Bash Shell and Beyond

By William Park

Series Introduction

In this monthly series, I will try to expose the power of the Bash shell. In particular, the readers will be introduced to Bash.Diff which is a collection of my patches incorporating many ideas from Ksh, Zsh, Awk, Python, and other languages.

Each article will focus on one theme or feature, which is not normally thought of in shell context. I will also make liberal use of shell functions, standard builtins, dynamically loadable builtins, and advanced features patched into Bash, in a format that can be easily sourced and maintained.

string.sh

In C, <string.h> defines strcat(3), strcpy(3), strlen(3), and strcmp(3) for string concatenation, copy, size, and test operations respectively. Such basic operations are needed constantly when programming in any language, and shell scripting is no exception.

strcat() and strcpy()

For string copy and concatenation, you would do something like

```
a=abc
a=${a}'123'                    # a=abc123
```

in shell. This is simple variable assignment. However, you can't have variable reference on the left-hand side (LHS) of '='. You have to either type the LHS variable name explicitly as above, or use `eval`, as in

```
x=a
eval "$x=abc"
eval "$x=\${$x}'123'"
```

to parse the "..." expressions twice. It quickly becomes painful to call `eval` all the time, especially when the variable names are parsed from a file or a string.

What is needed is a shell version of C functions strcat(3) and strcpy(3) which can be called with equal ease and simplicity. So, here they are:

```
strcat ()             # var+=string
{
    local _VAR=$1 _STRING=$2 _a _b

    case $#.$3 in
        2.) ;;
        3.*:*) _a=${3%:*} _b=${3#*:}
            set -- `python_to_shell_range "$_a" "$_b" ${#_STRING}`
            _STRING=${_STRING:$1:$2}
            ;;
        *)  echo "Usage: strcat var string [a:b]"
            return 2
            ;;
    esac
    eval "$_VAR=\${$_VAR}\$_STRING"
}

strcpy ()             # var=string
{
```

```
        local _VAR=$1 _STRING=$2 _a _b

        case $#.$3 in
            2.) ;;
            3.*:*) _a=${3%:*} _b=${3#*:}
                 set -- `python_to_shell_range "$_a" "$_b" ${#_STRING}`
                 _STRING=${_STRING:$1:$2}
                 ;;
            *)  echo "Usage: strcpy var string [a:b]"
                 return 2
                 ;;
        esac
        eval "$_VAR=\$_STRING"
    }
```

where 'var' is the name of variable you want to use to store the result. The above example, then, becomes

```
    x=a
    strcpy $x abc                 # a=abc
    strcat $x 123                 # a+=123
```

strlen()

In C, strlen(3) gives you the size of a string. In the shell, you would use parameter expansion (i.e., `${#var}`):

```
    a=abc123
    echo ${#a}                    # 6
```

Here is a shell version of C function strlen(3):

```
    strlen ()             # echo ${#string} ...
    {
        for i in "$@"; do
            echo ${#i}
        done
    }
```

which has the additional ability of accepting multiple strings, e.g.

```
    strlen abc123 0123456789              # 6 10
```

strcmp()

To compare two strings for equality, you use strcmp(3) in C. In shell, you would do something like

```
    [ $a = abc123 ]
```

Here is a shell version of C function strcmp(3):

```
    strcmp ()             # string == string
    {
        local _STRING1=$1 _STRING2=$2 _a _b

        case $#.$3 in
            2.) ;;
            3.*:*) _a=${3%:*} _b=${3#*:}
                 set -- `python_to_shell_range "$_a" "$_b" ${#_STRING1}`
                 _STRING1=${_STRING1:$1:$2}
                 set -- `python_to_shell_range "$_a" "$_b" ${#_STRING2}`
                 _STRING2=${_STRING2:$1:$2}
                 ;;
            *)  echo "Usage: strcmp string1 string2 [a:b]"
                 return 2
                 ;;
        esac
        [ "$_STRING1" == "$_STRING2" ]
```

```
}
```
so that the above example becomes
```
strcmp $a abc123
```

Python-style [a:b] substring

Extracting substrings is another common operation. In the shell, you would use parameter expansion (i.e., `${var:a:n}`), where 'a' is starting index and 'n' is the number of characters to extract. So,

```
b=0123456789
echo ${b:0:3} ${b: -3} ${b:1:${#b}-2}
```
will print the first 3 chars, the last 3 chars, and all chars except the first and the last, respectively.

The main problem with this syntax is that 'n' is a relative number starting at index 'a'. Usually, absolute index is more convenient, not only because it's more natural, but also because that's the way it is in C. Python has syntax var[a:b], where 'a' and 'b' are indexes which can be positive, negative, or omitted. Although it's roughly equivalent to ${var:a:b-a} in shell, missing 'a' and 'b' means the beginning and the end of string, and negative index means offset from the end of string.

The above shell functions strcat(), strcpy(), and strcmp() already support Python-style [a:b] format, provided you source an internal function.

```
# string[a:b] --> ${string:a:n}
#
# Convert Python-style string[a:b] range into Shell-style ${string:a:n} range,
# where
#    0 <= a <= b <= size  and  a + n = b
#
python_to_shell_range ()
{
    local -i size=$3
    local -i b=${2:-$size}
    local -i a=${1:-0}

    if [ $# -ne 3 ]; then
        echo "Usage: python_to_shell_range a b size"
        return 2
    fi

    [[ a -lt 0 ]] && a=$((a+size))
    [[ a -lt 0 ]] && a=0
    [[ a -gt size ]] && a=$size

    [[ b -lt 0 ]] && b=$((b+size))
    [[ b -lt 0 ]] && b=0
    [[ b -gt size ]] && b=$size
    [[ b -lt a ]] && b=$a

    echo $a $((b-a))
}
```
to convert Python range to shell range. It's not user-serviceable, but you can try it out:
```
python_to_shell_range '' 3 10          # 0 3
python_to_shell_range -3 '' 10                 # 7 3
python_to_shell_range 1 -1 10          # 1 8
```
Now, you can specify a substring for strcat(), strcpy(), and strcmp() using Python-style [a:b] range as the third parameter, like this:
```
b=0123456789
strcpy x $b :3                # x=012
strcpy y $b -3:               # y=789
```

```
strcpy z $b 1:-1              # z=12345678
echo $x $y $z
```

Chaining of tests

strcmp() tests two strings for equality. When there is a chain of 2 or more binary tests, like 'a < c > b' or '1 -lt 3 -gt 2', you have to break it up and test each pair:

```
[[ a < c ]] && [[ c > b ]]
[ 1 -lt 3 ] && [ 3 -gt 2 ]
```

This breaks up the flow of your code, not to mention being error-prone. Here is a shell function which enables you to simply write down the chains on command-line:

```
testchain ()          # string OP string OP string ...
{
     if [ $# -lt 3 ]; then
         echo "Usage: testchain string OP string [OP string ...]"
         return 2
     fi
     while [ $# -ge 3 ]; do
         test "$1" "$2" "$3" || return 1
         shift 2
     done
}
```

where 'OP' is any binary operator accepted by `test` command. You use it much like `test` command:

```
testchain a '<' c '>' b
testchain 1 -lt 3 -gt 2
```

Summary

The source code for the 6 shell functions listed in this article is also available from string.sh. To use it, just source it,

```
. string.sh
```

In the next article, we'll see how strcat(), strcpy(), strlen(), and strcmp() shell functions can be written in C and compiled as builtin commands. And *that* will be the first introduction to my patched Bash shell. :-)