed. As other shell variables, its value can be displayed by using echo.

Examples:

```
mars% ls -l File*
-rw-------. 1 sliao sliao 29 Jan 10 20:41 File1
-rw-------. 1 sliao sliao 21 Jan 10 20:41 File2
mars% cp File1 File3
mars% echo $status
0
mars% cp File4 File5
cp: cannot stat File4? No such file or directory
mars% echo $status
1
mars%
```

```
mars% w | grep sliao
sliao    pts/0    wnpgmb0412w-ds01 16:28    0.00s  0.07s  0.01s w
mars% echo $status
0
mars% w | grep NoSuchUser
mars% echo $status
1
mars%
```

$?

In Bourne shell, the shell variable to hold the exit status value is **$?**.

```
$ w | grep sliao
sliao    pts/0    wnpgmb0412w-ds01 16:28    0.00s  0.09s  0.01s w
$ echo $?
0
$ w | grep NoSuchUser
$ echo $?
1
$
```

Since there is no NoSuchUser in the output of w when the pipeline

```
w | grep NoSuchUser
```

is executed, the exit status value of the pipeline would be the exit status of the last command, **grep NoSuchUser**.

The test Command

A command called test can test or evaluate a condition composed of variables and operators. Its general format is

```
test expression
```

where **expression** represents the condition you are testing. The **test** command evaluates **expression**, and if the result is true, **test** returns an exit status of zero; otherwise it returns a nonzero exit value.

The **test** also returns a nonzero exit value if there is no argument:

```
mars% test
mars% echo $status
1
mars%
```

String Operators

There are some string operators that can be used in the **test** statement to perform string comparison.

For example, by using **test**, the following command will return a zero exit value:

```
mars% set name=sliao
mars% test "$name" = sliao
mars% echo $status
0
mars%
```

The **=** operator tests if the two operands are identical. In the above case, we tested whether the contents of the shell variable name is identical to the string sliao. If the two operands are identical, the **test** returns an exit value of zero; otherwise, it would return a nonzero.

The **!=** operator tests two strings for inequality. The exit status from **test** is zero if the two strings are not equal, and nonzero if the two string are identical.

```
mars% set name = sliao
mars% test "$name" != sliao
mars% echo $status
1
mars% test "$name" != xliao
mars% echo $status
0
mars%
```

Space characters can be part of a string.

Example:

```
mars% set name="sliao "
mars% test "$name" = sliao
mars% echo $status
1
mars% test "$name" != sliao
mars% echo $status
0
mars%
```

The double quotes quoting $name are necessary here to contain the space character as part of the string assigned to name.

```
mars% set name="sliao "
mars% test $name = sliao
mars% echo $status
0
mars%
```

In this example, the last character of the string "sliao ", a space character, is lost because $name is not enclosed by double quotes.

A string itself can also be used as an operator to test if the string has a null value. The **test** command returns an exit value zero if the string is not null, and returns a nonzero if the string has a null value.

Example:

```
mars% set name=sliao
mars% test "$name"
mars% echo $status
0
mars% set name=
mars% test "$name"
mars% echo $status
1
mars%
```

Another operator is **-n**, with the general format

-n string

The **-n** operator returns an exit value of zero if the length of **string** is nonzero.

The **-z** operator is the complement of **-n**. The expression

-z string

returns an exit status of zero if the length of **string** is zero.

```
mars% set name=sliao
mars% test -n "$name"
mars% echo $status
0
mars% test -z "$name"
mars% echo $status
1
mars% set name=
mars% test -n "$name"
mars% echo $status
1
mars% test -z "$name"
mars% echo $status
0
mars%
```

Operator	Returns **true** if
string1 = string2	**string1** and **string2** are identical
string1 != string2	**string1** and **string2** are not identical
string	**string** is not the null string
-n string	The length of **string** is nonzero
-z string	The length of **string** is zero

Integer Operators

The **test** command can be used to test numerical values as well. The format of **test** remains the same:

```
test expression
```

For example, the operator **-eq** tests if two integers are equal, and the operator **-lt** tests if the first integer is less than the second one.

Examples:

```
mars% set count=0
mars% test "$count" -eq 0
mars% echo $status
0
mars% set count=1
mars% test "$count" -eq 0
mars% echo $status
1
mars% test "$count" -lt 5
mars% echo $status
0
mars%
```

Some of the integer operators are listed here:

Operator	Returns **true** if
int1 -eq int2	**int1** is equal to **int2**
int1 -ge int2	**int1** is greater than or equal to **int2**
int1 -gt int2	**int1** is greater than **int2**
int1 -le int2	**int1** is less than or equal to **int2**
int1 -lt int2	**int1** is less than **int2**
int1 -ne int2	**int1** is not equal to **int2**

The shell itself makes no distinction about the type of value stored in a variable. It is the **test** operator that interprets the value as an integer when an integer operator is applied.

Examples:

```
mars% set x1="005"
mars% set x2="  10"
mars% test "$x1" = 5
mars% echo $status
1
mars% test "$x1" -eq 5
mars% echo $status
0
mars% test "$x2" = 10
mars% echo $status
1
mars% test "$x2" -eq 10
mars% echo $status
0
mars%
```

It is clear that the string operator **=** and the integer operator **-eq** are doing different things.

File Operators

Most shell programs have to deal with one or more files. Naturally, Unix allows **test** to test on the existence and properties of files.

Some file operators are list in here:

Operator	Returns true if
-r file	file exists and has read permission
-w file	file exists and has write permission
-x file	file exists and has execute permission
-f file	file exists and is a regular file
-d file	file exists and is a directory
-s file	file exists and has a size greater than 0

All of the above file operators are unary operators and expect a single argument to follow.

Examples:

```
mars% ls ~
2941        bin         mail        mbox
mars% test -f ~/mbox
mars% echo $status
0
mars% test -d ~/mbox
mars% echo $status
1
mars% test -d ~/bin
mars% echo $status
0
mars% test -x ~/mbox
mars% echo $status
1
mars% test -r ~/mbox
mars% echo $status
0
mars%
```

The Logical Operators: **-a** and **-o**

The operator **-a** performs a logical AND operation of two expressions and will return a true (an exit value of zero) only if the two joined expressions are both true.

Examples:

```
mars% set num=3
mars% test "$num" -ge 0 -a "$num" -lt 10
mars% echo $status
0
mars% set num=15
mars% test "$num" -ge 0 -a "$num" -lt 10
mars% echo $status
1
mars% test -f ~/mbox -a -r ~/mbox
mars% echo $status
0
mars% test -f ~/mbox -a -x ~/mbox
```

```
mars% echo $status
1
mars%
```

The **-o** performs a logical OR operation and works in similar format as the **-a** operator to form a logical OR of two expressions. The evaluation of the two ORed expressions will be true if either the first expression or the second expression is true.

Examples:

```
mars% test -r ~/mbox -o -x ~/mbox
mars% echo $status
0
mars% test -x ~/mbox -o -x ~/obox
mars% echo $status
1
mars% test -x ~/mbox -o -x ~/year2023
mars% echo $status
1
mars%
```

The **-o** operator has lower precedence than the **-a** operator. But the **-a** operator has lower precedence than the integer, string, and file operators.

So, the expression

```
test "$num" -ge 0 -a "$num" -lt 10
```

will be evaluated as

```
test ( $num -ge 0 ) -a ( $num -lt 10 )
```

However, the above **test** command cannot be executed and will get a message from the system:

```
Badly placed (.
```

because the parentheses `(` and `)` have a special meaning to the shell, so the above command will not be interpreted appropriately by the shell.

You still can use parentheses in a **`test`** expression to change the order of evaluation, but you need to quote the parentheses.

Example:

```
mars% set num=5
mars% test \( $num -ge 0 \) -a \( $num -lt 10 \)
mars% echo $status
0
mars%
```

An Alternate Format for `test`

The test command is used so often by shell programmers that an alternate format of the command is recognized. This format improves the readability of the command, especially when used in if commands.

The general format of the test command

```
test expression
```

can be expressed in the alternate format as

```
[ expression ]
```

The `[` initiates execution of the same **`test`** command, and in this format, **`test`** expects to see a closing `]` at the end of the **`expression`**. It is important to remember that when you use this form, you must surround the brackets, `[` and `]`, with spaces.

Control Statements

A shell is a command programming language. As other programming languages, it provides the control structures that allow making decisions.

The `if-then` Statement

The general format of the `if-then` statement is:

```
if test-command
then
     command
     command
     ...
fi
```

The system first executes `test-command` following the `if` keyword.

If `test-command` returns a zero exit value, then all commands between `then` and `fi` are executed.

If `test-command` is unsuccessful, then all commands between `then` and `fi` will be skipped.

The following is a program named classday.1 to remind you every Wednesday that you have class at 6:00 pm.

```
mars% date
Wed Mar  1 15:36:20 CST 2023
mars% cat classday.1
#!/bin/sh
# Program classday, version 1
if date | grep "Wed" > /dev/null
then
  echo "It's Wednesday, you have class at 6:00 pm."
fi
mars% classday.1
It's Wednesday, you have class at 6:00 pm.
mars%
```

If you do not want to display the output from pipeline

```
if date | grep "Wed"
```

you can redirect it to the device /dev/null. Since the redirection task will always be successful, the exit status of the pipeline only depends on if there is a string Wed in the output of the command date.

The if-then-else Statement

The introduction of the else statement allows the if-then structure to make a two-way decision. The general format of the if-then-else control structure is:

```
if test-command
then
  command
  command
  ...
else
  command
  command
  ...
fi
```

If **test-command** that follows if returns an exit status zero, then the if-then-else structure executes all commands between then and else, and the commands between else and fi are skipped.

If the **test-command** returns a nonzero exit value, the commands between then and else will be skipped and the commands between else and fi are executed. In either case, only one set of commands will be executed: the first set if the exit status is zero, or the second set if it is nonzero.

As an example, we want to improve the program classday.1 to display a message if today is not a class day.

```
mars% cat classday.2
#!/bin/sh
# Program classday, version 2
if date | grep "Wed" > /dev/null
then
      echo "It's Wednesday, you have class at 6:00 pm."
else
      echo "You don't have class today."
fi
mars% date
Wed Mar  1 16:43:07 CST 2023
mars% classday.2
It's Wednesday, you have class at 6:00 pm.
mars%
```

The if-then-elif Statement

When your programs become more complex and have more decisions to make, you may need to use nested if-else statements. However, a special elif construct provided by Unix can do this task easily. The general format of the if-then-elif statement is:

```
if test-command1
then
   command
   command
   ...
elif test-command2
then
   command
   command
   ...
...
elif test-commandn
then
   command
   command
   ...
else
   command
   command
   ...
fi
```

test-command*1*, **test-command*2***, ..., **test-command*n*** are executed in turn and their exit status are tested. If one of them returns an exit status of zero, the commands between the then follows and another elif, else, or fi are executed.

If none of the test-commands returns a zero exit value, the commands after the else are executed. The else is optional in this structure.

If you have classes on Tuesday and Thursday, you may want to improve the program classday.2 further by using the if-then-elif structure:

```
mars% cat classday.3
#!/bin/sh
# Program classday, version 3
if date | grep "Tue" > /dev/null
then
     echo "It's Tuesday, you have class at 4:00 pm."
elif date | grep "Thu" > /dev/null
then
     echo "It's Thursday, you have class at 4:00 pm."
else
     echo "You don't have class today."
fi
mars% date
Wed Mar  1 19:40:46 CST 2023
mars% classday.3
You don't have class today.
mars%
```

The exit Command

The exit command causes a shell program, or a shell to exit immediately. The general format of the exit command is

```
exit n
```

where **n** is the exit value returned to the system. If you omit **n**, the exit value will be that of the last command executed. The value of **n** can be from **0** to **255**, inclusive.

When we use a program that requires arguments, we may pass a wrong number of arguments to the program. For example, the program rmname.2 should remove one entry every time. In the newer version rmname.3, we ask the program to inform user if a wrong number of arguments is passed to it.

```
mars% cat rmname.3
#!/bin/sh
# Program rmname, version 3
if [ "$#" -ne 1 ]
then
   echo "Incorrect number of arguments."
   echo "Usage: $0 name"
   exit 1
fi
grep -v "$1" phonebook > /tmp/phonebook$$
mv /tmp/phonebook$$ phonebook
mars% cat phonebook
Pesce Tony      333-2345
Pester Larry    555-4567
Petate James    555-7654
Peter Brian     444-9876
Peter Bruno     555-6789
mars% rmname.3 Peter Brian
Incorrect number of arguments.
Usage: rmname.3 name
mars% rmname.3 'Peter Brian'
mars% cat phonebook
Pesce Tony      333-2345
Pester Larry    555-4567
Petate James    555-7654
Peter Bruno     555-6789
mars%
```

First, the program rmname.3 checks for the correct number of arguments. If the number does not fit the requirement, the program will display a two-line message and is terminated by the exit command. If the right number of arguments is supplied, the program will remove the name and the associated telephone number from the file phonebook.

The case Statement

Another useful structure for making decisions is the case.

The format of the case statement is

```
case test-string in
  pattern_1 ) command
              ...
              command;;
  pattern_2 ) command
              ...
              command;;
              ...
  pattern_n ) command
              command
              ...
              command;;
esac
```

Please note that each block of commands is terminated by double semicolons, `;;`.

The string **`test-string`** is compared with the patterns **`pattern_1`**, **`pattern_2`**, ..., and **`pattern_n`** successively until a match is found. Then, the commands listed between the matching pattern and the double semicolons are executed. Once the double semicolons are reached, the execution of the **`case`** is terminated.

The following program named translator takes the English numbers and translates it to its French equivalent:

```
mars% cat translator.1
#!/bin/sh
# Program translator, version 1
if test "$#" -ne 1
then
   echo "Usage: $0 word"
   exit 1
fi
case "$1" in
   zero)   echo "zero";;
   one)    echo "un";;
   two)    echo "deux";;
   three)  echo "trois";;
   four)   echo "quatre";;
   five)   echo "cinq";;
   six)    echo "six";;
   seven)  echo "sept";;
   eight)  echo "huit";;
   nine)   echo "neuf";;
esac
mars% translator.1
Usage: translator.1 word
mars% translator.1 one
un
mars% translator.1 seven
sept
mars% translator.1 ten
mars%
```

The last example shows what happens when you type in one word which does not match any pattern in the case statement, there is no echo command is executed.

Special Pattern Matching Characters

The shell allows us to use the same special characters for specifying the patterns in a case statement as you can for filename substitution.

The special characters and strings you can use in the case statement:

Pattern	Matches
*	Matches any string of characters
?	Matches any single character
[...]	Matches any one of the characters enclosed

Since the pattern * matches everything, it is frequently used at the very end of the case as a default. If none of the previous patterns in the case is matched, the * will be matched.

Here is another version of the program translator:

```
mars% cat translator.2
#!/bin/sh
# Program translator, version 2
if test "$#" -ne 1
then
   echo "Usage: $0 word"
   exit 1
fi
case "$1" in
 zero)   echo "zero";;
 one)    echo "un";;
 two)    echo "deux";;
 three)  echo "trois";;
 four)   echo "quatre";;
 five)   echo "cinq";;
 six)    echo "six";;
 seven)  echo "sept";;
 eight)  echo "huit";;
 nine)   echo "neuf";;
```

```
 *)      echo "Sorry, I was not trained to translate this
word"
esac
mars% translator.2 five
cinq
mars% translator.2 ten
Sorry, I was not trained to translate this word
mars%
```

&& and ||

The shell has two simple commands, **&&** and **||**, to make a decision based on whether the preceding command succeeds or fails.

&& causes the following command list to be executed if the preceding pipeline returns a zero exit value;

|| executes the following command list if the preceding pipeline returns a nonzero exit value.

For example,

```
command1 && command2
```

will execute **command1** first, and if **command1** returns an exit value of zero, then **command2** will be executed. If **command1** returns an exit value of nonzero, **command2** will be skipped.

We can rewrite the first version of the program classday.1 to:

```
mars% cat classday.1.1
#!/bin/sh
# Program classday, version 1.1
date | grep "Wed" > /dev/null &&
echo "It's Wednesday, you have class at 6:00 pm."
mars% date
Wed Mar  1 19:57:15 CST 2023
mars% classday.1.1
It's Wednesday, you have class at 6:00 pm.
mars%
```

In fact, **&&** works as a shorthand form of the if-then statement.

The **||** command works similarly as **&&** except the second command executes only if the exit value of the first command is nonzero.

For example,

```
command1 || command2
```

will execute **command2** only when **command1** fails, or the exit value of **command1** is nonzero. The following is an example to use the **||** command:

```
mars% cat classday.2.1
#!/bin/sh
# Program classday, version 2.1
date | grep "Wed" > /dev/null &&
echo "It's Wednesday, you have class at 6:00 pm."
date | grep "Wed" > /dev/null ||
echo "You don't have class today."
mars% date
Thu Mar  2 10:19:19 CST 2023
mars% classday.2.1
You don't have class today.
mars%
```

The **||** command works more like an if-else construct.

Loops

Like most of computing languages, a Unix shell allows users to set up program loops. These loops enable us to execute a set of commands repeatedly either for a specified number of times, or until some condition is met. There are three built-in loop commands: for, while, and until.

The for Command

The for command executes a set of commands for a specified number of times. Its basic format is:

```
for var in value_1 value_2 ... value_n
do
  command
  command
  command
  ...
done
```

The commands enclosed between **do** and **done** form the body of the for loop. These commands are to be executed for **n** times, or the number of times that variables are listed after the keyword **in**. Here is a simple example to show the usage of for command:

```
mars% cat for.1
#!/bin/sh
# Program for, version 1
for i in 1 2 3
do
   echo $i
done
mars% for.1
1
2
3
mars%
```

The $* Variable

In the for loop, we can use the aforementioned special shell variable $* to refer all of the parameters passed to a program on the command line.

Example:

```
mars% cat for.2
#!/bin/sh
# Program for, version 2
echo The number of arguments passed is $#
for arg in $*
do
   echo $arg
done
mars% for.2 myfile file1 file2 file3
The number of arguments passed is 4
myfile
file1
file2
file3
mars%
```

The following example shows that if each argument entered is a file, a directory, or neither.

```
mars% cat for.3
#!/bin/sh
# Program for, version 3
for i in $*
do
   if [ -d "$i" ]
   then
       echo "$i is a directory."
   elif [ -f "$i" ]
   then
       echo "$i is a file."
   else
       echo "$i is neither a file nor a directory."
   fi
done
mars%
```

```
mars% for.3 myfile mydir list.1 not_a_file
myfile is a file.
mydir is a directory.
list.1 is a file.
not_a_file is neither a file nor a directory.
mars%
```

If we want to use a white space as part of an argument for a for loop associating with **$***, we may have some problem. For example, if we run the program for.2 with two different sets of arguments:

```
mars% cat for.2
#!/bin/sh
# Program for, version 2
echo The number of arguments passed is $#
for arg in $*
do
   echo $arg
done
mars% for.2 a b c
The number of arguments passed is 3
a
b
c
mars% for.2 'a b' c
The number of arguments passed is 2
a
b
c
mars%
```

Even though 'a b' was passed as a single argument to for.2, the variable **$*** in the for loop was replaced by three arguments other than two. Thus the loop was executed three times. The reason is that a Unix shell will replace the value of **$*** with **$1**, **$2**, ..., up to **$9**, which cannot take an argument containing a white space.

However, by using a new special variable **$@**, we can solve this problem.

The `$@` Variable

When we use the special shell variable `"$@"`, Unix shells will replace `$1` with `"$1"`, `$2` with `"$2"`, and so on.

The double quotes around `$@` are necessary. Without these double quotes, the `$@` variable will behave just like `$*`.

Now we replace the `$*` with `"$@"`:

```
mars% cat for.4
#!/bin/sh
# Program for, version 4
echo The number of arguments passed is $#
for arg in "$@"
do
   echo $arg
done
mars% for.4 a b c
The number of arguments passed is 3
a
b
c
mars% for.4 'a b' c
The number of arguments passed is 2
a b
c
mars%
```

The variable `"$@"` is so commonly used in the for loop that there is a new notation for the for loop.

The for Loop without List

A special notation is recognized by a Unix shell when using for commands. If you write

```
for var
do
   command
   command
   ...
done
```

then the shell will automatically sequence through all of the arguments typed on the command line, just as if you had written

```
for var in "$@"
do
   command
   command
   ...
done
```

Here is another version of program for:

```
mars% cat for.5
#!/bin/sh
# Program for, version 5
echo The number of arguments passed is $#
for arg
do
   echo $arg
done
mars% for.5 a b c
The number of arguments passed is 3
a
b
c
mars% for.5 'a b' c
The number of arguments passed is 2
a b
c
mars%
```

An example to use the for loop

The following program prints all regular files from the current working directory with the default printer.

```
mars% cat printall
#!/bin/sh
# Program printall
for arg in *
do
  if [ -f "$arg" ]
  then
     lpr "$arg"
     echo "$arg sent to printer"
  fi
done
mars%
```

The while Loop

The second looping command is while. The format of this command is

```
while command_to_be_executed
do
   command
   command
   ...
done
```

The **command_to_be_executed** will be executed first and its exit status is tested. If the status is zero, then the commands enclosed between **do** and **done** are executed. Then **command_to_be_executed** is executed and its exit status is tested again. This process continues until **command_to_be_executed** returns a nonzero exit status. At that point, execution of the loop is terminated.

Note that the commands between **do** and **done** might never be executed if **command_to_be_executed** returns a nonzero exit status the first time it is executed.

Here is a simple example which will display command line arguments one per line:

```
mars% cat testwhile
#!/bin/sh
# Program testwhile
while [ "$#" -ne 0 ]
do
   echo "$1"
   shift
done
mars% testwhile a b c
a
b
c
mars% testwhile 'a b' c
a b
c
mars%
```

The next program prints as many copies of a file as requested. A user is asked to enter two arguments: the first argument specifies the file name and the second argument is the number of hard copies required.

```
mars% cat printcopies
#!/bin/sh
# Program printcopies
if [ "$#" -eq 2 ]
then
  i=$2
  while [ "$i" -gt 0 ]
     do
     lpr "$1"
     echo "$1 sent to printer"
     i=`expr $i - 1`
     done
else
  echo "Please enter 2 arguments where"
  echo "Argument 1 is: File name;"
  echo "Argument 2 is: Number of hard copies
required."
fi
mars%
```

The following is another example to use the while loop. The program will display the square of each integer between **1** and **10**.

```
mars% cat square
#!/bin/sh
max=10
counter=1
while [ $counter -le $max ]
do
  square=`expr $counter \* $counter`
  echo "The square of $counter is $square"
  counter=`expr $counter + 1`
done
mars%
```

```
mars% square
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
mars%
```

The until Command

The until command continues execution as long as the command listed after the until command returns a nonzero exit status. When a zero exit status is returned, the loop is terminated. The general format of the until is:

```
until command_to_be_excuted
do
  command
  command
  ...
done
```

Like the while loop, the commands between **do** and **done** might never be executed if **command_to_be_excuted** returns a zero exit status the first time it is executed.

The until command is useful for writing programs that are waiting for a particular event to occur. For example, if you are waiting for sliao to logon, one approach is to write a program to inform you when it happens.

You could execute a program periodically until sliao eventually logs on, or you could write a program to continually check until he does.

Unix has a command called sleep that suspends execution of a program for a specified number of seconds. The general format is

```
sleep n
```

where **n** is the number of seconds that the program will be suspended.

At the end of that interval, the program resumes execution where it left off with the command that immediately follows the sleep.

Here is a simple program that works as a clock:

```
mars% cat clock
while true
do
clear
date '+%H:%M:%S'
sleep 1
done
mars%
```

The following is a program to wait someone to log on.

```
mars% cat waiting.1
#!/bin/sh
# Program waiting, version 1
if [ "$#" -ne 1 ]
then
  echo "Usage: "$0" user"
  exit 1
fi
userid="$1"
until who | grep "^$userid "  > /dev/null
do
  sleep 10
done
echo "$userid has logged on"
mars% who
sliao    pts/0        2023-03-04 09:54 (wnpgmb0412w-
ds01-161-15-243.dynamic.bellmts.net)
mars% waiting.1 sliao
sliao has logged on
mars%
```

After checking that one argument is provided, the program assigns **`$1`** to userid. Then the **`until`** loop is started. This loop will be executed until the exit status returned by grep is zero. As long as the monitored userid is not logged on, the body of the loop, the **`sleep`** command, will be executed. When the **`until`** loop is exited, a message will be displayed at the terminal that shows the waited userid has logged on.

Since program waiting.1 only checks once per 10 seconds for the monitored userid's logging on, it does not use too much the system's resources when it is running.

Using the program this way is not very practical since it ties up your terminal until the monitored userid logs on. A better idea is to run the program in the background so you can use your terminal for other work.

If you type in a command followed by the **&** character, then that command will be sent to the background for execution. This means that the command will no longer tie up your terminal and you can then proceed with other work.

Now, we run the program waiting.1 in the background:

```
waiting.1 sliao &
```

When the program finds the specified userid, it will echo a one line message. However, you might either miss the message or do not want it to be displayed on your screen for now.

An alternative is to have an option for mailing the message to you. If the option is not selected, then the message will be displayed on the screen.

```
mars% cat waiting.2
#!/bin/sh
# Program waiting, version 2
if [ "$1" = -m ]
then
  mailopt=TRUE
  shift
else
  mailopt=FALSE
fi
if [ "$#" -eq 0 -o "$#" -gt 1 ]
then
  echo "Usage: "$0" [-m] user"
  echo "      -m means to be informed by mail"
  exit 1
fi
userid="$1"
until who | grep "^$userid "  > /dev/null
do
  sleep 10
done
if [ "$mailopt" = FALSE ]
then
  echo "$userid has logged on"
else
  echo "$userid has logged on" | Mail $USER
fi
mars%
```

Breaking a Loop

Sometimes you may want to make an exit from a loop immediately, or skip all commands within a loop. The Unix shell commands **break** and **continue** will allow you to do so.

break

If you want to make an immediate exit from a loop, you can use the break command to exit from the loop (not from the program) by calling

```
break
```

When **break** is executed, the control is sent immediately out of the loop. Then execution continues as normal with the command that follows the loop.

The **break** command is often used to exit from infinite loops, usually when an error condition or the end of processing is detected. Here is an example:

```
while true
do
  input=`get_input`
  if [ "$input" = "quit" -o "$input" = "exit" ]
  then
       break
  else
       process_input "$input"
  fi
done
```

The true command serves no purpose but to return an exit status of zero. The while loop will continue to execute the **get_input** and **process_input** programs until input is equal to quit or exit. When that happens, the **break** command will be executed to terminate the while loop.

If there are more than two loops, the **break** command can be used in the form of

```
break n
```

to exit the **n** innermost loops immediately.

continue

The **continue** command will skip all remaining commands in the loop after **continue**.

The following for loop will search all names in name-list to find whether a specified name exits.

```
for name-list
do
  if [ "$name-list" != "name_specified" ]
  then
       echo "name_specified not found"
       continue
  fi

  # Process a program
  ...
done
```

Each value of name-list will be checked. If it is not the name_specified, a message is displayed and further processing is skipped. Execution of the for loop then continues with the next value in the name-list. The above for loop is equivalent to

```
for name-list
do
  if [ "$name-list" != "name_specified" ]
  then
       echo "name_specified not found"
  else
       # Process a program
  ...
  fi
done
```

Like the **`break`**, an option number can follow the **`continue`**, so

```
continue n
```

causes the commands in the innermost **`n`** loops to be skipped.

A Loop on One Line

Sometimes, you might want to type a simple for loop on a single command line. It is allowed to do so under the Bourne shell. All you need to do is to put a semicolon after the last item in the list, and one after each command in the loop. For example,

```
mars% sh
$ for i in *; do echo $i; done
```

will display all file names from the current working directory.

The same rules apply to while and until loops, but there are no semicolons after then and else.

Defining a Function

In the Bourne, you can define a function:

```
function_name () { list; }
```

The function can be referenced by **function_name**. The body of the function is the **list** of commands between **{** and **}**. The **{** must be followed by a space. The **{** and **}** are unnecessary if the body of the function is a command.

Example:

```
mars% cat clock.2
#!/bin/sh
SleepFiveSeconds () { sleep 5; }
while true
do
clear
date '+%H:%M:%S'
SleepFiveSeconds
done
mars%
```

Reading Data: read

General format:

```
read variable . . .
```

When the read command is executed, the shell reads a line from standard input and assigns the first word to the first variable, the second word to the second variable, and so on.

If there are more words on the line than there are variables listed, then the excess words get assigned to the last variable. For example,

```
read a b
```

will read a line from standard input, storing the first word in variable a, and the remainder of the line in variable b.

```
read my_text
```

will read and store an entire line into variable my_text.

Example:

```
mars% cat mycp
#!/bin/sh
if [ "$#" -ne 2 ]
then
        echo "Usage: $0 from to"
        exit 1
fi

from="$1"; to="$2"

if [ -f "$to" ]
then
        echo "Overwrite $to (yes/no)?"
        read answer
        if [ "$answer" != yes ]
        then
                echo "Copy not performed."
                exit 0
        fi
fi

cp $from $to
mars%

mars% ls -l file*
-rw-r--r--. 1 sliao sliao 6 Mar  4 10:09 file1
-rw-r--r--. 1 sliao sliao 6 Mar  4 10:09 file2
mars% mycp file1 file2
Overwrite file2 (yes/no)?
yes
mars% ls -l file*
-rw-r--r--. 1 sliao sliao 6 Mar  4 10:09 file1
-rw-r--r--. 1 sliao sliao 6 Mar  4 10:13 file2
mars%
```

Using read in a Piped while Loop

Using read in a while loop can solve the problem of utilizing standard input line by line at the end of a pipeline.

Example:

```
mars% cat read_in_while
#!/bin/sh
who | while read name
do
 echo $name
done
mars% who
sliao    pts/0        2023-03-04 09:54 (wnpgmb0412w-ds01-
161-15-243.dynamic.bellmts.net)
sliao    pts/1        2023-03-04 10:15 (wnpgmb0412w-ds01-
161-15-243.dynamic.bellmts.net)
mars% read_in_while
sliao pts/0 2023-03-04 09:54 (wnpgmb0412w-ds01-161-15-
243.dynamic.bellmts.net)
sliao pts/1 2023-03-04 10:15 (wnpgmb0412w-ds01-161-15-
243.dynamic.bellmts.net)
mars%
```

true and false

The true command does nothing, successfully.

The false command does nothing, unsuccessfully.

Examples:

```
mars% echo $status
0
mars% false
mars% echo $status
1
mars% true
mars% echo $status
0
mars%
```

The true command is typically used in a while loop that executes forever:

```
while true
do
command
done
```

Example (You should never try!):

```
while true
do
mkdir x
cd x
done
```

Processes and Job Control

A process is a program that is executing. Every time you enter a command that executes a program, Unix creates a new process.

Displaying the Status of Your Processes: ps

Without options, ps (process status) will only display the process ID, terminal identifier, cumulative execution time, and the command name.

The ps command has a large number of options that vary from system to system. By using these options, you can display a great deal of technical information about each process.

Examples:

```
 mars% ps
    PID TTY          TIME CMD
  55722 pts/0    00:00:00 csh
  55776 pts/0    00:00:00 ps
 mars% w | grep sliao
 sliao    pts/0    wnpgmb0412w-ds01 21:03    2.00s  0.04s
 0.01s w
 sliao    pts/1    wnpgmb0412w-ds01 21:03   13.00s  0.01s
 0.01s -csh
 mars% ps -u sliao
    PID TTY          TIME CMD
  55718 ?        00:00:00 sshd
  55722 pts/0    00:00:00 csh
  55755 pts/1    00:00:00 csh
  55779 pts/0    00:00:00 ps
mars%
```

Foreground and Background Processes

When you enter a command to run a program, the shell normally waits for the program to finish before asking you to enter another command. In this case, we say the process is running in the foreground.

However, it is possible to start a program, and then move right along to the next command. This is called that running a program in the background.

To start a program running in the background, all you need to do is to type an **&** character at the end of the command.

Example:

```
mars% ps -u sliao
   PID TTY          TIME CMD
 55718 ?        00:00:00 sshd
 55722 pts/0    00:00:00 csh
 56001 pts/0    00:00:00 ps
mars% circle&
[1] 56002
mars% ps -u sliao
   PID TTY          TIME CMD
 55718 ?        00:00:00 sshd
 55722 pts/0    00:00:00 csh
 56002 pts/0    00:00:00 circle
 56006 pts/0    00:00:00 ps
mars%
[1]    Done                          circle
mars% ps -u sliao
   PID TTY          TIME CMD
 55718 ?        00:00:00 sshd
 55722 pts/0    00:00:00 csh
 56087 pts/0    00:00:00 ps
mars%
```

The state of the process:

PID	The process ID
TTY	The controlling terminal for the process
TIME	The cumulative execution time for the process

Suspending a Process: Job Control

If you are working with a program and you want to pause it temporarily, you can suspend the process by pressing **^Z**. Then, you can enter any command.

When a process is suspended, it waits indefinitely. To restart it, you can use either the **fg** (foreground) or **bg** (background) commands.

This capability – being able to suspend and restart processes – is called Job Control.

Displaying a List of Suspended Jobs: jobs

To keep track of your suspended jobs, you can use the jobs command. The syntax is:

jobs [-l]

-l displays the job number and process ID

Example:

```
mars% jobs
mars% circle
^Z
Suspended
mars% jobs
[1]  + Suspended                    circle
mars% jobs -l
[1]  + 56141 Suspended                    circle
mars%
```

Moving a Suspended Job to the Foreground: fg

When you restart a suspended job, you can use the **fg** command to move the job to the foreground. The syntax is:

fg [%*job* ...]

where ***job*** is the name or number of a suspended job.

If you enter the command by itself, the shell will restart the current job, which is the most recently suspended job.

If you want to move a different program to the foreground, you can specify the one you want.

Specification	Meaning
`%nn`	job number `nn`
`%`	the current job
`%+`	same as `%`
`%-`	the previous job

Examples:

```
mars% jobs
 [1]   + Suspended                              circle
 [2]   - Running                                square
mars% fg
circle
^Z
Suspended
mars% fg %-
square
^Z
Suspended
mars% fg %1
circle
^Z
Suspended
mars% jobs
 [1]   + Suspended                              circle
 [2]   - Suspended                              square
mars% fg %
circle
^Z
Suspended
mars%
```

Moving a Suspended Job to the Background: **`bg`**

To move a suspended job to the background, you can use the **`bg`** command. The syntax is:

`bg [%`*`job`*`]`

where ***`job`*** is the name or number of a suspended job.

Suspending a Background Process: **`stop`**

To suspend a job that is running in background, use the **`stop`** command:

`stop %`*`job`*

where ***job*** is the name or number of the running process.

Examples:

```
mars% jobs
[1]  + Suspended                        circle
[2]  - Suspended                        square
mars% bg %1
[1]    circle &
mars% jobs
[1]    Running                          circle
[2]  + Suspended                        square
mars% stop %1
mars%
[1]  + Suspended (signal)               circle
mars% jobs
[1]  + Suspended (signal)               circle
[2]  - Suspended                        square
mars%
```

Killing a Process: `kill`

When you are using a regular program running in the foreground, you can terminate it by pressing the **`intr`** key (**`^C`** key for most of systems).

How can you terminate a program that is running in the background? You can do this by using the **`kill`** command. The syntax is:

`kill [-`*`signal`*`]` ***`pid`***

where ***`signal`*** is the type of signal you want to send, and ***`pid`*** identifies the process that you want to terminate.

In fact, the **`kill`** command is designed to send a signal to a process (or processes) specified by each ***`pid`***. Each signal has its own name and identification number.

Although most of systems have many different signals, only two of them are of interest to regular users.

Signal number 15 is called TERM (terminate) which has the effect of killing the process (and this is why the command is called kill).

Most of the time, a process will recognize the TERM signal and shut itself down. However, a program could have been designed to ignore this signal (or something may go wrong).

Signal number 9 is called KILL, which is a stronger form of TERM. Unlike TERM, the KILL signal, by definition, cannot be ignored. Using KILL ensures a sure kill.

Examples:

```
mars% ps
   PID TTY          TIME CMD
 56612 pts/1    00:00:00 csh
 56786 pts/1    00:00:00 circle
 56788 pts/1    00:00:00 square
 56802 pts/1    00:00:00 ps
mars% kill 56786
mars%
[4]    Terminated                    circle
mars% ps
   PID TTY          TIME CMD
 56612 pts/1    00:00:00 csh
 56788 pts/1    00:00:00 square
 56803 pts/1    00:00:00 ps
mars% circle
^Z
Suspended
mars% jobs
[5]  - Running                       square
[6]  + Suspended                     circle
mars% kill -9 %6
mars%
[6]    Killed                        circle
mars% ps
   PID TTY          TIME CMD
 56612 pts/1    00:00:00 csh
 56788 pts/1    00:00:00 square
 56816 pts/1    00:00:00 ps
mars%
```