
Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de Botequim
Parte VII

De pouco adianta ter acesso à informação se ela não puder ser apresentada de forma atraente e que facilite a compreensão. O comando *tput* pode ser usado por shell scripts para posicionar caracteres e criar todos os tipos de efeito com o texto mostrado na tela. Garçom, solta uma geladinha!

por Julio Cezar Neves

Cumequié, rapaz! Derreteu os pensamentos para fazer o scriptzinho que eu te pedi?

– É, eu realmente tive de colocar muita pensação na tela preta, mas acho que finalmente consegui! Bem, pelo menos nos testes que fiz a coisa funcionou, mas você tem sempre que botar chifres em cabeça de cachorro!

– Não é bem assim. É que programar em Shell Script é muito fácil, mas o que é realmente importante são as dicas e macetes que não são triviais. As correções que faço são justamente para mostrá-los. Mas vamos pedir dois chopes enquanto dou uma olhadela no teu script lá na **listagem 1**. Aê Chico, traz dois chopes! E não se esqueça que um deles é sem colarinho!

– Peraí, deixa eu ver se entendi o que você fez: você coloca na variável `Dir` a última linha do arquivo a ser restaurado, em nosso caso `/tmp/$LOGNAME/$1` (onde `$LOGNAME` é o nome do usuário logado, e `$1` é o primeiro parâmetro que você passou ao script), já que foi lá que armazenamos o nome e caminho originais do arquivo antes de movê-lo para o diretório (definido na variável `Dir`). O comando `grep -v` apaga essa linha, restaurando o arquivo ao estado original, e o manda de volta pra onde ele veio. A última linha o apaga da "lixeira". Sensacional! Impecável! Nenhum erro! Viu? Você já está pegando as manhas do shell!

– Então vamos lá, chega de lesco-lesco e blá-blá-blá, sobre o quê nós vamos falar hoje?

– É, tô vendo que o bichinho do shell te pegou. Vamos ver como ler dados, mas antes vou te mostrar um comando que te dá todas as ferramentas para formatar uma tela de entrada de dados.

Listagem 1 – restaura.sh

```
01 #!/bin/bash
02 #
03 # Restaura arquivos deletados via erreeme
04 #
05
06 if [ $# -eq 0 ]
07 then
08     echo "Uso: $0 <Nome do arquivo a ser restaurado>"
09     exit 1
10 fi
11 # Pega nome do arquivo/diretório original na última linha
12 Dir='tail -1 /tmp/$LOGNAME/$1'
13 # O grep -v exclui a última linha e recria o arquivo com o diretório
14 # e nome originalmente usados
15 grep -v $Dir /tmp/$LOGNAME/$1 > $Dir/$1
16 # Remove o arquivo que já estava moribundo
17 rm /tmp/$LOGNAME/$1
```

O comando tput

O principal uso desse comando é o posicionamento do cursor na tela. Alguns parâmetros podem não funcionar se o modelo de terminal definido pela variável de ambiente `$TERM` não suportá-los. A **tabela 1** apresenta apenas os principais parâmetros e os efeitos resultantes, mas existem muito mais deles. Para saber tudo sobre o *tput*, veja a referência [1].

Vamos fazer um programa bem besta e fácil para ilustrar melhor o uso desse comando. É uma versão do famigerado "Alô Mundo", só que dessa vez a frase será escrita no centro da tela e em vídeo reverso. Depois disso, o cursor voltará para a posição original. Veja a **listagem 2**.

Como o programa já está todo comentado, acho que a única linha que precisa de explicação é a 8, onde criamos a variável `Coluna`. O estranho ali é aquele número 9, que na verdade indica o tamanho da cadeia de caracteres que vou escrever na tela. Dessa forma, este programa somente conseguiria centralizar cadeias de 9 caracteres, mas veja isto:

```
$ var=Papo
$ echo ${#var}
4
$ var="Papo de Botequim"
$ echo ${#var}
16
```

Tabela 1: Parâmetros do tput

Parâmetro	Efeito
`cup lin col`	***CU**rsor **P**osition* – Posiciona o cursor na linha *lin* e coluna *col*. A origem (0,0) fica no canto superior esquerdo da tela.
`bold`	Coloca a tela em modo negrito
`rev`	Coloca a tela em modo de vídeo reverso
`smso`	Idêntico ao anterior
`smul`	Sublinha os caracteres
`blink`	Deixa os caracteres piscando
`sgr0`	Restaura a tela a seu modo normal
`reset`	Limpa o terminal e restaura suas definições de acordo com *terminfo*, ou seja, o terminal volta ao comportamento padrão definido pela variável de ambiente `$TERM`
`lines`	Informa a quantidade de linhas que compõem a tela
`cols`	Informa a quantidade de colunas que compõem a tela
`el`	***E**rase **L**ine* – Apaga a linha a partir da posição do cursor
`ed`	***E**rase **D**isplay* – Apaga a tela a partir da posição do cursor
`il n`	***I**nsert **L**ines* – Insere n linhas a partir da posição do cursor
`dl n`	***D**elete **L**ines* – Remove n linhas a partir da posição do cursor
`dch n`	***D**elete **CH**aracters* – Apaga n caracteres a partir da posição do cursor
`sc`	***S**ave **C**ursor position* – Salva a posição do cursor
`rc`	***R**estore **C**ursor position* – Coloca o cursor na posição marcada pelo último `sc`

Ahhh, melhorou! Então agora sabemos que a construção `${#variavel}` devolve a quantidade de caracteres da variável. Assim sendo, vamos otimizar o nosso programa para que ele escreva em vídeo reverso, no centro da tela (e independente do número de caracteres) a cadeia de caracteres passada como parâmetro e depois retorne o cursor à posição em que estava antes da execução do script. Veja o resultado na **listagem 3**.

Este script é igual ao anterior, só que trocamos o valor fixo na variável Coluna (9) por `${#1}`, onde esse `1` é `$1`, ou seja, essa construção devolve o número de caracteres do primeiro parâmetro passado para o programa. Se o parâmetro tivesse espaços em branco, seria preciso colocá-lo entre aspas, senão o `$1` levaria em conta somente o pedaço antes do primeiro espaço. Para evitar este aborrecimento, é só substituir o `$1` por `$*`, que como sabemos é o conjunto de todos os parâmetros. Então a linha 8 ficaria assim:

```
# Centralizando a mensagem na tela
Coluna=`$(((Colunas - ${#*}) / 2))`
```

e a linha 12 (`echo $1`) passaria a ser:

```
echo $*
```

Lendo dados da tela

Bem, a partir de agora vamos aprender tudo sobre leitura. Só não posso ensinar a ler cartas e búzios porque se soubesse estaria rico, num *pub* Londrino tomando um *scotch* e não em um boteco tomando chope. Mas vamos em frente.

Da última vez em que nos encontramos eu dei uma palhinha sobre o comando `read`. Antes de entrarmos em detalhes, veja só isso:

```
$ read var1 var2 var3
Papo de Botequim
$ echo $var1
Papo
$ echo $var2
de
$ echo $var3
Botequim
$ read var1 var2
Papo de Botequim
$ echo $var1
Papo
$ echo $var2
de Botequim
```

Como você viu, o *read* recebe uma lista de parâmetros separada por espaços em branco e coloca cada item dessa lista em uma variável. Se a quantidade de variáveis for menor que a quantidade de itens, a última variável recebe o restante deles. Eu disse lista separada por espaços em branco, mas agora que você já conhece tudo sobre o `$IFS` (*Inter Field Separator* – Separador entre campos), que

Listagem 2: alo.sh

```
01 #!/bin/bash
02 # Script bobo para testar
03 # o comando tput (versao 1)
04
05 Colunas=`tput cols`     #  Salva a quantidade de colunas na tela
06 Linhas=`tput lines`     #  Salva a quantidade linhas na tela
07 Linha=$((Linhas / 2))   # Qual é a linha central da tela?
08 Coluna=$(((Colunas - 9) / 2)) # Centraliza a mensagem na tela
09 tput sc                # Salva a posição do cursor
10 tput cup $Linha $Coluna #  Posiciona o cursor antes de escrever
11 tput rev               #  Vídeo reverso
12 echo Alô Mundo
13 tput sgr0              #  Restaura o vídeo ao normal
14 tput rc                #  Restaura o cursor à posição original
```

Listagem 3: alo.sh melhorado

```
01 #!/bin/bash
02 # Script bobo para testar
03 # o comando tput (versão 2.0)
04
05 Colunas=`tput cols`     #  Salva a quantidade de colunas na tela
06 Linhas=`tput lines`     #  Salva a quantidade de linhas na tela
07 Linha=$((Linhas / 2))   # Qual é a linha central da tela?
08 Coluna=$(((Colunas - ${#1}) / 2)) # Centraliza a mensagem na tela
09 tput sc                # Salva a posicao do cursor
10 tput cup $Linha $Coluna #  Posiciona o cursor antes de escrever
11 tput rev               #  Vídeo reverso
12 echo $1
13 tput sgr0              #  Restaura o vídeo ao normal
14 tput rc                #  Devolve o cursor à posição original
```

eu te apresentei quando falávamos do comando `for`, será que ainda acredita nisso? Vamos testar:

```
$ oIFS="$IFS"
$ IFS=:
$ read var1 var2 var3
Papo de Botequim
$ echo $var1
Papo de Botequim
$ echo $var2
$ echo $var3
$ read var1 var2 var3
Papo:de:Botequim
$ echo $var1
Papo
$ echo $var2
de
$ echo $var3
Botequim
$ IFS="$oIFS"
```

Viu? eu estava furado! O `read` lê uma lista, assim como o `for`, separada pelos caracteres da variável `$IFS`. Veja como isso pode facilitar a sua vida:

```
$ grep julio /etc/passwd
julio:x:500:544:Julio C. Neves - 7070:➥
/home/julio:/bin/bash
$ oIFS="$IFS"    #  Salva o IFS antigo.
$ IFS=:
$ grep julio /etc/passwd | read lname ➥
lixo uid gid coment home shell
$ echo -e "$lname\n$uid\n$gid\n$coment➥
\n$home\n$shell"
julio
500
544
Julio C. Neves - 7070
/home/julio
/bin/bash
$ IFS="$oIFS"   #  Restaura o IFS
```

Como você viu, a saída do `grep` foi redirecionada para o comando `read`, que leu todos os campos de uma só tacada. A opção `-e` do `echo` foi usada para que o `\n` fosse entendido como uma quebra de linha (*new line*) e não como um literal. Sob o Bash existem diversas opções do `read` que servem para facilitar a sua vida. Veja a **tabela 2**.

E agora direto aos exemplos curtos para demonstrar estas opções. Para ler um campo "Matrícula":

```
# -n não salta linha
$ echo -n "Matricula: "; read Mat
Matricula: 12345
$ echo $Mat
12345
```

Podemos simplificar as coisas usando a opção `-p`:

```
$ read -p "Matricula: " Mat
Matricula: 12345
$ echo $Mat
12345
```

E podemos ler apenas uma quantidade pré-determinada de caracteres:

```
$ read -n5 -p"CEP: " Num ; read -n3 ➥
-p- Compl
CEP: 12345-678$
$ echo $Num
12345
$ echo $Compl
678
```

No exemplo acima executamos duas vezes o comando `read`: um para a primeira parte do CEP e outra para o seu complemento, deste modo formatando a entrada de dados. O cifrão (`$`) logo após o último algarismo digitado é necessário porque o `read` não inclui por padrão um caractere *new line* implícito, como o echo.

Para ler só durante um determinado limite de tempo (também conhecido como *time out*):

```
$ read -t2 -p "Digite seu nome completo: ➥
" Nom || echo 'Eita moleza!'
Digite seu nome completo: Eita moleza!
$ echo $Nom
```

O exemplo acima foi uma brincadeira, pois eu só tinha 2 segundos para digitar o meu nome completo e mal tive tempo de teclar um *J* (aquele colado no *Eita*), mas ele serviu para mostrar duas coisas:

- **1)** O comando após o par de barras verticais (o *ou* – *or* – lógico, lembra-se?) será executado caso a digitação não tenha sido concluída no tempo estipulado;
- **2)** A variável `Nom` permaneceu vazia. Ela só receberá um valor quando o ***ENTER*** for teclado.

```
$ read -sp "Senha: "
Senha: $ echo $REPLY
segredo :)
```

Aproveitei um erro no exemplo anterior para mostrar um macete. Quando escrevi a primeira linha, esqueci de colocar o nome da variável que iria receber a senha e só notei isso quando ia escrevê-la. Felizmente a variável `$REPLY` do Bash contém a última seqüência de caracteres digitada – e me aproveitei disso para não perder a viagem. Teste você mesmo o que acabei de fazer.

O exemplo que dei, na verdade, era para mostrar que a opção `-s` impede que o que está sendo digitado seja mostrado na tela. Como no exemplo anterior, a falta do *new line* fez com que o prompt de comando (`$`) permanecesse na mesma linha.

Agora que sabemos ler da tela, vejamos como se lêem os dados dos arquivos.

Lendo arquivos

Como eu já havia lhe dito, e você deve se lembrar, o `while` testa um comando e executa um bloco de instruções enquanto esse comando for bem sucedido. Ora, quando você está lendo um arquivo para o qual você tem permissão de leitura, o `read` só será mal sucedido quando alcançar o `EOF` (*End Of File* – Fim do Arquivo). Portanto, podemos ler um arquivo de duas maneiras. A primeira é redirecionando a entrada do arquivo para o bloco `while`, assim:

```
while read Linha
do
    echo $Linha
done < arquivo
```

Tabela 2: Opções do read

Opção	Ação
`-p prompt`	Escreve "prompt" na tela antes de fazer a leitura
`-n num`	Lê até `num` caracteres
`-t seg`	Espera `seg` segundos para que a leitura seja concluída
`-s`	Não exibe na tela os caracteres digitados.

A segunda é redirecionando a saída de um `cat` para o `while`, da seguinte maneira:

```
cat arquivo |
while read Linha
do
    echo $Linha
done
```

Cada um dos processos tem suas vantagens e desvantagens. O primeiro é mais rápido e não necessita de um subshell para assisti-lo mas, em contrapartida, o redirecionamento fica pouco visível em um bloco de instruções grande, o que por vezes prejudica a visualização do código. O segundo processo traz a vantagem de que, como o nome do arquivo está antes do while, a visualização do código é mais fácil. Entretanto, o Pipe (|) chama um subshell para interpretá-lo, tornando o processo mais lento e pesado. Para ilustrar o que foi dito, veja os exemplos a seguir:

```
$ cat readpipe.sh
#!/bin/bash
# readpipe.sh
# Exemplo de read passando um arquivo
# por um pipe.
Ultimo="(vazio)"
# Passa o script ($0) para o while
cat $0 | while read Linha
do
    Ultimo="$Linha"
    echo "-$Ultimo-"
done
echo "Acabou, Último=:$Ultimo:"
```

Vamos ver o resultado de sua execução:

```
$ readpipe.sh
-#!/bin/bash-
-# readpipe.sh-
-# Exemplo de read passando um arquivo
-# por um pipe.-
--
-Ultimo="(vazio)"-
-# Passa o script ($0) para o while-
-cat $0 | -
-while read Linha-
-do-
-Ultimo="$Linha"-
-echo "-$Ultimo-"-
-done-
-echo "Acabou, Último=:$Ultimo:"-
 Acabou, Último=:(vazio):
```

Como você viu, o script lista suas próprias linhas com um sinal de menos (-) antes e outro depois de cada uma e, no final, exibe o conteúdo da variável `$Ultimo`. Repare, no entanto, que o conteúdo dessa variável permanece vazio. Ué, será que a variável não foi atualizada? Foi, e isso pode ser comprovado porque a linha `echo "-$Ultimo-"` lista corretamente as linhas. Então por que isso aconteceu?

Como eu disse, o bloco de instruções redirecionado pelo pipe (|) é executado em um subshell e, lá, as variáveis são atualizadas. Quando esse subshell termina, as atualizações das variáveis vão para as profundezas do inferno junto com ele. Repare que vou fazer uma pequena mudança no script, passando o arquivo por redirecionamento de entrada (<), e as coisas passarão a funcionar na mais perfeita ordem:

```
$ cat redirread.sh
#!/bin/bash
# redirread.sh
# Exemplo de read passando o arquivo
# por um pipe.
Ultimo="(vazio)"
# Passa o script ($0) para o while
while read Linha
do
    Ultimo="$Linha"
    echo "-$Ultimo-"
done < $0
echo "Acabou, Último=:$Ultimo:"
```

Veja como ele roda perfeitamente:

```
$ redirread.sh
-#!/bin/bash-
-# redirread.sh-
-# Exemplo de read passando o arquivo
-# por um pipe.-
--
-Ultimo="(vazio)"-
-while read Linha-
-do-
-Ultimo="$Linha"-
-echo "-$Ultimo-"-
-# Passa o script ($0) para o while-
-done < $0-
-echo "Acabou, Último=:$Ultimo:"-
 Acabou, Último=:echo "Acabou,➥
 Último=:$Ultimo:":
```

Bem, amigos da Rede Shell, para finalizar a aula sobre o comando `read` só falta mais um pequeno e importante macete que vou mostrar com um exemplo prático. Suponha que você queira listar um arquivo e quer que a cada dez registros essa listagem pare para que o operador possa ler o conteúdo da tela, e que ela só continue depois de o operador pressionar qualquer tecla. Para não gastar papel (da Linux Magazine), vou fazer essa listagem na horizontal. Meu arquivo (`numeros`) tem 30 registros com números seqüenciais. Veja:

```
$ seq 30 > numeros
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler
# Versão 1
while read Num
do
    let ContLin++   # Contando...
    # -n para não saltar linha
    echo -n "$Num "
    ((ContLin % 10)) > /dev/null || read
done < numeros
```

Na tentativa de fazer um programa genérico criamos a variável `$ContLin` (na vida real, os registros não são somente números seqüenciais) e, quando testamos se o resto da divisão era zero, mandamos a saída para `/dev/null`, pra que ela não apareça na tela. Mas quando fui executar o programa descobri a seguinte zebra:

```
$ 10porpag.sh
1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 ➥
18 19 20 21 23 24 25 26 27 28 29 30
```

Repare que faltou o número 11 e a listagem não parou no `read`. Toda a entrada do loop estava redirecionada para o arquivo `numeros` e a leitura foi feita em cima desse arquivo, perdendo o número 11. Vamos mostrar como deveria ficar o código para que ele passe a funcionar a contento:

```
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler - Versão 2
while read Num
do
    let ContLin++   # Contando...
    # -n para não saltar linha
    echo -n "$Num "
    ((ContLin % 10)) > /dev/null || read ➥
< /dev/tty
done < numeros
```

Repare que agora a entrada do `read` foi redirecionada para `/dev/tty`, que nada mais é senão o terminal corrente, explicitando desta forma que aquela leitura seria feita do teclado e não do arquivo `numeros`. É bom realçar que isso não acontece somente quando usamos o redirecionamento de entrada; se tivéssemos usado o redirecionamento via pipe (`|`), o mesmo teria ocorrido. Veja agora a execução do script:

```
$ 10porpag.sh
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

Isso está quase bom, mas ainda falta um pouco para ficar excelente. Vamos melhorar o exemplo para que você o reproduza e teste (mas, antes de testar, aumente o número de registros em `numeros` ou reduza o tamanho da tela, para que haja quebra de linha).

```
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler
# Versão 3
clear
while read Num
do
    # Contando...
    ((ContLin++))
    echo "$Num"
    ((ContLin % (`tput lines` - 3))) ||
        {
        # para ler qualquer caractere
        read -n1 -p"Tecle Algo " < /dev/tty
        # limpa a tela após a leitura
        clear
        }
done < numeros
```

A mudança substancial feita neste exemplo é com relação à quebra de página, já que ela é feita a cada quantidade-de-linhas-da-tela (`tput lines`) menos (`-`) três, isto é, se a tela tem 25 linhas, o programa listará 22 registros e parará para leitura. No comando `read` também foi feita uma alteração, inserido o parâmetro `-n1` para ler somente um caractere qualquer, não necessariamente um ***ENTER***, e a opção `-p` para exibir uma mensagem.

– Bem meu amigo, por hoje é só porque acho que você já está de saco cheio...
– Num tô não, pode continuar...
– Se você não estiver eu estou... Mas já que você está tão empolgado com o shell, vou te deixar um serviço bastante simples para você melhorar a sua cdteca: Monte toda a tela com um único `echo` e depois posicione o cursor à frente de cada campo para receber o valor que será digitado pelo operador.

Não se esqueçam que, em caso de qualquer dúvida ou falta de companhia para um chope é só mandar um e-mail para julio.neves@gmail.com. Vou aproveitar também para fazer uma propaganda: digam aos amigos que quem estiver a fim de fazer um curso "porreta" de programação em shell deve mandar um e-mail para julio.neves@tecnohall.com.br para informar-se. Até mais!

Informações

[1] Página oficial do Tput: http://www.cs.utah.edu/dept/old/texinfo/tput/tput.html#SEC4

[2] Página oficial do Bash: http://www.gnu.org/software/bash/bash.html