34 – Strings and Numbers

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However, many programming problems need to be solved using smaller units of data such as strings and numbers.

In this chapter, we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 7, "Seeing the World as the Shell Sees It"), there is a well-known command line program called `bc`, which performs higher-level math.

Parameter Expansion

Though parameter expansion came up in Chapter 7, we did not cover it in detail because most parameter expansions are used in scripts rather than on the command line. We have already worked with some forms of parameter expansion, for example, shell variables. The shell provides many more.

> **Note**: It's always good practice to enclose parameter expansions in double quotes to prevent unwanted word splitting, unless there is a specific reason not to. This is especially true when dealing with filenames since they can often include embedded spaces and other assorted nastiness.

Basic Parameters

The simplest form of parameter expansion is reflected in the ordinary use of variables. Here's an example:

```
$a
```

When expanded, this becomes whatever the variable `a` contains. Simple parameters may also be surrounded by braces.

```
${a}
```

This has no effect on the expansion, but is required if the variable is adjacent to other text, which may confuse the shell. In this example, we attempt to create a filename by appending the string `_file` to the contents of the variable `a`.

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

If we perform this sequence of commands, the result will be nothing because the shell will try to expand a variable named `a_file` rather than `a`. This problem can be solved by adding braces around the "real" variable name.

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

We have also seen that positional parameters greater than nine can be accessed by surrounding the number in braces. For example, to access the eleventh positional parameter, we can do this:

`${11}`

Expansions to Manage Empty Variables

Several parameter expansions are intended to deal with nonexistent and empty variables. These expansions are handy for handling missing positional parameters and assigning default values to parameters.

`${`*`parameter`*`:-`*`word`*`}`

If *parameter* is unset (i.e., does not exist) or is empty, this expansion results in the value of *word*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
```

```
bar
```

`${`*`parameter`*`:=`*`word`*`}`

If *parameter* is unset or empty, this expansion results in the value of *word*. In addition, the value of *word* is assigned to *parameter*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Note: Positional and other special parameters cannot be assigned this way.

`${`*`parameter`*`:?`*`word`*`}`

If *parameter* is unset or empty, this expansion causes the script to exit with an error, and the contents of *word* are sent to standard error. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

`${`*`parameter`*`:+`*`word`*`}`

If *parameter* is unset or empty, the expansion results in nothing. If *parameter* is not empty, the value of *word* is substituted for *parameter*; however, the value of *parameter* is not changed.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

Expansions That Return Variable Names

The shell has the ability to return the names of variables. This is used in some rather exotic situations.

`${!`*`prefix`*`*}`
`${!`*`prefix`*`@}`

This expansion returns the names of existing variables with names beginning with *prefix*. According to the `bash` documentation, both forms of the expansion perform identically. Here, we list all the variables in the environment with names that begin with BASH:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL
BASH_VERSINFO BASH_VERSION
```

String Operations

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well suited for operations on pathnames.

`${#`*`parameter`*`}`

expands into the length of the string contained by *parameter*. Normally, *parameter* is a string; however, if *parameter* is either `@` or `*`, then the expansion results in the number of positional parameters.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

`${parameter:offset}`
`${parameter:offset:length}`

These expansions are used to extract a portion of the string contained in *parameter*. The extraction begins at *offset* characters from the beginning of the string and continues until the end of the string, unless *length* is specified.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

If the value of *offset* is negative, it is taken to mean it starts from the end of the string rather than the beginning. Note that negative values must be preceded by a space to prevent confusion with the `${parameter:-word}` expansion. *length*, if present, must not be less than zero.

If *parameter* is `@`, the result of the expansion is *length* positional parameters, starting at *offset*.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

`${parameter#pattern}`
`${parameter##pattern}`

These expansions remove a leading portion of the string contained in *parameter* defined by *pattern*. *pattern* is a wildcard pattern like those used in pathname expansion. The difference in the two forms is that the # form removes the shortest match, while the ## form removes the longest match.

```
[me@linuxbox ~]$ foo=file.txt.zip
```

```
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

${*parameter*%*pattern*}
${*parameter*%%*pattern*}

These expansions are the same as the previous # and ## expansions, except they remove text from the end of the string contained in *parameter* rather than from the beginning.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

${*parameter*/*pattern*/*string*}
${*parameter*//*pattern*/*string*}
${*parameter*/#*pattern*/*string*}
${*parameter*/%*pattern*/*string*}

This expansion performs a search-and-replace operation upon the contents of *parameter*. If text is found matching wildcard *pattern*, it is replaced with the contents of *string*. In the normal form, only the first occurrence of *pattern* is replaced. In the // form, all occurrences are replaced. The /# form requires that the match occur at the beginning of the string, and the /% form requires the match to occur at the end of the string. In every form, /*string* may be omitted, causing the text matched by *pattern* to be deleted.

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as sed and cut. Expansions can improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the `longest-word` program discussed in the previous chapter to use the parameter expansion `${#j}` in place of the command substitution `$(echo -n $j | wc -c)` and its resulting subshell, like so:

```
#!/bin/bash

# longest-word3: find longest string in a file

for i; do
    if [[ -r "$i" ]]; then
         max_word=
         max_len=0
         for j in $(strings $i); do
              len="${#j}"
              if (( len > max_len )); then
                   max_len="$len"
                   max_word="$j"
              fi
         done
         echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

Next, we will compare the efficiency of the two versions by using the `time` command.

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real     0m3.618s
user     0m1.544s
sys 0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real     0m0.060s
user     0m0.056s
sys 0m0.008s
```

The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds—a significant improvement.

Case Conversion

`bash` has four parameter expansions and two `declare` command options to support the uppercase/lowercase conversion of strings.

So what is case conversion good for? Aside from the obvious aesthetic value, it has an important role in programming. Let's consider the case of a database lookup. Imagine that a user has entered a string into a data input field that we want to look up in a database. It's possible the user will enter the value in all uppercase letters or lowercase letters or a combination of both. We certainly don't want to populate our database with every possible permutation of uppercase and lowercase spellings. What to do?

A common approach to this problem is to *normalize* the user's input. That is, convert it into a standardized form before we attempt the database lookup. We can do this by converting all the characters in the user's input to either lower or uppercase and ensure that the database entries are normalized the same way.

The `declare` command can be used to normalize strings to either uppercase or lowercase. Using `declare`, we can force a variable to always contain the desired format no matter what is assigned to it.

```
#!/bin/bash

# ul-declare: demonstrate case conversion via declare

declare -u upper
declare -l lower

if [[ $1 ]]; then
        upper="$1"
        lower="$1"
        echo "$upper"
        echo "$lower"
fi
```

In the preceding script, we use `declare` to create two variables, `upper` and `lower`. We assign the value of the first command line argument (positional parameter 1) to each of the variables and then display them on the screen.

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

As we can see, the command line argument (`aBc`) has been normalized.

In addition to `declare`, there are four parameter expansions that perform upper/lower-case conversion as described in Table 34-1.

Table 34-1: Case Conversion Parameter Expansions

Format	Result
`${`*`parameter,,pattern`*`}`	Expand the value of *parameter* into all lowercase. *`pattern`* is an optional shell pattern (for example, `[A-F]`) that will limit which characters are converted. See the `bash` man page for a full description of patterns.
`${`*`parameter,pattern`*`}`	Expand the value of *parameter*, changing only the first character to lowercase.
`${`*`parameter^^pattern`*`}`	Expand the value of *parameter* into all uppercase letters.
`${`*`parameter^pattern`*`}`	Expand the value of *parameter*, changing only the first character to uppercase (capitalization).

Here is a script that demonstrates these expansions:

```
#!/bin/bash

# ul-param: demonstrate case conversion via parameter expansion

if [[ "$1" ]]; then
        echo "${1,,}"
        echo "${1,}"
        echo "${1^^}"
        echo "${1^}"
fi
```

Here is the script in action:

```
[me@linuxbox ~]$ ul-param aBc
abc
aBc
ABC
ABc
```

Again, we process the first command line argument and output the four variations supported by the parameter expansions. While this script uses the first positional parameter, *parameter* may be any string, variable, or string expression.

Arithmetic Evaluation and Expansion

We looked at arithmetic expansion in Chapter 7. It is used to perform various arithmetic operations on integers. Its basic form is as follows:

`$((`*`expression`*`))`

where *expression* is a valid arithmetic expression.

This is related to the compound command `(( ))` used for arithmetic evaluation (truth tests) we encountered in Chapter 27.

In previous chapters, we saw some of the common types of expressions and operators. Here, we will look at a more complete list.

Number Bases

In Chapter 9, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base. Table 34-2 lists the notations used to specify bases.

Table 34-2: Specifying Different Number Bases

Notation	Description
number	By default, numbers without any notation are treated as decimal (base 10) integers.
0*number*	In arithmetic expressions, numbers with a leading zero are considered octal.
0x*number*	Hexadecimal notation.
base#number	*number* is in *base*

Here are some examples:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

In the previous examples, we print the value of the hexadecimal number `ff` (the largest two-digit number) and the largest eight-digit binary (base 2) number.

Unary Operators

There are two unary operators, `+` and `-`, which are used to indicate whether a number is positive or negative, respectively. An example is `-5`.

Simple Arithmetic

The ordinary arithmetic operators are listed in Table 34-3.

Table 34-3: Arithmetic Operators

Operator	**Description**
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Integer division
`**`	Exponentiation
`%`	Modulo (remainder)

Most of these are self-explanatory, but integer division and modulo require further discussion.

Since the shell's arithmetic operates only on integers, the results of division are always whole numbers.

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

This makes the determination of a remainder in a division operation more important.

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

By using the division and modulo operators, we can determine that 5 divided by 2 results in 2, with a remainder of 1.

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop's execution. In the following example, we display a line of numbers, highlighting each multiple of 5:

```
#!/bin/bash

# modulo: demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
         printf "<%d> " "$i"
    else
         printf "%d " "$i"
    fi
done
printf "\n"
```

When executed, the results look like this:

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Assignment

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo

```

```
[me@linuxbox ~]$ if (( foo = 5 )); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

In the preceding example, we first assign an empty value to the variable `foo` and verify that it is indeed empty. Next, we perform an `if` with the compound command `(( foo = 5 ))`. This process does two interesting things: it assigns the value of 5 to the variable `foo`, and it evaluates to true because `foo` was assigned a non-zero value.

> **Note:** It is important to remember the exact meaning of `=` in the previous expression. A single `=` performs assignment. `foo = 5` says "make `foo` equal to 5," while `==` evaluates equivalence. `foo == 5` says "does `foo` equal 5?" This is a common feature in many programming languages. In the shell, this can be a little confusing because the `test` command accepts a single `=` for string equivalence. This is yet another reason to use the more modern `[[ ]]` and `(( ))` compound commands in place of `test`.

In addition to the `=` notation, the shell also provides notations that perform some very useful assignments as shown in Table 34-4.

Table 34-4: Assignment Operators

Notation	Description
parameter = value	Simple assignment. Assigns *value* to *parameter*.
parameter += value	Addition. Equivalent to *parameter* = *parameter* + *value*.
parameter -= value	Subtraction. Equivalent to *parameter* = *parameter* – *value*.
*parameter *= value*	Multiplication. Equivalent to *parameter* = *parameter* * *value*.
parameter /= value	Integer division. Equivalent to *parameter* = *parameter* / *value*.
parameter %= value	Modulo. Equivalent to *parameter* = *parameter* % *value*.
parameter++	Variable post-increment. Equivalent to *parameter* = *parameter* + 1 (however, see the following

	discussion).
parameter−−	Variable post-decrement. Equivalent to *parameter* = *parameter* − 1.
++parameter	Variable pre-increment. Equivalent to *parameter* = *parameter* + 1.
--parameter	Variable pre-decrement. Equivalent to *parameter* = *parameter* − 1.

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (++) and decrement (−−) operators, which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated into a number of other programming languages, including bash.

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decremented) before the parameter is returned. If placed after, the operation is performed *after* the parameter is returned. This is rather strange, but it is the intended behavior. Here is a demonstration:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

If we assign the value of one to the variable foo and then increment it with the ++ operator placed after the parameter name, foo is returned with the value of one. However, if we look at the value of the variable a second time, we see the incremented value. If we place the ++ operator in front of the parameter, we get the more expected behavior.

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

For most shell applications, prefixing the operator will be the most useful.

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit.

```
#!/bin/bash

# modulo2: demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
    if (((i % 5) == 0 )); then
         printf "<%d> " "$i"
    else
         printf "%d " "$i"
    fi
done
printf "\n"
```

Bit Operations

One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low-level tasks, often involving setting or reading bit-flags. The bit operators are listed in Table 34-5.

Table 34-5: Bit Operators

Operator	Description
~	Bitwise negation. Negate all the bits in a number.
<<	Left bitwise shift. Shift all the bits in a number to the left.
>>	Right bitwise shift. Shift all the bits in a number to the right.
&	Bitwise AND. Perform an AND operation on all the bits in two numbers.
|	Bitwise OR. Perform an OR operation on all the bits in two numbers.
^	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers.

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.

Here we will demonstrate producing a list of powers of 2, using the left bitwise shift operator:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```

Logic

As we discovered in Chapter 27, the `(( ))` compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Table 34-6 provides the complete list.

Table 34-6: Comparison Operators

Operator	Description
`<=`	Less than or equal to.
`>=`	Greater than or equal to.
`<`	Less than.
`>`	Greater than.
`==`	Equal to.
`!=`	Not equal to.
`&&`	Logical AND.
`||`	Logical OR.
`expr1``?`*`expr2`*`:`*`expr3`*	Comparison (ternary) operator. If expression *expr1* evaluates to be non-zero (arithmetic true), then *expr2;* else *expr3*.

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The `(( ))` compound command maps the results into the shell's normal exit codes.

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

The strangest of the logical operators is the *ternary operator*. This operator (which is modeled after the one in the C programming language) performs a stand-alone logical test. It can be used as a kind of `if/then/else` statement. It acts on three arithmetic expressions (strings won't work), and if the first expression is true (or non-zero), the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable `a` switches from zero to one or vice versa.

Please note that performing assignment within the expressions is not straightforward. When attempted, `bash` will declare an error.

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error
token is "-=1")
```

This problem can be mitigated by surrounding the assignment expression with parentheses.

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Next is a more complete example of using arithmetic operators in a script that produces a simple table of numbers.

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" "$a" "$b" "$c"
    ((a<10?++a:(finished=1)))
done
```

In this script, we implement an until loop based on the value of the `finished` variable. Initially, the variable is set to zero (arithmetic false), and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable `a`. At the end of the loop, the value of the counter variable is evaluated. If it is less than 10 (the maximum number of iterations), it is incremented by one, or else the variable `finished` is given the value of one, making `finished` arithmetically true, thereby terminating the loop. Running the script gives this result:

```
[me@linuxbox ~]$ arith-loop
a   a**2 a**3
=   ==== ====
0   0    0
1   1    1
2   4    8
3   9    27
4   16   64
5   25   125
6   36   216
7   49   343
8   64   512
9   81   729
10  100  1000
```

bc – An Arbitrary Precision Calculator Language

We have seen how the shell can handle many types of integer arithmetic, but what if we need to perform higher math or even just use floating-point numbers? The answer is, we can't. At least not directly with the shell. To do this, we need to use an external program. There are several approaches we can take. Embedding Perl or AWK programs is one possible solution, but unfortunately, it's outside the scope of this book.

Another approach is to use a specialized calculator program. One such program found on many Linux systems is called `bc`.

The `bc` program reads a file written in its own C-like language and executes it. A `bc` script may be a separate file, or it may be read from standard input. The `bc` language supports quite a few features including variables, loops, and programmer-defined functions. We won't cover `bc` entirely here, just enough to get a taste. `bc` is well documented by its man page.

Let's start with a simple example. We'll write a `bc` script to add 2 plus 2.

```
/* A very simple bc script */

2 + 2
```

The first line of the script is a comment. `bc` uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with `/*` and end with `*/`.

Using bc

If we save the previous `bc` script as `foo.bc`, we can run it this way:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the `-q` (quiet) option.

bc can also be used interactively.

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

When using bc interactively, we simply type the calculations we want to perform, and the results are immediately displayed. The bc command quit ends the interactive session.

It is also possible to pass a script to bc via standard input.

```
[me@linuxbox ~]$ bc < foo.bc
4
```

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

An Example Script

As a real-world example, we will construct a script that performs a common calculation, monthly loan payments. In the script below, we use a here document to pass a script to bc:

```
#!/bin/bash

# loan-calc: script to calculate monthly loan payments

PROGNAME="${0##*/}" # Use parameter expansion to get basename

usage () {
    cat <<- EOF
    Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

```

```
    Where:

    PRINCIPAL is the amount of the loan.
    INTEREST is the APR as a number (7% = 0.07).
    MONTHS is the length of the loan's term.

    EOF
}

if (($# != 3)); then
    usage
    exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
    scale = 10
    i = $interest / 12
    p = $principal
    n = $months
    a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
    print a, "\n"
EOF
```

When executed, the results look like this:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

This example calculates the monthly payment for a $135,000 loan at 7.75 percent APR for 180 months (15 years). Notice the precision of the answer. This is determined by the value given to the special `scale` variable in the `bc` script. A full description of the `bc` scripting language is provided by the `bc` man page. While its mathematical notation is slightly different from that of the shell (`bc` more closely resembles C), most of it will be quite familiar, based on what we have learned so far.

Summing Up

In this chapter, we learned about many of the little things that can be used to get the "real work" done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our `loan-calc` script demonstrates that even simple scripts can be created to do some really useful things.

Extra Credit

While the basic functionality of the `loan-calc` script is in place, the script is far from complete. For extra credit, try improving the `loan-calc` script with the following features:

- Full verification of the command line arguments
- A command line option to implement an "interactive" mode that will prompt the user to input the principal, interest rate, and term of the loan
- A better format for the output

Further Reading

- The *Bash Hackers Wiki* has a good discussion of parameter expansion: http://wiki.bash-hackers.org/syntax/pe
- The *Bash Reference Manual* covers this, too: http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion
- The *Wikipedia* has a good article describing bit operations: http://en.wikipedia.org/wiki/Bit_operation
- and an article on ternary operations: http://en.wikipedia.org/wiki/Ternary_operation
- as well as a description of the formula for calculating loan payments used in our `loan-calc` script: http://en.wikipedia.org/wiki/Amortization_calculator