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.