Keyboard Input And Arithmetic

Up to now, our scripts have not been interactive. That is, they did not require any input from the user. In this lesson, we will see how your scripts can ask questions, and get and use responses.

read

To get input from the keyboard, you use the **read** command. The **read** command takes input from the keyboard and assigns it to a variable. Here is an example:

```
#!/bin/bash

echo -n "Enter some text > "
read text
echo "You entered: $text"
```

As you can see, we displayed a prompt on line 3. Note that "-n" given to the **echo** command causes it to keep the cursor on the same line; i.e., it does not output a linefeed at the end of

the prompt.

Next, we invoke the **read** command with "text" as its argument. What this does is wait for the user to type something followed by a carriage return (the Enter key) and then assign whatever was typed to the variable text.

Here is the script in action:

```
[me@linuxbox me]$ read_demo.bash
Enter some text > this is some text
You entered: this is some text
```

If you don't give the **read** command the name of a variable to assign its input, it will use the environment variable REPLY.

The **read** command also takes some command line options. The two most interesting ones are -t and -s. The -t option followed by a number of seconds provides an automatic timeout for the **read** command. This means that the **read** command will give up after the specified number of seconds if no response has been received from the user. This option could be used in the case of a script that must continue (perhaps resorting to a default response) even if the user does not answer the prompts. Here is the -t option in action:

```
#!/bin/bash

echo -n "Hurry up and type something! > "
if read -t 3 response; then
```

```
    echo "Great, you made it in time!"
else
    echo "Sorry, you are too slow!"
fi
```

The `-s` option causes the user's typing not to be displayed. This is useful when you are asking the user to type in a password or other confidential information.

Arithmetic

Since we are working on a computer, it is natural to expect that it can perform some simple arithmetic. The shell provides features for *integer arithmetic*.

What's an integer? That means whole numbers like 1, 2, 458, -2859. It does not mean fractional numbers like 0.5, .333, or 3.1415. If you must deal with fractional numbers, there is a separate program called **bc** which provides an arbitrary precision calculator language. It can be used in shell scripts, but is beyond the scope of this tutorial.

Let's say you want to use the command line as a primitive calculator. You can do it like this:

```
[me@linuxbox me]$ echo $((2+2))
```

As you can see, when you surround an arithmetic expression with the double parentheses, the shell will perform arithmetic expansion.

Notice that whitespace is ignored:

```
[me@linuxbox me]$ echo $((2+2))
4
[me@linuxbox me]$ echo $(( 2+2 ))
4
[me@linuxbox me]$ echo $(( 2 + 2 ))
4
```

The shell can perform a variety of common (and not so common) arithmetic operations. Here is an example:

```
#!/bin/bash

first_num=0
second_num=0

echo -n "Enter the first number --> "
read first_num
echo -n "Enter the second number -> "
read second_num

echo "first number + second number = $((first_num + second_num))"
echo "first number - second number = $((first_num - second_num))"
echo "first number * second number = $((first_num * second_num))"
echo "first number / second number = $((first_num / second_num))"
```

```
echo "first number % second number = $((first_num % second_num))"
echo "first number raised to the"
echo "power of the second number   = $((first_num ** second_num))"
```

Notice how the leading "$" is not needed to reference variables inside the arithmetic expression such as "`first_num + second_num`".

Try this program out and watch how it handles division (remember, this is integer division) and how it handles large numbers. Numbers that get too large *overflow* like the odometer in a car when you exceed the number of miles it was designed to count. It starts over but first it goes through all the negative numbers because of how integers are represented in memory. Division by zero (which is mathematically invalid) does cause an error.

I'm sure that you recognize the first four operations as addition, subtraction, multiplication and division, but that the fifth one may be unfamiliar. The "`%`" symbol represents remainder (also known as *modulo*). This operation performs division but instead of returning a quotient like division, it returns the remainder. While this might not seem very useful, it does, in fact, provide great utility when writing programs. For example, when a remainder operation returns zero, it indicates that the first number is an exact multiple of the second. This can be very handy:

```
#!/bin/bash

number=0

echo -n "Enter a number > "
read number
```

```
echo "Number is $number"
if [ $((number % 2)) -eq 0 ]; then
    echo "Number is even"
else
    echo "Number is odd"
fi
```

Or, in this program that formats an arbitrary number of seconds into hours and minutes:

```
#!/bin/bash

seconds=0

echo -n "Enter number of seconds > "
read seconds

hours=$((seconds / 3600))
seconds=$((seconds % 3600))
minutes=$((seconds / 60))
seconds=$((seconds % 60))

echo "$hours hour(s) $minutes minute(s) $seconds second(s)"
```