The **bc** calculator

E. Krishnan

Most Linux distributions include the Unix utility named **bc**. (It stands for **b**ench **c**alculator, as against the older Unix **dc** or **d**esk **c**alculator) It is an arbitrary precision, programmable calculator, run from the command-line. Let's take a quick look at some of its simple uses before moving on to more advanced features.

Basic usage

To start **bc**, type `bc` at the command-line. (In a GUI interface, open a terminal and type `bc` at the prompt). On pressing the Enter key, we get a welcome message and the cursor rests below it:

```
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
```

Now type this:

```
((1+2)*3-4)/5
```

On pressing Enter, we get 1 as the answer, in the next line.

Next try

```
4/5
```

The answer we get is 0 ! What is happening here?

By default, **bc** sets the number of decimal places in a quotient as 0; thus `4/5` gives `0` and `5/4` gives `1`. This is sometimes useful, for example, if we want to split the result of a division in integer quotient and remainder. Thus we type

```
3476/23
```

to get the integer quotient 151 and type

```
3476%23
```

to get the remainder 3, so that we can write

$$3476 = (23 \times 151) + 3$$

To get the result of a division as a decimal, we have to set the scale variable (whose default value is 0). Type

```
scale=2
```

and try `4/5` again. We then get .80 as the answer. (Here's a trick: instead of retyping a previous command, press the up-arrow key; by repeatedly pressing this key, we get all the earlier commands, starting from the latest.)

Now type

```
scale=100
```

and then

```
sqrt(2)
```

This gives

```
1.414213562373095048801688724209698078569671875376948073176679737990 \
7324784621070388503875343276415727
```

which is $\sqrt{2}$ correct to 100 decimal places! (Remember the adjective *arbitrary precision*?) Try setting `scale=1000` and then `sqrt(2)` again. This raises the question: how many digits can we get after the decimal point? The answer is 2147483647, which is $2^{31} - 1$.

Apart from changing scale, we can also change the *base* in which input and output are represented, using the variables ibase and obase. Thus to convert the decimal number 123 to binary, we type the commands below one by one, pressing Enter after each:

```
bc
obase=2
123
```

and we get

```
1111011
```

We can change back to decimal output with `obase=10`. In much the same way, to convert from binary to decimal, we type:

```
bc
ibase=2
101
```

which gives

```
5
```

In this case, `ibase=10` does not change back to the decimal as the following sequence of commands will show:

```
ibase=10
10+1
```

This gives 3 and not 11, as expected. The trouble is that since *all* inputs are now read as binary, the 10 in `obase=10` is treated as a binary number, that is 2 again! The way to change back is

```
obase=1010
```

since 1010 is the binary representation of 10. (Or more generally, we can do `ibase=A`, where `A` is treated as the hexadecimal number 10. The hexadecimal, or hex for short, is base 16 number system, in which the alphabets A to F stand for the decimal numbers from 10 to 15)

To quit **bc**, simply type `quit` and press Enter; or press the Ctrl and the d keys together. Also, if we don't want to see the welcome message at start-up, we can run **bc** with the **-q** option, as `bc -q`.

Higher math

There is a standard math library available with **bc**. To use the commands available in it, invoke **bc** with the option **l** (the English letter *ell* and not the number one) as

```
bc -l
```

This sets `scale` to 20 (which we can increase, if need be) and makes the following commands available:

`s(x)`	sine of x radians
`c(x)`	cosine of x radians
`a(x)`	arctangent of x, given in radians
`e(x)`	e^x
`l(x)`	natural logarithm of x

and a few others. Thus to get the value of e to 100 decimal places, type the following sequence of commands (pressing `Enter` after each).

```
bc -ql
scale=100
e(1)
```

How do we get the famous π? Remember that

$$\tan^{-1}(-1) = \tfrac{1}{4}\pi$$

so that

$$\pi = 4\tan^{-1}(1)$$

Thus using the command

```
4*a(1)
```

we get the value of π correct to the number of decimals set by `scale`. For example, with `scale=100`, we get

```
3.1415926535897932384626433832795028841971693993751058209749445923078 \
8164062862089986280348253421170676
```

(Here's is something funny: set `scale=360` and try `4*a(1)`. The last three digits are 360, right? Another relation between π and 360 !)

What if we want to compute $\sin 40°$? Using the conversion,

$$1° = \frac{\pi}{180}\,\mathrm{rad}$$

we can use `bc` to do this job by typing

```
s((4*a(1)/180)*40)
```

If we want to compute the sine of a good number of angles given in degrees, the above method becomes a bit tedious. We next see how such tasks can be automated.

New functions

Apart from the functions listed above, **bc** allows us to define our own functions. Suppose we want a function, which computes $\sin x°$ instead of $\sin(x\,\mathrm{rad})$, as `s(x)` does by default. Let's name this function sd. For any given number x, we want sd(x) to compute the sine of $x°$. We have s(x), which computes the sine of x rad. Using the conversion from radian to degrees, we need only set

$$\mathrm{sd}(x) = \mathrm{s}\left(\frac{\pi}{180}x\right)$$

How do we tell bc this? We type

```
define sd(x){return s((4*a(1)/180)*x)}
```

and press Enter. Then for this entire **bc** session— that is, till we type quit to stop **bc**—we can use sd to compute the sines of angles in degrees.

As another example, suppose we want to compute $\sqrt[3]{2}$. We have seen that **bc** has the sqrt function to calculate square roots, but cube roots are not built into it. Also, the operation x^y, which computes powers, works only for integral values of y. So, we define a new function to compute x^y for any x and y, as in any scientific calculator. (But remember, no scientific calculator can give answers correct to 2147483647 decimal places!)

We first note that for any real numbers x and y with $x > 0$

$$x^y = \mathrm{e}^{y\ln(x)}$$

where $\ln(x)$ is the natural logarithm of x. Since, we have e(x) to compute e^x and l(x) to compute $\ln(x)$ in **bc**, we can use them to define a *power* function $p(x, y)$ to compute x^y as follows:

```
define p(x,y){return(e(y*l(x)))}
```

Then

```
p(2,1/3)
```

gives

```
1.25992104989487316476
```

which is $\sqrt[3]{2}$ correct to 20 decimal places. And p(e(1),4*a(1)) gives e^π (correct to 20 decimal places) as 23.14069263277926900427

Now suppose we want the sine values of $1°, 2°, 3°, 4°, 5°$. We can type sd(1), sd(2),... in succession. But there's an easier way, which we discuss next.

Programming

We mentioned at the beginning that **bc** is a *programmable* calculator. Thus it has various programming constructs such as loops and conditionals. Let's consider the problem of listing sine values mentioned above. We do this using the for loop. Call bc -l and define $\mathrm{sd}(x)$ as before. Then type

```
for(x=1;x<=5;x++)sd(x)
```

This would give the five sine values one below the other. The only piece of the above code that requires an explanation is the x++ part. It means "increment the value of x by 1". (Note also that different parts of the code are separated by semicolons.) Thus the program computes sd(1), then increments 1 to 2 and computes sd(2) and so on till sd(5).

As an example of using a conditional, let's see how we can use **bc** to compute factorials. Note that we can define

$$f(n) = 1 \times 2 \times 3 \times \cdots \times n$$

recursively as

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ nf(n-1), & \text{otherwise} \end{cases}$$

This we state in the language of **bc** as follows:

```
define f(n){if (n==1) return(1); return(n*f(n-1))}
```

Now typing `f(10)` gives $1 \times 2 \times 3 \times \cdots \times 10$ as 3628800.

Note especially the use of double `=` in the definition above. The expression `n==1` checks whether the relation $n = 1$ is true for the number x. (The expression `x=1` simply assigns the value 1 for the variable n.) Also, the alternative `return(n*f(n-1)` is given *without* any preceding "else" or "otherwise".

As another instance of **bc** programming, recall how we defined the $p(x, y)$ function to compute x^y. This definition has a minor flaw; if we give 2 as the base and 3 as the exponent, then we get 2^3 as 7.99999999999999999982 and not 8. One (rather unsatisfactory) solution to this problem is to use the built-in function `x^y` for integral values of `y` and the function `p(x,y)` otherwise. A more elegant method is to modify `p(x,y)` to take care of both situations.

For this, we must first have a method for **bc** to check whether a given number is an integer or not. The idea is to extract the integer part of a number and then to compare it with the original number. To do the first, we recall that if scale is set to 0, then division gives only the integer part. So, we can try

```
define i(x){scale=0;y=x/1;return(y)}
```

and this gives, for example, `i(3.45)` as 3. But this has the undesirable side-effect of setting scale to 0, so that we have to manually reset it to the original value. To overcome this, we resort to a routine trick in programming, storing the value of a variable in another temporary variable. The following **bc** session will make the idea clear. (Here the inputs typed are given in `typewriter` font and the outputs of bc in sans serif, as we have been doing throughout this article.)

```
x=4;x
4
temp=x;x;temp
4
4
x=5;x;temp
5
4
x=temp;x;temp
4
4
```

The crucial point is that in the third input, though `x` is changed to 5, the variable `temp` retains its original value 4. Keeping this in mind, we can modify `i(x)` as follows:

```
define i(x){s=scale;scale=0;y=x/1;scale=s;return(y)}
```

The trouble with this is that the variables `s` and `t` retain their values once `i(x)` is used (as can be verified by typing `s` or `y` after this command is used.) To make them "local", we use the auto prefix. Thus the final code is:

```
define i(x){auto s,y;s=scale;scale=0;y=x/1;scale=s;return(y)}
```

Once this definition has been made, we can modify our `p(x,y)` as below:

```
define p(x,y){if (i(y)==y) return(x^i(y));return(e(y*l(x)))}
```

We next see how certain features of the command-line makes working with **bc** quicker and more flexible.

Working with shells

The Linux command-line is the basic interface between the user and the operating system and is actually a programming environment. It is usually called a *shell*. Several types of shells are available and the one most commonly used is the bash. (It is an abbreviation of *Bourne Again SHell* which in turn is a pun on the *Bourne Shell* developed by Stephen Bourne for Unix systems.)

One basic feature of bash (or any other shell) is that we can combine commands, passing on the result of one to another. For example, the echo command simply echoes its argument onto the terminal. For example, typing

```
echo "Hello"
```

gives back Hello on the terminal. Now try

```
echo "2+3"|bc
```

This gives 5. Here the | symbol (called a *pipe*) is used to pass the output of `echo "2+3"`, which is 2+3, to bc which in turn does the computation and gives 5. Thus

```
echo "define h(x){if(x==1)return(1);return(h(x-1)+1/x)}h(100)"|bc -l
```

would compute the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{100}$ as

5.18737751763962026041

Instead of displaying the result on the terminal, we can write it to a file by typing

```
echo "define h(x){if(x==1)return(1);return(h(x-1)+1/x)}h(100)"|bc -l>sum.txt
```

we get a file sum.txt containing the result of this computation. The > is said to *redirect* output, in this case to a file. If we want to *append* this number to the end of an already existing file (without overwriting its contents), then we should use >> instead of >, as in

```
echo "define h(x){if(x==1)return(1);return(h(x-1)+1/x)}h(100)"|bc -l>>sum.txt
```

We can also get the result on the terminal *and* write it to a file using the **tee** command, which reads from the terminal and writes to both the terminal and a specified file (Recall what a T pipe does in plumbing). Thus

```
echo "define h(x){if(x==1)return(1);return(h(x-1)+1/x)}h(100)"|bc -l|tee sum.txt
```

would display the number above in the terminal and write it to the file as well. Again, if we want the output to be appended to a file, we use **tee** with the **-a** option, as in

```
echo "define h(x){if(x==1)return(1);return(h(x-1)+1/x)}h(100)"|bc-l|tee -a sum.txt
```

Finally, we see how we can customize **bc** include our own favourite operations.

Extensions

The new functions we define during a **bc** session last only till we quit the session. That means, everytime we use **bc**, these must be redefined. We next see how we can have our definitions available every time we start **bc**.

We can put all our reusable definitions in a file and load it on **bc** start-up, much as the switch -l loads the math library. For this, we create a file, say extensions.bc, containing these definitions, with a text editor such as **gedit** or **Emacs** (but *not* a word processor like **OpenOffice Writer**, unless we save it as a *text* file) and call **bc** by

```
bc -l extensions.bc
```

This allows us to use all the definitions in the file extensions.bc. We can even rename this entire command as **ebc** or something. To do this, we first copy the file extensions.bc to the bin directory under our home directory. Then we create a *text* file named ebc containing just the line

```
bc -ql ~/bin/extensions.bc
```

and make this executable with the command

```
chmod a+x ebc
```

After this, if we call ebc, we get **bc** with the math library and our own extensions loaded. For this to work, the bin directory under our home directory must be included among the various directories which **bash** searches for executables. We can check this by typing `echo $PATH`. If the output does not contain $HOME/bin, then we add the lines

```
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
```

to the **.profile** file under our home directory.

In creating extensions, it is a good idea to name the defined functions as `factorial` or `power`, so that more generic names like `f` or `p` can be used for other temporary functions. Thus for example, we may define in our extension file,

```
/* To find the integer part of a number */

define int(x){
   auto oldscale,y
   oldscale=scale
   scale=0
   y=x/1
   scale=oldscale
   return(y)
}

/* To find powers of numbers */

define power(x,y){
  if (int(y)==y)
  return(x^int(y))
  return(e(y*l(x)))
 }
```

Note that the different commands within a definition are not separated by semicolons; this is not necessary if each command is on a line by itself. Also, it is a good practice to give comments on each definitions, enclosed between `*/` and `*/` as above. It would be a great help when you share your extensions with others, in the true spirit of Free Software.