11. Comandos Condicionais

Sumário

11. Comandos Condicionais

- 11.1. O comando if
- 11.2. O comando test
- 11.4. Abreviando o comando test
- 11.5. O novo comando test [[...]]
- 11.6. O novo comando test com metacaracteres
- 11.7. O novo comando test com Expressões Regulares
- 11.8. Usando o operador aritmético
- 11.9. O comando case

11. Comandos Condicionais

Vamulá sem muito mimimi... Veja as linhas de comando a seguir:

```
$ ls musicas
musicas
$ echo $?
0
$ ls ArqInexistente
ls: ArqInexistente: No such file or directory
$ echo $?
1
$ who | grep jneves
jneves   pts/1        Sep 18 13:40 (10.2.4.144)
$ echo $?
0
$ who | grep juliana
$ echo $?
1
```

— O que é esse **$?** faz aí? Começando por cifrão (**$**) parece ser uma variável, certo?

— Sim, é uma variável que contém o código de retorno da última instrução executada. Posso te garantir que se esta instrução foi bem sucedida, **$?** terá o valor zero, caso contrário seu valor será diferente de zero.

11.1. O comando if

O que o nosso comando condicional **if** faz, como vimos no tópico anterior, é testar a variável **$?** **e não uma condição**, como você está **mal** habituado pelas outras linguagens. Então vamos ver a sua sintaxe:

```
if CMD
then
    CMD1
    CMD2
    CMDn
else
    CMD3
    CMD4
    CMDm
fi
```

ou seja: caso comando **CMD** tenha sido executado com sucesso, os comandos do bloco do **then** (**CMD1**, **CMD2** e **CMDn**) serão executados, caso contrário, os comandos executados serão os do bloco opcional do **else** (**CMD3**, **CMD4** e **CMDm**), terminando com um **fi**. E caso você queira, existe também a possibilidade de usar **elif**. É muito raro eu usar essa cláusula já que, como veremos breve, o *Shell* também tem o comando **case**, que foi feito exatamente para isso.

Vamos ver na prática como isso funciona usando um scriptizinho que serve para incluir usuários no **/etc/passwd**:

```
$ cat incusu
#!/bin/bash
# Versão 1
if  grep ^$1 /etc/passwd
then
    echo Usuario \'$1\' já existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' incluído em /etc/passwd
    else
        echo "Problemas no cadastramento. Você é root?"
    fi
fi
```

Repare que o **if** está testando direto o comando **grep** e esta é a sua finalidade. Caso o **if** seja bem sucedido, ou seja, o usuário (cujo nome está em **$1**) foi encontrado em **/etc/passwd**, os comandos do bloco do **then** serão executados (neste exemplo é somente o **echo**). Caso contrário, as instruções do bloco do **else** é que serão executadas, quando um novo **if** testa se o comando **useradd** foi executado a contento, criando o registro do usuário em **/etc/passwd**, ou não, quando dará a mensagem de erro.

Vejamos sua execução, primeiramente passando um usuário já cadastrado:

```
$ incusu jneves
jneves:x:54002:1001:Julio Neves:/home/jneves:/bin/bash
Usuario 'jneves' ja existe
```

Como já vimos diversas vezes, mas é sempre bom insistir no tema para que você já fique precavido, no exemplo dado surgiu uma linha indesejada, ela é a saída do comando **grep**. Para evitar que isso aconteça, devemos desviar a saída desta instrução para **/dev/null** ou fechar o *stderr* (saída de erro primária) usando um **2>&-** (sendo este melhor) e estas são as únicas formas que temos para fazer isso no UNIX (quando digo "no UNIX", pressupõe um "usando sh ou ksh"). Mas a característica do **if** de testar comandos é tão importante, que o GNU **grep** (o do Linux), tem uma implementação especial para isso, que é a opção **-q** (de *quiet*) que não manda nada para a saída, somente devolve o código de retorno (**$?**).

Então nosso *script* ficaria assim:

```
$ cat incusu
#!/bin/bash
# Versão 2
if  grep ^$1 /etc/passwd 2>&-     # ou 2> /dev/null ou, preferencialmente,
                                  # if grep -q ^$1 /etc/passwd
then
    echo Usuario \'$1\' já existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' incluído em /etc/passwd
    else
        echo "Problemas no cadastramento. Você é root?"
    fi
fi
```

Agora vamos testá-lo como usuário normal (não *root*):

```
$ incusu ZeNinguem
./incusu[6]: useradd:  not found
Problemas no cadastramento. Você é root?
```

Epa, aquele erro não era para acontecer! Para evitar que isso aconteça devemos mandar também a saída de erro (*strerr*, lembra?) do **useradd** para **/dev/null**, ficando na versão final assim:

```
$ cat incusu
#!/bin/bash
# Versão 3
if  grep -q ^$1 /etc/passwd
then
    echo Usuario \'$1\' já existe
else
    if  useradd $1 2>&-       # Ou 2> /dev/null
    then
        echo Usuário \'$1\' incluído em /etc/passwd
    else
        echo "Problemas no cadastramento. Você é root?"
    fi
fi
```

Depois destas alterações e de fazer um **su -** ou um **sudo** para me tornar *root*, vejamos o seu comportamento:

```
$ incusu botelho
Usuário 'botelho' incluido em /etc/passwd
```

E novamente:

```
$ incusu botelho
Usuário 'botelho' já existe
```

Lembra que eu falei que ao longo das nossas aulas os nossos programas iriam se aprimorando? Então vejamos agora como poderíamos melhorar o nosso programa para incluir músicas:

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 3)
#
if  grep "^$1$" musicas > /dev/null
then
    echo Este álbum já está cadastrado
else
    echo $1 >> musicas
    sort musicas -o musicas
fi
```

Como você viu, é uma pequena evolução da versão anterior, assim, antes de incluir um registro (que pela versão anterior poderia ser duplicado), testamos se o registro começava (**^**) e terminava (**$**) igual ao parâmetro passado (**$1**). O uso do circunflexo (**^**) no início da cadeia e cifrão (**$**) no fim, são *rexexp* que servem para testar se o parâmetro passado (o álbum e seus dados) são exatamente iguais a algum registro anteriormente cadastrado e não somente igual a um pedaço de algum dos registros.

Vamos executá-lo passando um álbum já cadastrado:

```
$ musinc "album 4^Artista7~Musica7:Artista8~Musica8"
Este álbum já está cadastrado
```

E agora um não cadastrado:

```
$ musinc "album 5^Artista9~Musica9:Artista10~Musica10"
$ cat musicas
album 1^Artista1~Musica1:Artista2~Musica2
album 2^Artista3~Musica3:Artista4~Musica4
album 3^Artista5~Musica5:Artista6~Musica5
album 4^Artista7~Musica7:Artista8~Musica8
album 5^Artista9~Musica9:Artista10~Musica10
```

— Como você viu, o programa melhorou um pouquinho, mas ainda não está pronto. À medida que eu for te ensinando a programar em shell, nossa CDteca irá ficando cada vez melhor.

11.2. O comando test

— Entendi tudo que você me explicou, mas ainda não sei como fazer um **if** para testar condições, ou seja o uso normal do comando.

— Veja bem: você só pode dizer que testar condições é o uso normal do **if**, somente nas linguagens que você conhece e assim mesmo este teste se limita a somente 7 condições:

1. Igual;
2. Não igual;
3. Maior;
4. Menor;
5. Maior ou igual;
6. Menor ou igual e
7. Valores booleanos.

Só. Só isso e mais nada, absolutamente nada, mais. Aqui, como já vimos, isso não é papel do **if**, porque ele está preocupado em tarefas mais nobres, que é testar a execução de um comando. Mas não se desespere, é somente para testar condições que existe o comando **test**, que não manda nada para a saída (*stdout*), somente devolve o seu código de retorno.

Falei em código de retorno (**$?**) me lembro logo do **if**, e é isso mesmo. Para testar condições use o comando **test** e o **if** para testar a saída do **test**. A grande diferença disso é que o **test**, o comando de teste de condiçoes do linux, não testa 7, mas 35 condiçoes e veja agora suas principais opções (existem muitas outras).

Opções do Comando test para arquivos	
Opção	**Verdadeiro se:**
-e arq	**arq** existe
-s arq	**arq** existe e tem tamanho maior que zero
-f arq	**arq** existe e é um arquivo regular
-d arq	**arq** existe e é um diretório;
-r arq	**arq** existe e com direito de leitura
-w arq	**arq** existe e com direito de escrita
-x arq	**arq** existe e com direito de execução
arq1 -nt arq2	**arq1** é mais novo que **arq2**
arq1 -ot arq2	**arq1** é mais antigo que **arq2**
-t FD	Se o descritor de arquivos **FD** não estiver em uso

Veja agora as principais opções para teste de cadeias de caracteres:

Opções do comando test para cadeias de caracteres	
Opção	**Verdadeiro se:**
-z cadeia	Tamanho de **cadeia** é zero
-n cadeia	Tamanho de **cadeia** é maior que zero
cadeia	**cadeia** tem tamanho maior que zero
c1 = c2	As cadeias **c1** e **c2** são idênticas

E pensa que acabou? Engano seu! Agora é que vem o que você está mais acostumado, ou seja as famosas comparações com numéricos. Veja a tabela:

Opções do comando test para números		
Opção	**Verdadeiro se:**	**Significado**
n1 -eq n2	**n1** e **n2** são iguais	equal
n1 -ne n2	**n1** e **n2** não são iguais	not equal
n1 -gt n2	**n1** é maior que **n2**	greater than
n1 -ge n2	**n1** é maior ou igual a **n2**	greater or equal
n1 -lt n2	**n1** é menor que **n2**	less than
n1 -le n2	**n1** é menor ou igual a **n2**	less or equal

Além de tudo, some-se a estas opções as seguintes facilidades:

Operadores	
Operador	**Finalidade**
Parênteses **( )**	Agrupar
Exclamação **!**	Negar
-a	**E** lógico
-o	**OU** lógico

Ufa! Como você viu tem coisa prá chuchu, e como eu te disse no início, o nosso **if** é muito mais poderoso que o dos outros. Vamos ver em uns exemplos como isso tudo funciona, primeiramente testaremos a existência do diretório **dir**:

Exemplo

```
if  test -d dir  # test é um programa, então seus
                 #  argumentos são separados por espaços
then
   cd dir
else
   mkdir dir
   cd dir
fi
```

No exemplo, testei se existia um diretório **dir** definido, caso negativo (**else**), ele seria criado. Já sei, você vai criticar a minha lógica dizendo que o *script* não está otimizado. Eu sei, mas queria que você o entendesse assim, para então poder usar o ponto-de-espantação (**!**) como um negador do **test**. Veja só:

```
if  test ! -d dir  # test é um programa, então seus
                   # argumentos são separados por espaços
then
   mkdir dir
fi
cd dir
```

Desta forma o diretório **dir** seria criado somente se ele ainda não existisse, e esta negativa deve-se ao ponto-de-exclamação (**!**) precedendo a opção **-d**. Ao fim da execução deste fragmento de *script*, o programa estaria com certeza dentro do diretório **dir**.

Vamos ver dois exemplos para entender a diferença da comparação entre números e entre cadeias.

```
cad1=1
cad2=01
if  test $cad1 = $cad2
then
   echo As variáveis são iguais.
else
   echo As variáveis são diferentes.
fi
```

Executando o fragmento de programa acima vem:

```
As variáveis são diferentes.
```

Vamos agora alterá-lo um pouco para que a comparação seja numérica:

```
cad1=1
cad2=01
if  test $cad1 -eq $cad2
then
   echo As variáveis são iguais.
else
   echo As variáveis são diferentes.
fi
```

E vamos executá-lo novamente:

```
As variáveis são iguais.
```

Como você viu nas duas execuções obtive resultados diferentes porque a cadeia **01** é realmente diferente da cadeia **1**, porém, a coisa muda quando as variáveis são testadas numericamente, já que, matemáticamente falando, o número **1** é igual ao número **01**.

Continuando nos exemplos (para ser um anti-man), vamos criar um para mostrar o uso dos conectores **-o** ("**OU**" lógico) e **-a** ("**E**" lógico), veja um exemplo animal feito direto no *prompt* (me desculpem os zoólogos, mas eu não entendendo nada de reino, filo, classe, ordem, família, gênero e espécie, desta forma o que estou chamando de família ou de gênero tem grande chance de estar incorreto):

```
$ Familia=felinae
$ Genero=gato
$ if  test $Familia = canidea -a $Genero = lobo -o $Familia = felinae -a $Genero = leão
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Pode passar a mão
```

Neste exemplo caso o animal fosse da família canídea **E** (**-a**) do gênero lobo, **OU** (**-o**) da familia felina **E** (**-a**) do gênero leão, seria dado um alerta, caso contrário a mensagem seria de incentivo.

Os sinais de maior (**>**) no início das linhas internas ao **if** são os *prompts* de continuação (que estão definidos na variável **$PS2**) e quando o *Shell* identifica que um comando continuará na linha seguinte, automaticamente ele o coloca até que o comando seja encerrado.

Vamos mudar o exemplo para ver se continua funcionando:

```
$ Familia=felino
$ Genero=gato
$ if  test $Familia = felino -o $Familia = canideo -a $Genero = onça -o $Genero = lobo
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Cuidado
```

Obviamente a operação redundou em erro, isto foi porque a opção **-a** tem precedência sobre a **-o**, e desta forma o que primeiro foi avaliado foi a expressão:

```
$Familia = canideo -a $Genero = onça
```

Que foi avaliada como falsa, retornando o seguinte:

```
$Familia = felino -o FALSO -o $Genero = lobo
```

Que resolvida vem:

```
VERDADEIRO -o FALSO -o FALSO
```

Como agora todos conectores são **-o**, e para que uma série de expressões conectadas entre si por diversos **"OU"** lógicos seja verdadeira, basta que uma delas seja, a expressão final resultou como **VERDADEIRO** e o **then** foi executado de forma errada. Para que isso volte a funcionar façamos o seguinte:

```
$ if  test \( $Familia = felino -o $Familia = canideo \) -a \( $Genero = onça -o $Genero = lobo
\)
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Pode passar a mão
```

Desta forma, com o uso dos parênteses agrupamos as expressões com o conector **-o**, priorizando as suas execuções e resultando:

```
 VERDADEIRO -a FALSO
```

Para que seja **VERDADEIRO** o resultado das duas expressões ligadas pelo conector **"E"** (**-a**) é necessário que ambas sejam verdadeiras, o que não é o caso do exemplo acima. Assim o resultado final foi **FALSO** sendo então o **else** corretamente executado.

Se quisermos escolher um CD que tenha faixas de 2 artistas diferentes, nos sentimos tentados a usar um **if** com o conector **-a**, mas é sempre bom lembrarmos que o *bash* nos dá muitos recursos, e isso poderia ser feito de forma muito mais simples com um único comando **grep**, da seguinte maneira:

```
$ grep Artista1 musicas | grep Artista2
```

Da mesma forma para escolhermos CDs que tenham a participação do **Artista1** ou do **Artista2**, não é necessário montarmos um **if** com o conector **-o**. O **egrep** (ou **grep -E**, sendo este mais aconselhável) também resolve isso para nós. Veja como:

```
$ grep -E '(Artista1|Artista2)' musicas
```

Ou (nesse caso específico) o próprio **grep** puro e simples poderia nos quebrar o galho:

```
 $ grep Artista[12] musicas
```

No **grep -E** acima, foi usada uma expressão regular, onde a barra vertical (**|**) trabalha como um **"OU"** lógico e os parênteses são usados para limitar a amplitude deste **"OU"**. Já no **grep** da linha seguinte, a palavra **Artista** deve ser seguida por um dos valores da lista formada pelos colchetes (**[ ]**), isto é, **1** ou **2**.

11.3. O comando test mais amigável

— Tá legal, eu aceito o argumento, o **if** do *Shell* é muito mais poderoso que os outros caretas, mas cá pra nós, essa construção de **if test** ... é muito esquisita, é pouco legível.

— É você tem razão, eu também não gosto disso e acho que ninguém gosta. Acho que foi por isso, que o *Shell* incorporou outra sintaxe que pode ser usada no lugar do comando **test**.

Exemplo

Para isso vamos pegar aquele exemplo para fazer uma troca de diretórios, que era assim:

```
if  test ! -d dir
then
   mkdir dir
fi
cd dir
```

e utilizando a nova sintaxe, vamos fazê-lo assim:

```
if  [ ! -d dir ]    # [] equivalem ao test, então
                    # argumentos devem ter espaços entre si
then
   mkdir dir
fi
cd dir
```

Ou seja, o comando **test** pode ser substituído por um par de colchetes (**[]**), separados por espaços em branco dos argumentos, o que aumentará enormemente a legibilidade, pois o comando **if** ficará com a sintaxe semelhante à das outras linguagens e por isso este será o modo que o comando **test** será usado daqui para a frente.

11.4. Abreviando o comando test

Se você pensa que acabou, está muito enganado. Repare a tabela (tabela verdade) a seguir:

Valores Booleanos	E	OU
VERDADEIRO-VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO-FALSO	FALSO	VERDADEIRO
FALSO-VERDADEIRO	FALSO	VERDADEIRO
FALSO-FALSO	FALSO	FALSO

Ou seja, quando o conector é **"E"** e a primeira condição é verdadeira, o resultado final pode ser **"VERDADEIRO"** ou **"FALSO"**, dependendo da segunda condição, já no conector **"OU"**, caso a primeira condição seja verdadeira, o resultado sempre será **"VERDADEIRO"** e se a primeira for falsa, o resultado dependerá da segunda condição.

Ora, os caras que desenvolveram o interpretador não são bobos e estão sempre tentando otimizar ao máximo os algoritmos. Portanto, no caso do conector **" E"**, a segunda condição não será avaliada, caso a primeira seja falsa, já que o resultado será sempre **"FALSO"**. Já com o **"OU"**, a segunda será executada somente caso a primeira seja falsa.

Aproveitando disso, criaram uma forma abreviada de fazer testes. Batizaram o conector **"E"** de **&&** e o **"OU"** de **||**. Para ver como isso funciona, vamos usá-los como teste no nosso velho exemplo de trocarmos de diretório, que em sua última versão estava assim:

```
if  [ ! -d dir ]
then
   mkdir dir
fi
cd dir
```

Isso também poderia ser escrito da seguinte maneira:

```
[ ! -d dir ] && mkdir dir
cd dir
```

Ou ainda retirando a negação (**!**):

```
[ -d dir ] || mkdir dir
cd dir
```

No primeiro caso, se o primeiro comando (o **test** que está representado pelos colchetes) for bem sucedido, isto é, não existir o diretório **dir**, o **mkdir** será efetuado porque a primeira condição era verdadeira e o conector era **"E"**.

No exemplo seguinte, testamos se o diretório **dir** existia (no anterior testamos se ele não existia) e caso isso fosse verdade, o **mkdir** não seria executado porque o conector era "**OU**". Outra forma:

```
cd dir || mkdir dir
```

Neste caso, se o **cd** fosse mal sucedido, seria criado o diretório **dir** mas não seria feito o **cd** para dentro dele. Para executarmos mais de um comando desta forma, é necessário fazermos um grupamento de comandos, e isso se consegue com o uso de chaves (**{}**). Veja como seria o correto:

```
cd dir || {
   mkdir dir
   cd dir
}
```

Ainda não está legal, porque caso o diretório não exista, o **cd** dará a mensagem de erro correspondente. Então devemos fazer:

```
cd dir 2>&- || {
   mkdir dir
   cd dir
}
```

Como você viu o comando **if** nos permitiu fazer um **cd** seguro de diversas maneiras. É sempre bom lembrarmos que o seguro a que me referi é no tocante ao fato de que ao final da execução você sempre estará dentro de **dir**, desde que você tenha permissão para entrar em **dir**, permissão para criar um diretório em **../dir**, haja espaço em disco, ...

11.5. O novo comando test [[...]]

Ufa! Você pensa que acabou? Ledo engano! Ainda tem uma forma de **test** a mais. Essa é legal porque ela te permite usar padrões para comparação. Estes padrões atendem às normas de **Geração de Nome de Arquivos** (*File Name Generation*, que são ligeiramente parecidas com as *Expressões Regulares*, mas não podem ser confundidas com estas) e usa também Expressões Regulares. A diferença de sintaxe deste para o **test** que acabamos de ver é que esse trabalha com dois pares de colchete da seguinte forma:

```
[[ EXP ]]
```

Onde **EXPR** é uma das que constam na tabela a seguir:

Expressões Condicionais Para Padrões	
Expressão	**Retorna**
cadeia == padrao	Verdadeiro se **cadeia** casa com padrão
cadeia = padrao	Verdadeiro se **cadeia** casa com padrão
cadeia != padrao	Verdadeiro se **cadeia** não casa com padrao
cadeia =~ REGEX	Verdadeiro se **cadeia** casa com *Expressão Regular* REGEX
cadeia1 < cadeia2	Verdadeiro se **cadeia1** vem antes de **cadeia2** alfabeticamente
cadeia1 > cadeia2	Verdadeiro se **cadeia1** vem depois de **cadeia2** alfabeticamente
expr1 && expr2	"**E**" lógico, verdadeiro se ambos **expr1** e **expr2** são verdadeiros
expr1 || expr2	"**OU**" lógico, verdadeiro se **expr1** ou **expr2** for verdadeiro

11.6. O novo comando test com metacaracteres

Esses metacaracteres aos quais me refiro, são aqueles que o *Shell* expande para nomes de arquivos:

- O ponto de interrogação (**?**) casa com somente 1 caractere, seja ele qual for;
- O asterísco (*****) casa com qualquer cadeia;
- A lista (**[...]**) casa com somente um caractere, desde que ele seja explicitado em seu interior.

Exemplo

Vamos ver se uma hora está entre 0 e 12:

```
$ echo $H
13
$ [[ $H == [0-9] || $H == 1[0-2] ]] || echo Hora inválida
 Hora inválida
$H=12
$ [[ $H == [0-9] || $H == 1[0-2] ]] || echo Hora inválida
$
```

Neste exemplo, testamos se o conteúdo da variável **$H** estava compreendido entre zero e nove (**[0-9]**) ou (**||**) se estava entre dez e doze (**1[0-2]**), dando uma mensagem de erro caso não fosse.

Para saber se uma variável tem o tamanho de um e somente um caractere, faça:

```
$ var=a
$ [[ $var == ? ]] && echo var tem um caractere
var tem um caractere
$ var=aa
$ [[ $var == ? ]] && echo var tem um caractere
$
```

É bom que se diga que, além disso que já vimos, o novo **test ([[...]])** também aceita todas as opções que vimos para o antigo **test ([...])** e, só para torná-lo muito mais possante, a partir da versão 3 do bash, passou a aceitar também *Expressões Regulares*.

11.7. O novo comando test com Expressões Regulares

A partir do bash versão 3, foi incorporado a esta forma de teste um operador representado por **=~**, cuja finalidade é fazer comparações com *Expressões Regulares*.

Exemplo

```
$ Cargo=Senador
$ [[ $Cargo =~ ^(Governa|Sena|Verea)dora?$ ]] && echo Tá na Lava Jato?
Tá na Lava Jato?
$ Cargo=Senadora
$ [[ $Cargo =~ ^(Governa|Sena|Verea)dora?$ ]] && echo Tá na Lava Jato?
Tá na Lava Jato?
$ Cargo=Diretor
$ [[ $Cargo =~ ^(Governa|Sena|Verea)dora?$ ]] && echo Tá na Lava Jato?
$
```

Vamos dar uma esmiuçada na *Expressão Regular* **^(Governa|Sena|Verea)dora?$**: ela casa com tudo que começa (**^**) por **Governa**, ou (**|**) **Sena**, ou (**|**) **Verea**, seguido de **dor** e seguido de um **a** opcional (**?**). O cifrão (**$**) serve para marcar o fim. Em outras palavras esta *Expressão Regular* casa com *Governador*, *Senador*, *Vereador*, *Governadora*, *Senadora* e *Vereadora*.

Vamos ver um *script* que confere um horário para ver se ele está certo

```
$ cat criticahora.sh
#!/bin/bash
#  Recebe um horário como parâmetro e confere para ver
#+ se é um horário válido e está no formato HH:MM.

if  [[ $1 =~ ([01][0-9]|2[0-3]):[0-5][0-9] ]]
then
    echo Horário aceito
else
    echo Horário inválido
fi

echo ${BASH_REMATCH[@]}
```

Nesse *script*, usamos um comando **test** com uma *regexp* (**=~**) para conferir o horário que seria passado por parâmetro (**$1**). Primeiramente vemos se a hora está entre 00 e 19 (**[01][0-9]**) **"OU"** (**|**) entre 20 e 23 (**2[0-3]**), seguido de dois pontos (**:**) e depois vem a especificação dos minutos variando entre 00 e 59 (**[0-5][0-9]**). A última linha explico daqui há pouco.

Vamos testá-lo:

1) Passando um horário válido:

```
$ criticahora.sh 12:34
Horário aceito
12:34 12     # <= Gerado pelo cmd da última linha do script
```

2) Passando um horário furado:

```
$ criticahora.sh 32:34
 Horário inválido
             # <= Gerado pelo cmd da última linha do script
```

3) Passando um horário muito furado:

```
$ criticahora.sh 921:129
Horário aceito
21:12 21     # <= Gerado pelo cmd da última linha do script
```

Agora vou explicar a ultima linha e você só vai entender, se tentou aprender *Expressões Regulares*.

BASH_REMATCH é o nome de uma variável do *Shell*, tipo vetor (*array*), que armazena os dados de um casamento via *Expressões Regulares*. Mais à frente, nós veremos no nosso curso, tudo sobre vetores, mas vou dar uma explicação rápida para que você entenda o *script*. Neste último exemplo, se desejássemos listar o primeiro elemento deste vetor (índice 0), teríamos de incluir a seguinte linha:

```
echo ${BASH_REMATCH[0]}
```

Para listar a segunda (indice 1) teríamos de incluir:

```
echo ${BASH_REMATCH[1]}
```

E a sintaxe **echo ${BASH_REMATCH[@]}** serve para listar todos os elementos do *array*. Este vetor possui:

- No índice zero todo o casamento total que ocorreu - Repare que no exemplo muito furado, quando passei **921:129**, este casamento total foi com **21:12;**
- Os índices subsequentes equivalem aos retrovisores gerados pelos grupos. Como o único grupo que existe é (**[01][0-9]|2[0-3]**), ele casou a hora **21**.

Só para firmar as ideias vamos criar um grupo nos minutos só para gerar um novo retrovisor. Veja:

```
$ cat criticahora.sh
#!/bin/bash
#  Recebe um horário como parâmetro e confere para ver
#+ se é um horário válido e está no formato HH:MM.

if  [[ $1 =~ ([01][0-9]|2[0-3]):([0-5][0-9]) ]]
then
    echo Horário aceito
else
    echo Horário inválido
fi

echo ${BASH_REMATCH[@]}
```

E executando vem:

```
$ criticahora.sh 921:129
Horário aceito
21:12 21 12
```

Ou seja, apareceu também o casamento dos minutos e pudemos perceber que houve casamento porque não limitamos a abrangência da nossa *regexp*. Vamos marcar o início (com um **^**) e o fim (com um **$**) e a *regexp* não casará mais com esse horário maluco.

```
$ cat criticahora.sh
#!/bin/bash
#  Recebe um horário como parâmetro e confere para ver
#+ se é um horário válido e está no formato HH:MM.

if  [[ $1 =~ ^([01][0-9]|2[0-3]):([0-5][0-9])$ ]]
then
    echo Horário aceito
else
    echo Horário inválido
fi

echo ${BASH_REMATCH[@]}

$ criticahora.sh 921:129
Horário inválido
                    #  <= Como não houve casamento, o echo ${BASH_REMATCH[@]} voltou vazio
$ criticahora.sh 12:34
Horário aceito
12:34 12 34
```

11.8. Usando o operador aritmético

Você pode usar o operador aritmético (os parênteses duplos) para fazer testes numéricos, veja:

```
Num=5
$ ((Num + 3 == 8)) && echo Resultado é igual a oito
Resultado é igual a oito
$ ((Num = 8 + 3)) && echo $Num é igual a 11
11 é igual a 11
```

Opa! que houve?

Quando usamos o operador aritmético, um sinal de igual (**=**) é uma atribuição. Para comparar, você precisa usar dois sinais de igual (**==**). Veja o que aconteceu:

```
$ echo $Num
11
```

O resultado de **8 + 3** foi atribuído à variável **$Num**. Para fazer testes, aqui podemos usar **==**, **!=**, **>**, **>=**, **<**, **<=**.

Como você pode imaginar, este uso de padrões para comparação, *Expressões Regulares* e os testes aritméticos aumentam muito o poderio do comando **test**. No início deste papo, afirmamos que o comando **if** do interpretador *Shell* é mais poderoso que o seu similar em outras linguagens. Agora que conhecemos todo o seu espectro de funções, diga-me: você concorda ou não com esta assertiva?

11.9. O comando case

Vejamos um exemplo didático: dependendo do valor da variável **$opc** o *script* deverá executar uma das opções: inclusão, exclusão, alteração ou fim. Veja como ficaria este fragmento de *script*:

```
if  [ $opc -eq 1 ]
then
   inclusao    # Chama rotina de inclusão
elif [ $opc -eq 2 ]
then
   exclusao    # Chama rotina de exclusão
elif [ $opc -eq 3 ]
then
   alteracao    # Chama rotina de alteração
elif [ $opc -eq 4 ]
then
   exit
else
   echo Digite uma opção entre 1 e 4
fi
```

Neste exemplo você viu o uso do **elif** como um **else if**, esta é a sintaxe válida e aceita, mas poderíamos fazer melhor, e isto seria com o comando **case**, que tem a sintaxe a seguir:

```
case $Var in
   PADR1) CMD1
          CMD2
          ...
          CMDn ;;
   PADR2) CMD1
          CMD2
          ...
          CMDn ;;
   PADRn) CMD1
          CMD2
          ...
          CMDn ;;

esac
```

Onde a variável **$VAR** é comparada aos padrões **PADR1**, ..., **PADRn** e caso um deles atenda, o bloco de comandos **CMD1**, ..., **CMDn** correspondente é executado até encontrar um duplo ponto-e-vírgula (**;;**), quando o fluxo do programa se desviará para instrução imediatamente após o **esac**.

Na formação dos padrões, são aceitos os seguintes caracteres:

Caracteres Para Formação de Padrões	
Caractere	**Significado**
*	Qualquer caractere ocorrendo zero ou mais vezes
?	Qualquer caractere ocorrendo uma vez
[...]	Lista de caracteres
|	"**OU**" lógico

Como você pode ver, são os metacaracteres que expandem para nomes de arquivos, acrescidos do "**OU**" lógico, representado pela barra vertical (**|**)

Para mostrar como fica melhor, vamos repetir o exemplo anterior, só que desta vez usaremos o **case** e não o **if** ... **elif** ... **else** ... **fi**.

```
case $opc in
   1) inclusao ;;
   2) exclusao ;;
   3) alteracao ;;
   4) exit ;;
   *) echo Digite uma opção entre 1 e 4
esac
```

Como você deve ter percebido, eu usei o asterisco como a última opção, isto é, se o asterisco atende a qualquer coisa, então ele servirá para qualquer coisa que não esteja no intervalo de 1 a 4. Outra coisa a ser notada é que o duplo ponto e vírgula não é necessário antes do **esac**.

Exemplo

Vamos agora fazer um *script* mais radical. Ele te dará bom dia, boa tarde ou boa noite dependendo da hora que for executado, mas primeiramente veja estes comandos:

```
$ date
Tue Nov  9 19:37:30 BRST 2004
$ date +%H
19
```

O comando **date** informa a data completa do sistema, mas ele tem diversas opções para seu mascaramento. Neste comando, a formatação começa com um sinal de mais (**+**) e os caracteres de formatação vêm após um sinal de percentagem (**%**), assim o **%H** significa a hora do sistema. Dito isso vamos ao exemplo:

```
$ cat boasvindas.sh
#!/bin/bash
#  Programa bem educado que
#  dá bom-dia, boa-tarde ou
#  boa-noite conforme a hora
Hora=$(date +%H)
case $Hora in
0? | 1[01]) echo Bom Dia
            ;;
1[2-7]    ) echo Boa Tarde
            ;;
*         ) echo Boa Noite
            ;;
esac
exit
```

Peguei pesado, né? Que nada, vamos esmiuçar a resolução caso a caso (ou seria *case a case*? ☺)

Expressão	Significado
0|1[01]	Significa zero seguido de qualquer coisa (**?**), ou (**|**) um seguido de zero ou um (**[01]**) ou seja, esta linha pegou **01**, **02**, ... **09**, **10** e **11**;
1[2-7]	Significa um seguido da lista de dois a sete, ou seja, esta linha pegou **12**, **13**, ... **17**;
*****	Significa tudo que não casou com nenhum dos padrões anteriores.

O bash 4.0 introduziu duas novas facilidades no comando **case**. A partir desta versão, existem mais dois terminadores de bloco além do **;;**, que são:

Expressão	Significado
;;&	Quando um bloco de comandos for encerrado com este terminador, o programa não sairá do **case**, mas testará os próximos padrões;
;&	Neste caso, o próximo bloco será executado, sem sequer testar o seu padrão.

Exemplo

Suponha que no seu programa possam ocorrer 4 tipos de erro e você resolva representar cada um deles por uma potência de 2, isto é 1, 2, 4 e 8, de forma que a soma dos erros ocorridos gerem um número único para representá-los (é assim que se formam os números binários). Assim, se ocorrem erros dos tipos 1 e 4, será passado **5 (4+1)** para o programa, se os erros forem 1, 4 e 8, será passado **13 (8+4+1)**.

Observe a tabela a seguir:

Soma	Erros			
	8	4	2	1
15	X	X	X	X
14	X	X	X	-
13	X	X	-	X
12	X	X	-	-
11	X	-	X	X
10	X	-	X	-
9	X	-	-	X
8	X	-	-	-
7	-	X	X	X
6	-	X	X	-
5	-	X	-	X
4	-	X	-	-
3	-	-	X	X
2	-	-	X	-
1	-	-	-	X
0	-	-	-	-

```
$ cat case.sh
#!/bin/bash
#  Recebe um código formado pela soma de 4 tipos
#+ de erro e dá as msgs correspondentes. Assim,
#+ se houveram erros tipo 4 e 2, o //script// receberá 6
#+ Se os erros foram 1 e 2, será passado 3. Enfim
#+ os códigos de erro seguem uma formação binária.

Bin=$(bc <<< "obase=2; $1")  #  Passa para binário
Zeros=0000
Len=${#Bin}        #  Pega tamanho de $Bin
Bin=${Zeros:$Len}$Bin        #  Preenche com zeros à esquerda
#  Essa expansão de parâmetros pega da variável $zeros
#+ desde o tamanho de $Bin até o final e concatena o resultado com $Bin.
#+ Poderíamos fazer o mesmo que foi feito acima
#+ com um comando printf, como veremos mais à frente.
#+ experimente fazer:
#+     Bin=101; printf '%04i\n' $Bin
#+ que vc verá.

case $Bin in
    1[01][01][01]) echo Erro tipo 8;;&
    [01]1[01][01]) echo Erro tipo 4;;&
    [01][01]1[01]) echo Erro tipo 2;;&
    [01][01][01]1) echo Erro tipo 1;;&
             0000) echo Não há erro;;&
                *) echo Binário final: $Bin
esac
#  Como sabemos que $Bin só tem zeros e uns, poderíamos ter feito:
#+ case $Bin in
#+     1???) echo Erro tipo 8;;&
#+     ?1??) echo Erro tipo 4;;&
#+     ??1?) echo Erro tipo 2;;&
#+     ???1) echo Erro tipo 1;;&
#+     0000) echo Não há erro;;&
#+        *) echo Binário final: $Bin
#+  esac
```

Repare que todas as opções serão testadas para saber quais são *bits* ligados (**zero**=desligado, **um**=ligado). No final aparece o binário gerado para que você possa comparar com o resultado. Testando:

```
$ case.sh 5
Erro tipo 4
Erro tipo 1
Binário final: 0101
$ case.sh 13
Erro tipo 8
Erro tipo 4
Erro tipo 1
Binário final: 1101
```

Veja também este fragmento de código adaptado de http://tldp.org/LDP/abs/html/bashver4.html

```
case "$1" in
        [[:print:]] )  echo $1 é um caractere imprimível;;&
        # O terminador ;;& testará o próximo padrão
        [[:alnum:]] )  echo $1 é um carac. alfa/numérico;;&
        [[:alpha:]] )  echo $1 é um carac. alfabético   ;;&
        [[:lower:]] )  echo $1 é uma letra minúscula    ;;&
        [[:digit:]] )  echo $1 é um caractere numérico  ;&
        # O terminador ;& executará o próximo bloco...
        %%%@@@^^^-- )  echo "************************"  ;;
        #     ... mesmo com esse padrão maluco.
esac
```

Sua execução passando **3** resultaria:

```
3 é um caractere imprimível
3 é um carac. alfa/numérico
3 é um caractere numérico
********************************
```

Passando **m**:

```
m é um caractere imprimível
m é um carac. alfa/numérico
m é um carac. alfabético
m é uma letra minúscula
```

Passando **/**:

```
/ é um caractere imprimível
```

Aconselho a todos que usam bash que ponham no seu arquivo **~/.bashrc** (mesmo que ele não exista), a seguinte linha:

```
export LC_COLLATE=C
```

Para evitar dissabores, pois sem isso, a ordem dos caracteres do Bash não necessariamente obedecem à ordem que esperamos dele e isso frequentemente dá problemas no uso de listas, principalmente no comando **case**.

— Cara, até agora eu falei muito. Agora eu vou te passar um exercício para você fazer em casa e me dar a resposta na próxima vez que nos encontrarmos por aqui, tá legal?

É o seguinte: quero que você faça um programa que receberá como parâmetro o nome de um arquivo e que quando executado salvará este arquivo com o nome original seguido de um til (**~**) e colocará este arquivo dentro do **vi** (o melhor editor que se tem notícia) para ser editado. Isso é para ter sempre a última cópia boa deste arquivo caso o cara faça alterações indevidas. Obviamente, você fará as críticas necessárias, como verificar se foi passado um parâmetro, se o arquivo passado existe, ... Enfim, o que te der na telha e você achar que deve constar do *script*. Deu prá entender?

— Hum, hum...

12. Comandos de loop ou laço

Sumário

12. Comandos de loop ou laço

12.1. Conceitos de Loop (ou laço)

12.2. O comando for

12.2.1. Primeira sintaxe do comando for

12.2.1.1. Inter Field Separator (IFS)

12.2.2. Segunda sintaxe do comando for

12.2.3. Terceira sintaxe do comando for

12.2.4. Um Pouco Mais de for e Matemática

12.3. O comando while

12.4. O comando until

12.5. Atalhos no loop (continue e break)

12. Comandos de loop ou laço

— Fala cara! E as idéias estão em ordem? Já fundiu a cuca ou você ainda aguenta mais *Shell*?

— Guento! Tô gostando muito! Gostei tanto que até caprichei no exercício que você passou. Lembra que você me pediu para fazer um programa que receberia como parâmetro o nome de um arquivo e que quando executado salvaria este arquivo com o nome original seguido de um til (**~**) e colocaria este arquivo dentro do **vi**?

— Claro que lembro, me mostre e explique como você fez.

```
$ cat vira
#!/bin/bash
#
#  vira - vi resguardando arquivo anterior
#         == =            »

# Verificando se foi passado 1 parametro
if  [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 "
    exit 1
fi

Arq=$1
# Caso o arquivo não exista, nao ha copia para ser salva
if  [ ! -f "$Arq" ]
then
    vi $Arq
    exit 0
fi

# Se nao puder alterar o arquivo vou usar o vi para que?
if  [ ! -w "$Arq" ]
then
    echo "Voce nao tem direito de gravacao em $Arq"
    exit 2
fi

# Ja que esta tudo OK, vou salvar a copia e chamar o vi
cp -f $Arq $Arq~
vi $Arq
exit 0
```

— É, beleza! Mas me diz uma coisa: porque você terminou o programa com um **exit 0**?

— Ahhh! Eu descobri que o número após o **exit** resultará no código de retorno do programa (o **$?**, lembra?), e desta forma, como foi tudo bem sucedido, ele encerraria com o **$? = 0**. Porém se você observar, verá que caso o programa não tenha recebido o nome do arquivo ou caso o operador não tivesse direito de gravação sobre este arquivo, o código de retorno (**$?**) seria diferente do zero.

— Grande garoto, aprendeu legal, mas é bom deixar claro que **exit 0**, simplesmente **exit**, ou não colocar **exit**, produzem igualmente um código de retorno (**$?**) igual a zero. Agora vamos falar sobre as instruções de *loop* ou laço, mas antes vou passar o conceito de bloco de programa.

12.1. Conceitos de Loop (ou laço)

Até agora já vimos alguns blocos de programa, como por exemplo quando te mostrei um código para fazer um **cd** para dentro de um diretório que era assim:

```
cd dir 2> /dev/null || {
    mkdir dir
    cd dir
}
```

O fragmento contido entre as duas chaves (**{}**), forma um bloco de comandos. Também neste exercício que acabamos de ver, em que salvamos o arquivo antes de editá-lo, existem vários blocos de comandos compreendidos entre os **then** e os **else** e entre os **else** e os **fi** do **if**.

Um bloco de comandos também pode estar dentro de um **case**, ou entre um **do** e um **done**.

— Peraí Julio, que **do** e **done** é esse, não me lembro de você ter falado nisso e olha que estou prestando muita atenção...

— Pois é, ainda não havia falado porque não havia chegado o momento propício. Todas as instruções de *loop* ou laço, executam os comandos do bloco compreendido entre o **do** e o **done**.

As instruções de *loop* ou laço são o **for**, o **while** e o **until** que passarei a te explicar uma a uma a partir de agora.

12.2. O comando for

Se você está habituado a programar, certamente já conhece o comando **for**, mas o que você não sabe é que o **for**, que é uma instrução intrinseca do *Shell* (isto significa que o código fonte do comando faz parte do código fonte do *Shell*, ou seja em bom programês é um *builtin*), é muito mais poderoso que os seus correlatos das outras linguagens.

Vamos entender a sua sintaxe, primeiramente em português e depois como funciona no duro.

```
para VAR em VAL1 VAL2 ... VALn
faça
    CMD1
    CMD2
    CMD3
feito
```

Onde a variável **VAR** assume cada um dos valores da lista **VAL1**, **VAL2** ... **VALn** e para cada um desses valores executa o bloco de comandos formado por **CMD1**, **CMD2** e **CMDn**.

Agora que já vimos o significado da instrução em português, vejamos a sintaxe correta:

12.2.1. Primeira sintaxe do comando for

```
for VAR in VAL1 VAL2 ... VALn
do
    CMD1
    CMD2
    CMD3
done
```

Mas veja que no lugar do **do** e o **done**, poderíamos ter usado um abre chaves (**{**) e um fecha chaves (**}**), respectivamente.

Vamos direto para os exemplos, para entender direito o funcionamento deste comando. Vamos escrever um *script* para listar todos os arquivos do nosso diretório separados por dois pontos, mas primeiro veja isso:

```
$ echo *
ArqDoDOS.txt1 confuso incusu logado musexc musicas musinc muslist
```

Isto é, o *Shell* viu o asterisco (*****) e expandiu-o com o nome de todos os arquivos do diretório e o comando **echo** jogou-os para a tela separados por espaços em branco. Visto isso vamos ver como resolver o problema a que nos propusemos:

```
$ cat testefor1
#!/bin/bash
# 1o. Prog didático para entender o for

for Arq in *
do
    echo -n $Arq:     # A opcao -n é para nao saltar linha
done
```

Então vamos executá-lo:

```
$ testefor1
ArqDoDOS.txt1:confuso:incusu:logado:musexc:musicas:musinc:muslist:$
```

Como você viu o *Shell* transformou o asterisco (que odeia ser chamado de asterístico) em uma lista de arquivos separados por espaços em branco. Quando o **for** viu aquela lista, ele disse: "Opa, lista separadas por espaços é comigo mesmo!"

O bloco de comandos a ser executado era somente o **echo**, que com a opção **-n** listou a variável **$Arq** seguida de dois-pontos (**:**), sem saltar a linha. O cifrão (**$**) do final da linha da execução é o *prompt*. que permaneceu na mesma linha também em função da opção **-n**. Outro exemplo simples (por enquanto):

```
$ cat testefor2
#!/bin/bash
# 2o. Prog didático para entender o for

for Palavra in Curso de Shell
do
    echo $Palavra
done
```

E executando vem:

```
$ testefor2
Curso
de
Shell
```

Como você viu, este exemplo é tão bobo e simples como o anterior, mas serve para mostrar o comportamento básico do **for**.

Veja só a força do **for**: ainda estamos na primeira sintaxe do comando e já estou mostrando novas formas de usá-lo. Lá atrás eu havia falado que o **for** usava listas separadas por espaços em branco, mas isso é uma meia verdade, era só para facilitar a compreensão.

12.2.1.1. Inter Field Separator (IFS)

No duro, as listas não são obrigatóriamente separadas por espaços mas antes de prosseguir, deixa eu te mostrar como se comporta uma variável do sistema chamada de **$IFS**. Repare seu conteúdo:

```
$ echo "$IFS" | od -h
0000000 0920 0a0a
0000004
```

Isto é, mandei a variável (protegida da interpretação do *Shell* pelas aspas) para um *dump* hexadecimal (**od -h**) e resultou:

Conteúdo da Variável $IFS	
Hexadecimal	Significado
09	**<TAB>**
20	**<ESPAÇO>**
0a	**<ENTER>**

Onde o último **0a** foi proveniente do **<ENTER>** dado ao final do comando e ele pode ser evitado se você usar a opção **-n** do **echo** que omite este **<ENTER>** automático. Veja:

```
$ echo -n "$IFS" | od -h
0000000 0920 000a
0000004
```

Para melhorar a explicação, vamos ver isso de outra forma:

```
$ echo ":$IFS:" | cat -et
: ^I$
:$
```

Preste atenção na dica a seguir para entender a construção deste comando **cat**:

No comando **cat**, a opção **-e** representa o **<ENTER>** como um cifrão **($)** e a opção **-t** representa o **<TAB>** como um **^I**. Usei os dois-pontos **(:)** para mostrar o início e o fim do **echo**. E desta forma, mais uma vez pudemos notar que os três caracteres estão presentes naquela variável.

Agora veja você, **IFS** significa *Inter Field Separator* ou, traduzindo, separador entre campos. Uma vez entendido isso, eu posso afirmar (porque vou provar) que o comando **for** não usa listas separadas por espaços em branco, mas sim pelo conteúdo da variável **$IFS**, cujo valor padrão (*default*) são esses caracteres que acabamos de ver. Para comprovarmos isso, vamos mostrar um script que recebe o nome do artista como parâmetro e lista as músicas que ele executa, mas primeiramente vamos ver como está o nosso arquivo **musicas**:

```
$ cat musicas
album 1^Artista1~Musica1:Artista2~Musica2
album 2^Artista3~Musica3:Artista4~Musica4
album 3^Artista5~Musica5:Artista6~Musica6
album 4^Artista7~Musica7:Artista1~Musica3
album 5^Artista9~Musica9:Artista10~Musica10
```

Em cima deste "leiaute" foi desenvolvido o script a seguir:

```
$ cat listartista
#!/bin/bash
# Dado um artista, mostra as suas musicas

if  [ $# -ne 1 ]
then
    echo Voce deveria ter passado um parametro
    exit 1
fi

IFS="
:"   # Coloquei o IFS como <ENTER> e ":". No bash, posso fazer assim: IFS=$'\n:'

for ArtMus in $(cut -f2 -d^ musicas)
do
    echo "$ArtMus" | grep $1 && echo $ArtMus | cut -f2 -d~
done
```

O *script*, como sempre, começa testando se os parâmetros foram passados corretamente, em seguida o **IFS** foi setado para **<ENTER>** e dois-pontos (**:**) (como demonstram as aspas em linhas diferentes), porque é ele que separa os blocos **Artistan~Musicam**. Desta forma, a variável **$ArtMus** irá receber cada um destes blocos do arquivo (repare que o **for** já recebe os registros sem o álbum em virtude do **cut** na sua linha). Caso encontre o parâmetro (**$1**) no bloco, o segundo **cut** listará somente o nome da música. Vamos executá-lo:

```
$ listartista Artista1
Artista1~Musica1
Musica1
Artista1~Musica3
Musica3
Artista10~Musica10
Musica10
```

Êpa! Aconteceram duas coisas indesejáveis: os blocos também foram listados e a **Musica10** idem. Além do mais, o nosso arquivo de músicas está muito simples, na vida real, tanto a música quanto o artista têm mais de um nome. Suponha que o artista fosse uma dupla sertaneja chamada *Perereca & Peteleca* (não gosto nem de dar a idéia com receio que isso se torne realidade ☺). Neste caso o **$1** seria *Perereca* e o resto deste lindo nome seria ignorado na pesquisa.

Para que isso não ocorresse, eu deveria passar o nome do artista entre aspas (") ou alterar **$1** por **$@** (que significa todos os parâmetros passados), que é a melhor solução, mas neste caso eu teria que modificar a crítica dos parâmetros e o **grep**. A nova crítica não seria se eu passei um parâmetro, mas pelo menos um parâmetro e quanto ao **grep**, veja só o que resultaria após a substituição do **$*** (que entraria no lugar do **$1**) pelos parâmetros:

```
echo "$ArtMus" | grep perereca & peteleca
```

O que resultaria em erro. O correto seria:

```
echo "$ArtMus" | grep -i "perereca & peteleca"
```

Onde foi colocado a opção **-i** para que a pesquisa ignorasse maiúsculas e minúsculas e as aspas também foram inseridas para que o nome do artista fosse visto como uma só cadeia monolítica.

Ainda falta consertar o erro dele ter listado o **Artista10**. Para isso o melhor é dizer ao **grep** que a cadeia está no início de **$ArtMus** (a expressão regular para dizer que está no início é **^**) e logo após vem um til (**~**). É necessário também que se redirecione a saída do **grep** para **/dev/null** para que os blocos não sejam mais listados. Veja então a nova (e definitiva) cara do programa:

```
$ cat listartista
#!/bin/bash
# Dado um artista, mostra as suas musicas
# versao 2

if  [ $# -eq 0 ]
then
    echo Voce deveria ter passado pelo menos um parametro
    exit 1
fi

IFS="
:"

for ArtMus in $(cut -f2 -d^ musicas)
do
    echo "$ArtMus" | grep -i "^$@~" > /dev/null && echo $ArtMus | cut -f2 -d~
done
```

Que executando vem:

```
$ listartista Artista1
Musica1
Musica3
```

Algumas observações quanto à variável **$IFS:**

- Diversas intruções além do **for** usam esta variável, assim sendo, quando você alterá-la, copie-a antes para outra variável e, após usá-la, volte o seu valor original.
- No bash, pode-se usar as sequências de escape (*escape sequences*), aquelas começadas com uma contrabarra, com a seguinte sintaxe: **VAR=$'\L'** onde **L** é um caractere válido numa sequência de escape.

Exemplo

Como fizemos no último exemplo, para que o **$IFS** receba o **<ENTER>** e os dois pontos (**:**), como conteúdo, bastava ter feito:

```
IFS=$'\n:'
```

Para voltar ao valor padrão (**Espaço <TAB> <ENTER>**), o melhor é fazer:

```
IFS=$' \t\n'
```

12.2.2. Segunda sintaxe do comando for

```
for VAR
do
    CMD1
    CMD2
    CMDn
done
```

— Ué, sem o **in** como ele vai saber que valor assumir?

— Pois é, né? Esta construção a primeira vista parece xquisita mas é bastante simples. Neste caso, **VAR** assumirá um a um cada um dos parâmetros passados para o programa.

Vamos logo aos exemplos para entender melhor. Vamos fazer um *script* que receba como parâmetro um monte de músicas e liste seus autores:

```
$ cat listamusica
#!/bin/bash
# Recebe parte dos nomes de musicas como parâmetro e
# lista os intérpretes. Se o nome for composto, deve
# ser passado entre aspas.
# ex. "Eu nao sou cachorro, não" "Churrasquinho de Mãe"
#
if  [ $# -eq 0 ]
then
    echo Uso: $0 musica1 [musica2] ... [musican]
    exit 1
fi
IFS="
:"
for Musica
do
    echo $Musica
    Str=$(grep -i "$Musica" musicas) || {
        echo "    Não encontrada"
        continue
    }
    for ArtMus in $(echo "$Str" | cut -f2 -d^)
    do
        echo "    $ArtMus" | grep -i "$Musica" | cut -f1 -d~
    done
done
```

Da mesma forma que os outros, começamos o exercício com uma crítica sobre os parâmetros recebidos, em seguida fizemos um **for** em que a variável **$Musica** receberá cada um dos parâmetros passados, colocando em **$Str** todos os álbuns que contém as músicas passadas. Em seguida, o outro **for** pega cada bloco **Artista~Musica** nos registros que estão em **$Str** e lista cada artista que execute aquela música.

Como sempre vamos executá-lo para ver se funciona mesmo:

```
$ listamusica musica3 Musica4 "Eguinha Pocotó"
musica3
    Artista3
    Artista1
Musica4
    Artista4
Eguinha Pocotó
    Não encontrada
```

A listagem ficou feinha porque ainda não sabemos formatar a saída, mas qualquer dia desses, quando você souber posicionar o cursor, fazer negrito, trabalhar com cores e etc, faremos esta listagem novamente usando todas estas perfumarias e ela ficará muito *fashion*.

A esta altura dos acontecimentos você deve estar se perguntando: "E aquele **for** tradicional das outras linguagens em que ele sai contando a partir de um número, com um determinado incremento até alcançar uma condição?"

E é aí que eu te respondo: "Eu não te disse que o nosso **for** é mais porreta que os outros?" Para fazer isso existem duas formas:

1 - Com a primeira sintaxe que vimos, como nos exemplos a seguir direto no *prompt*:

```
$ for i in $(seq 9)
> do
>     echo -n "$i "
> done
1 2 3 4 5 6 7 8 9
```

Neste a variável **i** assumiu os inteiros de 1 a 9 gerados pelo comando **seq** e a opção **-n** do **echo** foi usada para não saltar linha a cada número listado (sinto-me ecologicamente correto por não gastar um monte de papel quando isso pode ser evitado).

Mas ele também poderia ser resolvido (e rodando mais rápido), usando expansão de chaves. Veja:

```
$ for i in {1..9}
> do
>     echo -n "$i "
> done
1 2 3 4 5 6 7 8 9
```

Ainda usando o **for** com **seq**:

```
$ for i in $(seq 3 9)
> do
>     echo -n "$i "
> done
3 4 5 6 7 8 9
```

Esse também poderia ser resolvido com expansão de chaves. Bastaria trocar **$(seq 3 9)** por **{3..9}**. Experimente. Ou ainda na forma mais completa do **seq**:

```
$ for i in $(seq 0 3 9)
> do
>     echo -n "$i "
> done
0 3 6 9
```

Esse é outro que poderia ser resolvido com expansão de chaves. Bastaria trocar **$(seq 0 3 9)** por **{0..9..3}**. Experimente.

2 - A outra forma de fazer o desejado é com uma sintaxe muito semelhante ao **for** da linguagem C, como veremos em breve.

12.2.3. Terceira sintaxe do comando for

```
for ((VAR=INI; COND; INCR))
do
    CMD1
    CMD2
    CMDn
done
```

Onde:

- **VAR=INI** - Significa que a variável **VAR** será inicializada com o valor **INI;**
- **COND** - Siginifica que o *loop* ou laço do **for** será executado enquanto **VAR** não atingir a condição **COND;**
- **INCR** - Significa o incremento (positivo ou negativo) que a variável **VAR** sofrerá em cada passada do *loop*.

Como sempre vamos aos exemplos que a coisa fica mais fácil:

```
$ for ((i=1; i<=9; i++))
> do
>     echo -n "$i "
> done
1 2 3 4 5 6 7 8 9
```

Neste caso a variável **i** partiu do valor inicial 1, o bloco de comando (neste caso somente o **echo**) será executado enquanto **i** menor ou igual (**<=**) a 9 e o incremento de **i** será de 1 a cada passada do *loop*.

Repare que no **for** propriamente dito (e não no bloco de comandos) não coloquei um cifrão (**$**) antes do **i**, e a notação para incrementar (**i++**) é diferente do que vimos até agora. Isto é porque o uso de parênteses duplos (assim como o comando **let**) chama o interpretador aritmético do *Shell*, que sabe que você, por ser um cara esperto, jamais usaria um literal dentro de um comando aritmético e, assim sendo, interpreta qualquer literal, como um nome de variável, desde que seja válido para tal.

Como me referi ao comando **let**, só para mostrar como ele funciona e a versatilidade do **for**, vamos fazer a mesma coisa, porém omitindo a última parte do escopo do **for**, passando-a para o bloco de comandos. Vamos também omitir a inicialização de **$i**, mas não esqueça que após a execução do último exemplo, seu valor atual é **9**.

```
$ for ((; i<=19;))
> do
>     let i++
>     echo -n "$i "
> done
10 11 12 13 14 15 16 17 18 19 20
```

Nesse caso o **for** levou **$i** até **19** mas, como o incremento (**let**) fica antes do **echo** a listagem foi até **20**. Assim fazendo o incremento saiu do corpo do **for** e passou para o bloco de comandos, repare também que quando usei o **let**, não foi necessário sequer inicializar a variável **$i**.

Veja só os comandos a seguir dados diretamente no *prompt* para mostrar o que acabo de falar sobre a não inicialização da variável:

```
$ unset j     # Se $j existia, matei
$ echo $j

$ let j++
$ echo $j
1
```

Ou seja, a variável **$j** sequer existia e no primeiro **let** assumiu o valor **0** (zero) para, após o incremento, ter o valor **1**. Veja só como as coisas ficam simples:

```
$ for arq in *
> do
>     let i++
>     echo "$i -> $Arq"
> done
1 -> ArqDoDOS.txt1
2 -> confuso
3 -> incusu
4 -> listamusica
5 -> listartista
6 -> logado
7 -> musexc
8 -> musicas
9 -> musinc
10 -> muslist
11 -> testefor1
12 -> testefor2
```

Repare que o **for** usa **((...))** em sua sintaxe, e acabamos de falar no **let**. Ambos foram tratados pelo interpretador aritmético do *Shell*. Usando ainda esse interpretador, podemos diminuir este código, fazendo:

```
$ for arq in *
> do
>     echo "$((++i)) -> $Arq"
> done
1 -> ArqDoDOS.txt1
2 -> confuso
3 -> incusu
4 -> listamusica
5 -> listartista
6 -> logado
7 -> musexc
8 -> musicas
9 -> musinc
10 -> muslist
11 -> testefor1
12 -> testefor2
```

Esta 3ª. sintaxe do **for** é idêntica à do C. A diferença é que a do *Shell* usa parênteses duplos, para chamar o interpretador aritmético.

Vamos agora fazer um pequeno exercício para consolidar os conceitos que acabamos de ver. Faça um script para contar a quantidade de palavras de um arquivo texto, cujo nome seria recebido como parâmetro. **IMPORTANTE:** Essa contagem tem de ser feita usando o comando **for**, para que você se habitue ao seu uso. Não vale usar o **wc -w**.

Vamos relembrar como é o arquivo **ArqDoDOS.txt**.

```
$ cat ArqDoDOS.txt
Este arquivo
foi gerado pelo
DOS/Rwin e foi
baixado por um
ftp mal feito.
```

Agora vamos testar o programa passando este arquivo como parâmetro:

```
$ contpal.sh ArqDoDOS.txt
O arquivo ArqDoDOS.txt tem 14 palavras
```

Só para conferir:

```
$ wc -w < DOS.txt
14
```

— Beleza, funcionou legal!

12.2.4. Um Pouco Mais de for e Matemática

Voltando à vaca fria, na última vez que aqui estivemos, terminamos o nosso papo mostrando o *loop* de **for** a seguir:

```
for ((; i<=9;))
do
   let i++
   echo -n "$i "
done
```

Uma vez que chegamos neste ponto, creio ser bastante interessante citar que o *Shell* trabalha com o conceito de "Expansão Aritmética" (*Arithmetic Expansion*), que costumo chamar de interpretador aritmético, nome que considero mais apropriado. Como já vimos, a expansão aritmética é acionada por uma construção da forma:

```
 $((EXPR))
```

ou

```
 let EXPR
```

No último **for** citado usei a expansão das duas formas, mas não poderíamos seguir adiante sem saber que a expressão **EXPR** deve ser constituída unicamente por números ou um dos operadores da tabela seguir listadas a seguir:

Expansão Aritmética	
Operadores	**Resultado**
id++, **id--**	pós-incremento e pós-decremento de variáveis
++id, **-?id**	pré-incremento e pré-decremento de variáveis
******	exponenciação
*****, **/**, **%**	multiplicação, divisão, resto da divisão
+,**-**	adição, subtração
<=, **>=**, **<**, **>**	comparação
==, **!=**	igualdade, desigualdade
&&	**E** lógico
||	**OU** lógico

E aí você pode me perguntar:

— Ué! Mas quando você explicou o comando **for**, você deu exemplos de aritmética usando variáveis...

— É verdade, usei variáveis, porém não se esqueça - te disse isso na primeira aula - que o *Shell* substitui as variáveis por seu valores antes de executar o comando. Desta forma o *Shell* substituiu o valor da variável e o interpretador aritmético já o recebeu como um número. Mas vamos em frente que atrás vem gente: você pensa que o papo de *loop* (ou laço) se encerra no comando **for?** Ledo engano amigo, vamos a partir de agora ver mais dois.

12.3. O comando while

Todos os programadores conhecem este comando, porque ele é comum a todas as linguagens e nelas, o que normalmente ocorre é que um bloco de comandos é executado, **enquanto** (enquanto, em inglês, é **while**) uma determinada condição for verdadeira. Pois bem, isto é o que ocorre nas linguagens caretas! Em programação *Shell*, o bloco de comandos é executado enquanto um comando monitorado for verdadeiro. E é claro, se quiser testar uma condição use o comando **while** testando o comando **test**, exatamente como você aprendeu a fazer no **if**, lembra?

Então a sintaxe do comando fica assim:

```
while COMANDO
do
   CMD1
   CMD2
   ...
   CMDn
done
```

e desta forma o bloco de comandos formado pelas instruções **CMD1**, **CMD2**, ..., **CMDn** é executado enquanto a execução da instrução **COMANDO** (que pode ser só uma instrução ou diversas interligadas por *pipes*) for bem sucedida.

Suponha a seguinte cena: tem um verdadeiro avião, pousado na porta do meu trabalho, me esperando e eu preso no trabalho sem poder sair porque o meu chefe, que é um pé no saco (aliás chefe-chato é uma redundância, né?:), ainda estava na sua sala, que fica bem na minha passagem para a rua.

Ele começou a ficar com as antenas (provavelmente instaladas na cabeça dele pela esposa) ligadas depois da quinta vez que passei pela sua porta e olhei para ver se já havia ido embora. Então voltei para a minha mesa e fiz, no servidor, um *script* assim:

```
$ cat logaute.sh
#!/bin/bash

# Espero que a Xuxa não tenha
# copyright de xefe e xato :)

while who | grep xefe
do
    sleep 30
done
echo O xato se mandou, não hesite, dê exit e vá a luta
```

Neste *scriptizinho*, o comando **while** testa o *pipeline* composto pelo **who** e pelo **grep** e que será verdadeiro enquanto o **grep** localizar a palavra **xefe** na saída do **who**. Desta forma, o *script* dormirá por 30 segundos enquanto o chefe estiver *logado* (Argh!). Assim que ele se desconectar do servidor, o fluxo do *script* sairá do *loop* e dará a tão ansiada mensagem de liberdade.

Quando o executei adivinha o que aconteceu?

```
$ logaute.sh
xefe  pts/0   Jan  4 08:46 (10.2.4.144)
xefe  pts/0   Jan  4 08:47 (10.2.4.144)
...
xefe  pts/0   Jan  4 08:52 (10.2.4.144)
```

Isto é a cada 30 segundos seria enviado para a tela a saída do **grep**, o que não seria legal já que poluiria a tela do meu micro e a mensagem esperada poderia passar despercebida. Para evitar isso já sabemos que a saída do *pipeline* tem que ser redirecionada para **/dev/null** (uma outra forma de fazer o mesmo, seria usar a opção **-q** (*quiet*) do **grep**, como já vimos, mas isso funciona somente no **GNU grep**, não funciona no UNIX).

```
$ cat logaute.sh
#!/bin/bash

# Espero que a Xuxa não tenha
# copyright de xefe e xato :)

while who | grep -q xefe
do
    sleep 30
done
echo O xato se mandou, não hesite, dê exit e vá a luta
```

Agora quero montar um *script* que receba o nome (e eventuais parâmetros) de um programa que será executado em *background* e que me informe do seu término. Mas, para você entender este exemplo, primeiro tenho de mostar uma nova variável do sistema. Veja estes comandos diretos no *prompt*:

```
$ sleep 10&
[1] 16317
$ echo $!
16317
[1]+  Done                    sleep 10

$ echo $!
16317
```

Isto é, criei um processo em *background* para dormir por 10 segundos, somente para mostrar que a variável **$!** guarda o **PID** (*Process IDentification*) do último processo em *background*, mas repare que mesmo após o termino (mostrado pela linha do **done**), a variável reteve este valor. Como disse, a variável guarda o PID do último processo em *background*, assim sendo, ela só perderá esse valor quando outro processo for mandado para *background*.

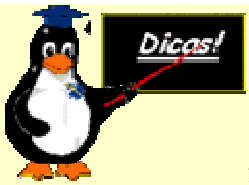
Bem sabendo isso já fica mais fácil de monitorar qualquer processo em *background*. Veja só como:

```
$ cat monbg.sh
#!/bin/bash

# Executa e  monitora um
# processo em background

$1 &      # Coloca em backgroud
while ps | grep -q $!
do
    sleep 5
done
echo Fim do Processo $1
```

Este *script* é bastante similar ao anterior, mas tem uns macetes a mais, veja só: ele tem que ser executado em *background* para não prender o *prompt*, mas o **$!** será o do programa passado como parâmetro já que ele foi colocado em *background* após o **monbg.sh** propriamente dito. Repare também a opção **-q** (*quiet*) do **grep**, ela serve para tranformá-lo num comando mineiro, isto é, para o **grep** "trabalhar em silêncio". O mesmo resultado poderia ser obtido se a linha fosse **while ps | grep $! > /dev/null**, como nos exemplos que vimos até agora.



Não esqueça: o Bash disponibiliza a variável **$!** que possui o **PID** (*Process IDentification*) do último processo executado em *background*.

Vamos melhorar o **musinc**, que é o nosso programa para incluir registros no arquivo **musicas**, mas antes preciso te ensinar a pegar um dado da tela, e já vou avisando: só vou dar uma pequena dica do comando **read** que seja o suficiente para resolver este nosso problema (é o **read** quem pega o dado da tela). Em outra aula vou te ensinar tudo sobre o assunto, inclusive como formatar tela, mas hoje estamos falando sobre *loops*.

A sintaxe do comando **read** que nos interessa por hoje é a seguinte:

```
read -p "prompt de leitura" var
```

Onde **prompt de leitura** é o texto que você quer que apareça escrito na tela, solicitando a digitação e o dado irá para a variável **var** quando o operador teclá-lo. Por exemplo:

```
$ read -p "Título do Álbum: " Tit
```

Bem, uma vez entendido isso, vamos à especificação do nosso problema: faremos um programa que inicialmente lerá o nome do álbum e em seguida fará um *loop* de leitura, pegando a música e o artista. Este *loop* termina quando for informada uma música vazia, isto é, ao ser solicitada a digitação da música, o operador dá um simples **<ENTER>**. Para facilitar a vida do operador, vamos oferecer como *default* o mesmo nome do artista da música anterior (já que é normal que o álbum seja todo do mesmo artista) até que ele deseje alterá-lo.

Vamos ver como ficou:

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 4)
#
clear
read -p "Título do Álbum: " Tit
[ "$Tit" ] || exit 1    # Fim da execução se título vazio
if grep "^$Tit\^" musicas > /dev/null # Pesquisa linhas começadas (regex ^) por $Tit seguida de
um circunflexo
then
    echo Este álbum já está cadastrado
    exit 1
fi
Reg="$Tit^"
Cont=1
oArt=Desconhecido
while true
do
    echo Dados da trilha $Cont:
    read -p "Música: " Mus
    [ "$Mus" ] || break       # Sai se vazio
    read -p "Artista: $oArt // " Art
    [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
    Reg="$Reg$oArt~$Mus:"     # Montando registro
    Cont=$((Cont + 1))
    # A linha anterior tb poderia ser Cont=$((Cont++)) ou let Cont++
done
echo "$Reg" >> musicas
sort musicas -o musicas
```

Este exemplo, começa com a leitura do título do álbum, que se não for informado, terminará a execução do programa. Em seguida um **grep** procura no início (**^**) de cada registro de musicas, o título informado seguido do separador (**^**) (que está precedido de uma contrabarra (****) para que o *Shell* não o interprete como uma *regex* (*Expressão Regular*).

Para ler os nomes dos artistas e as músicas do álbum, foi montado um *loop* de **while** simples, cujo único destaque é o fato de estar armazenando o artista da música anterior na variável **$oArt** que só terá o seu conteúdo alterado, quando algum dado for informado para a variável **$Art**, isto é, quando não teclou-se um simples **<ENTER>** para manter o artista anterior.

O que foi visto até agora sobre o **while** foi muito pouco. Este comando é muito utilizado, principalmente para leitura de arquivos, porém nos falta bagagem para prosseguir. Depois que aprendermos a ler, veremos esta instrução mais a fundo.

Leitura de arquivo significa ler um a um todos os registros, o que é sempre uma operação lenta. Fique atento para não usar o **while** quando seu uso for desnecessário. O *Shell* tem ferramentas como o **sed** e a família **grep** que vasculham arquivos de forma otimizada sem ser necessário o uso de comandos de *loop* para fazê-lo registro a registro (ou até palavra a palavra).

12.4. O comando until

O comando **until** funciona exatamente igual ao **while**, porém ao contrário. Disse tudo mas não disse nada, né? É o seguinte: ambos testam comandos; ambos possuem a mesma sintaxe e ambos atuam em *loop*, porém enquanto o **while** executa o bloco de intruções do *loop* **enquanto** um comando for bem sucedido, o **until** executa o bloco do *loop* **até que** o comando seja bem sucedido. Parece pouca coisa mas a diferença é fundamental.

A sintaxe do comando é praticamente a mesma do **while**. Veja:

```
until COMANDO
do
   CMD1
   CMD2
   ...
   CMDn
done
```

E desta forma o bloco de comandos formado pelas instruções **CMD1**, **CMD2**, ... **CMDn** é executado até que a execução da instrução **COMANDO** seja bem sucedida.

Como eu te disse, o **while** e **until** funcionam de forma antagônica e isso é muito fácil de demonstrar: em uma guerra sempre que se inventa uma arma, o inimigo busca uma solução para neutralizá-la. Baseado neste principio belicoso que o meu chefe, desenvolveu, no mesmo servidor que eu executava o **logaute.sh** um *script* para controlar o meu horário de chegada.

Um dia deu um problema da rede, ele me pediu para dar uma olhada no micro dele e me deixou sozinho em sua sala. Imediatamente comecei a bisbilhotar seus arquivos - porque guerra é guerra - e veja só o que descobri:

```
$ cat chegada.sh
#!/bin/bash

until who | grep julio
do
    sleep 30
done
echo $(date "+ Em %d/%m às %H:%Mh") >> relapso.log
```

Olha que safado! O cara estava montando um log com os horários que eu chegava, e ainda por cima chamou o arquivo que me monitorava de **relapso.log**! O que será que ele quis dizer com isso?

Neste *script*, o *pipeline* **who | grep julio**, será bem sucedido somente quando **julio** for encontrado no comando **who**, isto é, quando eu me "logar" no servidor. Até que isso aconteça, o comando **sleep**, que forma o bloco de instruções do **until**, porá o programa em espera por 30 segundos. Quando este *loop* encerrar-se, será dada uma mensagem para o **relapso.log** (ARGHH!). Supondo que no dia 20/01 eu me loguei às 11:23 horas, a mensagem seria a seguinte:

```
Em 20/01 às 11:23h
```

Quando vamos cadastrar músicas, o ideal seria que pudéssemos cadastrar diversos CDs, e na última versão que fizemos do **musinc**, isso não ocorre, a cada CD que cadastramos o programa termina.

Vejamos como melhorá-lo:

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 5)
#
Para»
until [ "$Para" ]
do
    clear
    read -p "Título do Álbum: " Tit
    if  [ ! "$Tit" ] # Se titulo vazio...
    then
        Para=1     # Liguei flag de saída
    else
        if grep "^$Tit\^" musicas > /dev/null
        then
            echo Este álbum já está cadastrado
            exit 1
        fi
        Reg="$Tit^"
        Cont=1
        oArt»
        while [ "$Tit" ]
        do
            echo Dados da trilha $Cont:
            read -p "Música: " Mus
            [ "$Mus" ] || break       # Sai se vazio
            read -p "Artista: $oArt // " Art
            [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
            Reg="$Reg$oArt~$Mus:"     # Montando registro
            Cont=$((Cont + 1))
            # A linha anterior tb poderia ser Cont=$((Cont++)) ou let Cont++
        done
        echo "$Reg" >> musicas
        sort musicas -o musicas
    fi
done
```

Nesta versão, um *loop* maior foi adicionado antes da leitura do título, que só terminará quando a variável **$Para** deixar de ser vazia. Caso o título do álbum não seja informado, a variável **$Para** receberá valor (no caso coloquei **1** mas poderia ter colocado qualquer coisa. O importante é que não seja vazia) para sair deste *loop*, terminando desta forma o programa. No resto, o *script* é idêntico à sua versão anterior.

12.5. Atalhos no loop (continue e break)

Nem sempre um ciclo de programa, compreendido entre um **do** e um **done**, sai pela porta da frente. Em algumas oportunidades, temos que colocar um comando que aborte de forma controlada este *loop*. De maneira inversa, algumas vezes desejamos que o fluxo de execução do programa volte antes de chegar ao **done**. Para isto, temos respectivamente, os comandos **break** (que já vimos rapidamente nos exemplos do comando **while**) e **continue**, que funcionam da seguinte forma:

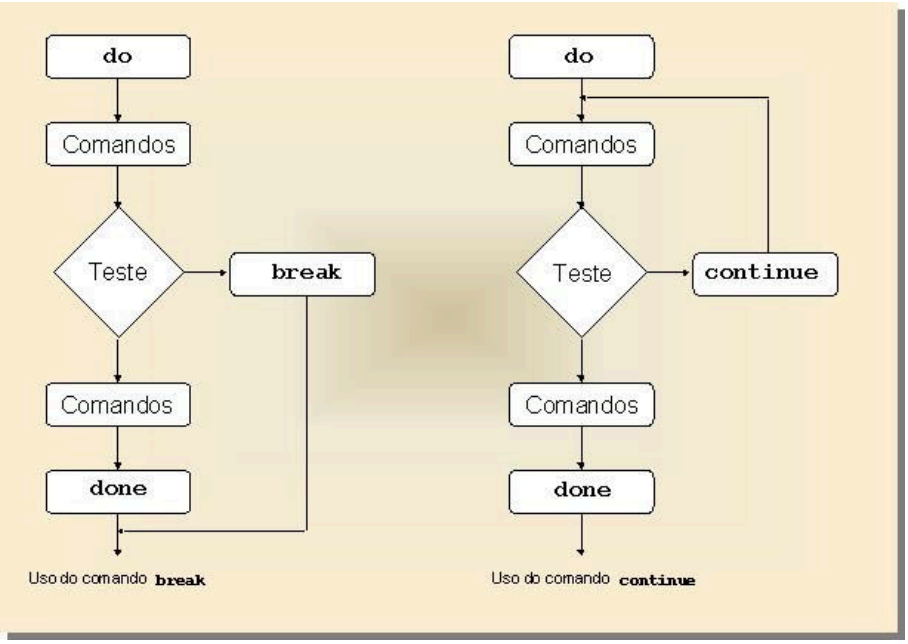
O que eu não havia dito anteriormente é que nas suas sintaxes genéricas eles aparecem da seguinte forma:

```
break [INT]
```

e

```
continue [INT]
```

Onde **INT** é um número inteiro que representa a quantidade dos *loops* mais internos sobre os quais os comandos irão atuar. Seu valor *default* é **1**.



Duvido que você nunca tenha deletado um arquivo e logo após deu um tabefe na testa se xingando porque não devia tê-lo removido. Pois é, na décima vez que fiz esta besteira, criei um *script* para simular uma lixeira, isto é, quando mando remover um (ou vários) arquivo(s), o programa "finge" que removeu-o, mas no duro o que fez foi mandá-lo(s) para o diretório **/tmp/LoginName_do_usuario**. Chamei este programa de **erreeme** e no **/etc/profile** coloquei a seguinte linha:

```
alias rm=erreeme
```

O programa era assim:

```
$ cat erreeme
#/bin/bash
#
#  Salvando Copia de Arquivo Antes de Remove-lo
#

if  [ $# -eq 0 ]  #  Tem de ter um ou mais arquivos para remover
then
    echo "Erro -> Uso: erreeme arq [arq] ... [arq]"
    echo "  O uso de metacaracteres é permitido. Ex. erreeme arq*"
    exit 1
fi

MeuDir="/tmp/$LOGNAME" # Variavel do sist. Contém o nome do usuário.
if  [ ! -d $MeuDir ]   # Se não existir o meu diretório sob o /tmp...
then
    mkdir $MeuDir      # Vou cria-lo
fi

if  [ ! -w $MeuDir ]   # Se não posso gravar no diretório...
then
    echo Impossivel salvar arquivos em $MeuDir. Mude permissao...
    exit 2
fi

Erro=0                 # Variavel para indicar o cod. de retorno do prg
for Arq                # For sem o "in" recebe os parametros passados
do
    if  [ ! -f $Arq ]  # Se este arquivo não existir...
    then
   echo $Arq nao existe.
   Erro=3
   continue       # Volta para o comando for
    fi

 DirOrig=`dirname $Arq` # Cmd. dirname informa nome do dir de $Arq
    if  [ ! -w $DirOrig ]  # Verifica permissão de gravacaoo no diretório
    then
   echo Sem permissao de remover no diretorio de $Arq
   Erro=4
   continue           # Volta para o comando for
    fi

    if  [ "$DirOrig" = "$MeuDir" ] # Se estou "removendo da lixeira"...
    then
   echo $Arq ficara sem copia de seguranca
   rm -i $Arq         # Pergunta antes de remover
   [ -f $Arq ] || echo $Arq removido  # Será que o usuario removeu?
   continue
    fi

    cd $DirOrig      #  Guardo no fim do arquivo o seu diretorio
    pwd >> $Arq      #+ original para usa-lo em um script de undelete
    mv $Arq $MeuDir  # Salvo e removo
    echo $Arq removido
done
exit $Erro # Passo eventual numero do erro para o codigo de retorno
```

Como você pode ver, a maior parte do *script* é formada por pequenas criticas aos parâmetros informados, mas como o *script* pode ter recebido diversos arquivos para remover, a cada arquivo que não se encaixa dentro do especificado, há um **continue**, para que a sequência volte para o *loop* do **for** de forma a receber outros arquivos.

Quando você está no Windows (com perdão da má palavra) e tenta remover aquele monte de lixo com nomes esquisitos como **HD04TG.TMP**, se der erro em um deles, os outros não são removidos, não é? Então, o **continue** foi usado para evitar que um impropério desses ocorra, isto é, quando um erro ocorre na remoção de um arquivo, o programa dará uma mensagem de erro, ficará marcado que houve pelo menos erro no lote, mas continuará removendo os outros que foram passados.

— Eu acho que a esta altura você deve estar curioso para ver o programa que restaura o arquivo removido, não é? Pois então aí vai um desafio: faça-o e vamos discuti-lo na nossa próxima aula.

— Cara, este programa é como tudo que se faz em *Shell*, extremamente fácil, é para ser feito em, no máximo 10 linhas. Não se esqueça que o arquivo está salvo em **/tmp/$LOGNAME** e que a sua última linha é o diretório em que ele residia antes de ser "removido". Também não se esqueça de criticar se foi passado o nome do arquivo a ser removido.

Sumário

13. Usando a tela
13.1. Formatando com o comando tput
13.2. Recebendo dados com o comando read
13.2.1. O comando read - Vamos ler arquivos?

13. Usando a tela

— Cumequié rapaz! Derreteu os pensamentos para fazer o scriptizinho que eu te pedi?

— É, eu realmente tive de colocar muita *pensação* na tela preta mas acho que consegui! Bem, pelo menos no testes que fiz a coisa funcionou, mas você tem sempre que botar chifres em cabeça de cachorro!

— Não é bem assim, programar em *Shell* é muito fácil, o que vale são as dicas e macetes que não são triviais. As observações que faço, são justamente para te mostrar os macetes e os pulos do gato. Vamos então olhar o seu *script*.

```
$ cat restaura
#!/bin/bash
#
# Restaura arquivos deletados via erreeme
#

if [ $# -eq 0 ]
then
    echo "Uso: $0 <Nome do Arquivo a Ser Restaurado>"
    exit 1
fi
# Pega nome do diretório original na última linha
Dir=$(tail -1 /tmp/$LOGNAME/$1)
# O grep -v exclui última linha e cria o
# arquivo com diretorio e nome originais
grep -v $Dir /tmp/$LOGNAME/$1 > $Dir/$1
# Remove arquivo que jah estava moribundo
rm /tmp/$LOGNAME/$1
```

— Peraí, deixa ver se entendi. Primeiramente você coloca na variável **Dir** a última linha do arquivo cujo nome é formado por **/tmp/<nome do operador> ($LOGNAME)/<parâmetro passado> ($1)** como nome do arquivo a ser restaurado. Em seguida o **grep -v** que você montou exclui a linha em que estava o nome do diretório, isto é, sempre a última e manda o restante do arquivo, que seria assim o arquivo já limpo, para o diretório original e depois remove o arquivo da "lixeira"; **S E N S A C I O N A L!** Impecável! Zero erro! Viu? você já está pegando as manhas do *Shell*!

— Então vamulá chega de lesco-lesco e blá-blá-blá, de que você vai falar hoje?

— É, tô vendo que o bichinho do *Shell* te pegou. Que bom, mas vamos ver como se pode (e deve) ler dados e formatar telas e primeiramente vamos entender um comando que te dá todas as ferramentas para você formatar a sua tela de entrada de dados.

13.1. Formatando com o comando tput

O maior uso deste comando é para posicionar o cursor na tela, mas também é muito usado para apagar dados da tela, saber a quantidade de linhas e colunas para poder posicionar corretamente um campo, apagar um campo cuja crítica detectou como errado. Enfim, quase toda a formatação da tela é feita por este comando.

Uns poucos atributos do comando **tput** podem eventualmente não funcionar se o modelo de terminal definido pela variável **$TERM** não tiver esta facilidade incorporada.

Na tabela a seguir, apresento os principais atributos do comando e os efeitos executados sobre o terminal, mas veja bem, existem muito mais do que esses, veja só:

```
$ tput it
8
```

Neste exemplo eu recebi o tamanho inicial da **<TAB>** (***I**nitial **T**ab*), mas me diga: para que eu quero saber isso? Se você quiser saber tudo sobre o comando **tput** (e olha que é coisa que não acaba mais), consulte a referência completa desenvolvida pela Escola de Computação da Universidade de Utah.

Principais Opções do Comando tput	
Opções do tput	**Efeito**
cup LIN COL	***CU**rsor **P**osition* - Posiciona o cursor na linha **LIN** e coluna **COL.** A origem é zero
home	O mesmo que **tput cup 0 0**, isto é, coloca o cursor no vertice superior esquerdo da tela
bold	Coloca a tela em modo de ênfase (negrito)
rev	Coloca a tela em modo de vídeo reverso
smso	Idêntico ao anterior
smul	A partir desta instrução, os caracteres teclados aparecerão sublinhados na tela
blink	Os caracteres teclados aparecerão piscando (nem todas as definições de terminais aceitam esse)

invis	A partir desse ponto, nada do que for digitado ou mandado para a tela aparecerá no vídeo (Obs: Inseguro para ler senhas)
sgr0	Após usar um dos atributos acima, use este para restaurar a tela ao seu modo normal
reset	Limpa o terminal e restaura suas definições de acordo com o **terminfo** ou seja, o terminal volta ao padrão definido pela variável **$TERM**
lines	Devolve a quantidade de linhas da tela no momento da instrução
cols	Devolve a quantidade de colunas da tela no momento da instrução
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
ech N	***E****rase* ***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**
flash	Faz um vídeo reverso muito rápido. Ideal para chamar a atenção sobre erros ou ocorrências interessantes (nem todas as definições de terminais aceitam essa opção)
smcup	Bate uma foto atual da tela e a salva para posterior recuperação
rmcup	Repõe na tela a foto batida com o comando **tput smcup**
setab N	Altera a cor de fundo (**SET B**ackground). **N** varia entre 0 e 7, 9 restaura cor *default*
setaf N	Altera a cor da fonte (**SET F**oreground). **N** varia entre 0 e 7, 9 restaura cor *default*

Vamos fazer um programa bem besta (e portanto fácil) para mostrar alguns atributos deste comando. É o famoso e famigerado "*Alô Mundo*" só que esta frase será escrita no centro da tela e em vídeo reverso e após isso, o cursor voltará para a posição em que estava antes de escrever esta tão criativa frase. Veja:

```
$ cat alo.sh
#!/bin/bash
# Script bobo para testar
# o comando tput (versao 1)

Colunas=`tput cols`            #  Salvando quantidade colunas
Linhas=`tput lines`            #  Salvando quantidade linhas
Linha=$((Linhas / 2))          #  Qual eh a linha do meio da tela?
Coluna=$(((Colunas - 9) / 2)) #  Calculando coluna para centrar na tela
tput sc                        #  Salvando posição do cursor
tput cup $Linha $Coluna        #  Posicionando para escrever
tput rev                       #  Video reverso
echo Alô Mundo
tput sgr0                      #  Restaura video ao normal
tput rc                        #  Restaura cursor aa posição original
```

Como o programa já está todo comentado, acho que a única explicação necessária seria para a linha em que é criada a variável **Coluna** e o estranho ali é aquele número **9**, mas ele é o tamanho da cadeia que pretendo escrever (*Alô Mundo*).

Desta forma este programa somente conseguiria centrar cadeias de 9 caracteres, mas veja isso:

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

Ahhh, melhorou! Então agora relembramos que a construção **${#variavel}** devolve a quantidade de caracteres de variavel (já tínhamos visto isso quando estudamos o comando **expr**).

Assim sendo, vamos otimizar o nosso programa para que ele escreva em vídeo reverso, no centro da tela a cadeia passada como parâmetro e depois o cursor volte à posição que estava antes da execução do script.

```
$ cat alo.sh
#!/bin/bash
# Script bobo para testar
# o comando tput (versao 2)

Colunas=`tput cols`                #  Salvando quantidade colunas
Linhas=`tput lines`                #  Salvando quantidade linhas
Linha=$((Linhas / 2))              #  Qual eh a linha do meio da tela?
Coluna=$(((Colunas - ${#1}) / 2)) #  Centrando a mensagem na tela
tput sc                            #  Salvando posicao do cursor
tput cup $Linha $Coluna            #  Posicionando para escrever
tput rev                           #  Video reverso
echo $1
tput sgr0                          #  Restaura video ao normal
tput rc                            #  Restaura cursor aa posição original
```

Este *script* é igual ao anterior, só que trocamos o valor fixo da versão anterior (**9**), por **${#1}**, onde este **1** é o **$1** ou seja, esta construção devolve o tamanho do primeiro parâmetro passado para o programa. Se o parâmetro que eu quiser passar tiver espaços em branco, teria que colocá-lo todo entre aspas, senão o **$1** seria somente o primeiro pedaço. Para evitar este aborrecimento, é só substituir o **$1** por **$***, que como sabemos é o conjunto de todos os parâmetros. Então aquela linha ficaria assim:

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

e a linha **echo $1** passaria a ser **echo $*.** Mas, quando executar, não esqueça de de passar como parâmetro, a frase que você deseja centrar.

Um outro exemplo, todo comentado, que usa um monte de artifícios do comando **tput** é o que veremos a seguir. Ele serve principalmente para mostrar como se salva uma tela para posterior recuperação. Enfim, uma aula de fotografia de telas ?, veja:

```
$ cat salva_tela
seq 2000 | xargs                          #  Sujando a tela
Lin=$(($(tput lines) / 2))                #  Calculando linha e coluna centrais da tela
Col=$(($(tput  cols) / 2))
tput cup $Lin; tput el; tput setaf 1      #  Posicionando, apagando a linha central e colorindo
echo "Em 10 segundos essa tela será fotografada e se apagará"
tput civis                                # Cursor invisível para melhorar apresentação
```

```
for ((i=10; i!=0; i--))
{
    tput cup $Lin $Col; tput el          # Posiciona no centro da tela e limpa núm anterior
    echo $i
    sleep 1
}
tput smcup                               #  Tirando uma foto da tela
clear                                    #  Poderia ter usado tput reset
tput cup $Lin
echo "Dentro de 10 segundos a tela inicial será recuperada"
for ((i=10; i>0; i--))
{
    tput cup $Lin $Col; tput el          #  Posicionou no centro da tela
    echo $i
    sleep 1
}
tput rmcup                               #  Restaurou a foto
tput cvvis;tput setaf 9                  #  Restaurou o cursor e cor
```

Para que você possa ver todas as combinações de cores de fonte e fundo (**tput setaf** e **tput setab**), execute o *script* a seguir:

```
$ cat colors1.sh
#!/bin/bash
for ((b=0; b<=7; b++))
{
    tput setab  9; tput setaf  9; echo -n "|"
    for ((f=0; f<=7; f++))
    {
        tput setab $b; tput setaf $f; echo -n " b=$b f=$f "
        tput setab  9; tput setaf  9; echo -n "|"
    }
    echo
}
tput setab 9; tput setaf 9
```

Antes eu disse que essas eram todas as combinações, mas se o **tput bold** estiver ativo, você terá mais 7 cores de fonte.

Se estavas sem saco e não quisestes executar esse *script* para ver as cores, seus códigos são os seguintes:

Cores do terminal	
Símbolo	**Cor**
0	Preto
1	Azul
2	Verde
3	Ciano
4	Vermelho
5	Magenta
6	Amarelo
7	Branco
9	Volta a cor *default*

E agora que já sabemos tudo sobre posicionamento e formatação, na próxima aula iremos ver como ler os dados da tela para, aí sim, fazer programas beleza.

13.2. Recebendo dados com o comando read

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

Em uma aula anterior já dei uma dicazinha sobre o comando **read**. Para começarmos a sua analise mais detalhada. Veja só isso:

```
$ read var1 var2 var3
Curso de Shell   # Entrada do read digitada na tela
$ echo $var1
Curso
$ echo $var2
de
$ echo $var3
Shell
$ read var1 var2
Curso de Shell
$ echo $var1
Curso
$ echo $var2
de Shell
```

Como você viu, o **read** recebe uma lista separada por espaços em branco e coloca cada item desta lista em uma variável. Se a quantidade de variáveis for menor que a quantidade de itens, a última variável recebe o restante.

Eu disse lista separada por espaços em branco? Agora que você já conhece tudo sobre a variável **$IFS** (*Inter Field Separator*) que eu te apresentei quando falávamos do comando **for**, será que ainda acredita nisso? Vamos testar direto no *prompt*:

```
$ oIFS="$IFS"
$ IFS=:                 # Salvando o IFS. Como estamos no prompt, temos de restaurá-lo
$ read var1 var2 var3
Curso de Shell          # Entrada do read digitada na tela
$ echo $var1
Curso de Shell
$ echo $var2
$ echo $var3
$ read var1 var2 var3
Curso:de:Shell          # Entrada do read digitada na tela
$ echo $var1
Curso
$ echo $var2
de
$ echo $var3
Shell
$ IFS="$oIFS"           # Restaurando o valor original do IFS
```

Viu, estava furado! O **read** lê uma lista, assim como o **for**, separada pelos caracteres da variável **$IFS.**

Alguns comandos de *Shell* aceitam que algumas variáveis tenham um valor atribuído que vai perdurar somente o tempo de execução desde comando. Dois bons exemplos disso são **$IFS+read** e **$LANG+date,** entre outros. Como mexeremos no **$IFS** no exemplo a seguir, vou agora te mostrar o caso do **$LANG+date:**

```
$ echo $LANG                    # Variável que define o idioma
pt_BR.UTF-8                     # Português do Brasil com acentuação
$ date
Qua Abr  5 17:06:03 BRT 2017    # Data em Português
$ LANG=en_EN date               # LANG teve seu valor alterado
                                # para Inglês para executar o comando date
Wed Apr  5 17:06:31 BRT 2017    # Comando date gerou data em Inglês
$ echo $LANG
pt_BR.UTF-8                     # Ao fim do comando, valor volta ao original
```

Repare que entre a atribuição e o comando, existe um espaço em branco. Se fosse um ponto e vírgula (;), a variável continuaria com o valor atribuído, até que outro valor fosse passado.

Então veja como o **$IFS** e essa dica podem facilitar a sua vida:

```
$ grep julio /etc/passwd
julio:x:500:544:Julio C. Neves - 7070:/home/julio:/bin/bash
$ IFS=: read lname lixo uid gid coment home shell <<< $(grep julio /etc/passwd)
$ echo -e "$lname\n$uid\n$gid\n$coment\n$home\n$shell"
julio
500
544
Julio C. Neves - 7070
/home/julio
/bin/bash
```

Como você viu, a *here string* (**<<<**) redirecionou a saída do **grep** 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 um salto 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 a seguir:

Opções do comando read no Bash	
Opção	**Ação**
-p PROMPT	Escreve o **PROMPT** na tela e fica aguardando você digitar o dado
-n NUM	Lê até **NUM** caracteres
-t SEG	Espera **SEG** segundos para que a leitura seja concluída
-s	O que está sendo teclado não aparece na tela (***s**ilent*)

E agora direto aos exemplos curtos para demonstrar estas opções.

Para ler um campo "**Matrícula**" no formato convencional (UNIX):

```
$ echo -n "Matricula: "; read Mat # Opção -n para não saltar linha
Matricula: 12345
$ echo $Mat
12345
```

Ou simplificando com a opção **-p:**

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

Para ler uma determinada quantidade de caracteres:

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

Neste exemplo fizemos dois **read**: um para a primeira parte do CEP e outra para o seu complemento, deste modo formatando a entrada de dados. O cifrão (**$**) após o último algarismo teclado, é porque o **read** não tem o *newline* implícito por *default* como o tem o **echo.**

Para ler que até um determinado tempo se esgote (conhecido como *timeout*):

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


$
```

Obviamente isto foi uma brincadeira, pois só tinha 3 segundos para digitar o meu nome completo e só me deu tempo de teclar um **J** (aquele colado no **Eta**), mas serviu para mostrar duas coisas:

1. O comando após o par de barras verticais (**||**) (o **ou** lógico, lembra-se?) será executado caso a digitação não tenha sido concluída no tempo estipulado. Isso ocorre porque o código de retorno do **read** voltou algo diferente de zero (neste caso, 142);
2. A variável **Nom** permaneceu vazia. Ela será valorada somente quando o **<ENTER>** for teclado.

Para ler um dado sem ser exibido na tela:

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

Aproveitei um erro para mostrar um macete. Quando escrevi a primeira linha, esqueci de colocar o nome da variável que iria receber a senha, e só notei quando ia listar o seu valor. Felizmente a variável **$REPLY** do Bash, possui a última cadeia lida e me aproveitei disso para não perder a viagem. Teste você mesmo o que acabei de fazer.

Mas o exemplo que dei, era para mostrar que a opção **-s** impede o que está sendo teclado de ir para a tela. Como no exemplo anterior, a falta do *newline* fez com que o *prompt* de comando (**$**) permanecesse na mesma linha (não é que tenha faltado um **<ENTER>**, mas como, devido a opção **-s**, nada aparecia na tela, nem o **<ENTER>** final deu o ar de sua graça).

Bem, agora que sabemos ler da tela, na próxima aula veremos como se lê os dados dos arquivos.

13.2.1. O comando read - Vamos ler arquivos?

Como eu já havia lhe dito, e você deve se lembrar, o **while** testa um comando e executa um bloco de instruções enquanto este comando for bem sucedido. Ora quando você está lendo um arquivo que lhe dá permissão de leitura, o **read** só será mal sucedido quando alcançar o EOF (*End Of File*), desta forma podemos ler um arquivo de duas maneiras:

1. Redirecionando a entrada do arquivo para o bloco do while assim:

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

6. 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:

Vantagens do primeiro processo:

- É mais rápido;
- Não necessita de um *subshell* para assisti-lo;

Desvantagem do primeiro processo:

- Em um bloco de instruções grande, o redirecionamento fica pouco visível o que por vezes prejudica a vizualização do código;

Vantagem do segundo processo:

- Como o nome do arquivo está antes do **while**, é mais fácil a vizualização do código.

Desvantagens do segundo processo:

- O *Pipe* (**|**) chama um *subshell* para interpretá-lo, tornando o processo mais lento, pesado e por vezes problemático (veja o exemplo a seguir).

Para ilustrar o que foi dito, veja estes exemplos a seguir:

```
$ cat frutas
abacate
maçã
morango
pera
tangerina
uva
$ cat lista_frutas.sh
#!/bin/bash
# Exemplo de read passando arquivo de frutas por pipe.

cat frutas |
while read Fruta
do
    echo $((++ContaFruta)) $Fruta  # Incrementa contador e lista-o com cada fruta
done
echo Meu arquivo tem :$ContaFruta: frutas cadastradas
```

Como você pode ver é um programa bem simples que lê cada fruta, incrementando um contador (**$ContaFruta**) e listando-o junto com a fruta correspondente, e ao final, lista a quantidade final apurada.

Vamos então executá-lo:

```
$ lista_frutas.sh
1 abacate
2 maçã
3 morango
4 pera
5 tangerina
6 uva
Meu arquivo tem :: frutas cadastradas
```

Mas tem algum pepino aí! Repare que ao final da execução, o nosso contador (**$ContaFruta**) permanece vazio).

— Ué, será que a variável não foi atualizada?

— Foi sim, e isso pode ser comprovado porque ele apareceu corretamente atualizado em todas as linhas.

— Então porque isso aconteceu?

— Por que como eu disse, o bloco de instruções redirecionado pelo *pipe* (**|**) é executado em um *subshell* e lá as variáveis são atualizadas. Quando este *subshell* termina, as atualizações das variáveis vão para os píncaros do inferno junto com ele. Repare que vou fazer uma pequena mudança nele, passando o arquivo por redirecionamento de entrada (**<**) e as coisas passarão a funcionar na mais perfeita ordem:

```
$ cat lista_frutas.sh
#!/bin/bash
# Exemplo de read passando arquivo de frutas por redirecionamento de entrada.

while read Fruta
do
    echo $((++ContaFruta)) $Fruta  # Incrementa contador e lista-o com cada fruta
done < frutas                  # Redirecionando a entrada do bloco de cmds do while
echo Meu arquivo tem :$ContaFruta: frutas cadastradas
```

E veja a sua perfeita execução:

```
$ lista_frutas.sh
1 abacate
2 maçã
3 morango
4 pera
5 tangerina
6 uva
Meu arquivo tem :6: frutas cadastradas
```

Agora um pequeno e importante macete que vou mostrar utilizando um exemplo prático. Suponha que você queira listar na tela um arquivo e a cada dez registros esta listagem pararia para que o operador pudesse ler o conteúdo da tela e ela só voltasse a rolar (*scroll*) após o operador digitar qualquer tecla. Para não gerar uma tripa vertical que iria de **lin 1** até **lin 30**, mostrei-a na horizontal para que você veja que o arquivo (**numeros**), tem 30 registros com números sequênciais. Veja:

```
$ seq 30 | xargs -i echo lin {} > numeros   #  Criando arquivo numeros
$ cat numeros
lin 1
lin 2
lin 3
...
...
lin 30
$ cat 10porpag.sh
#!/bin/bash
#  Prg de teste para escrever
#+ 10 linhas e parar para ler
#+ Versão 1

while read Num
do
    let ContLin++           # Contando...
    echo -n "$Num "         # -n para nao saltar linha
    ((ContLin % 10 == 0)) && read
done < numeros
```

Na tentativa de fazer um programa genérico criamos a variável **$ContLin** (por que na vida real, os registros não são somente números sequenciais) e parávamos para ler quando o resto da divisão por **10** fosse zero. Porém, quando fui executar deu a seguinte zebra:

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

Repare que faltaram as linhas **lin 11** e **lin 21** e a listagem não parou no **read.** O que houve foi que toda a entrada do *loop* estava redirecionada do arquivo **numeros** e desta forma, a leitura foi feita em cima deste arquivo, desta forma perdendo a **lin 11** (e também a **lin 21** ou qualquer outra linha posterior a um múltiplo de 10).

Vamos mostrar então como deveria ficar para funcionar a contento:

```
$ cat 10porpag.sh
#!/bin/bash
#  Prg de teste para escrever
#+ 10 linhas e parar para ler
#  Versão 2

while read Num
do
    let ContLin++           # Contando...
    echo -n "$Num "         # -n para nao saltar linha
    ((ContLin % 10 == 0)) && read < /dev/tty     # Explicitando que a entrada é pelo teclado
done < numeros
```

Observe que agora a entrada do **read** foi redirecionada por **/dev/tty**, que nada mais é senão o terminal corrente, explicitando desta forma que aquela leitura seria feita do teclado e não de **numeros.** É bom realçar que isto não acontece somente quando usamos o redirecionamento de entrada, se houvéssemos usado o redirecionamento via *pipe* (**|**), o mesmo teria ocorrido.

Veja agora a sua execução:

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

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

```
$ cat 10porpag.sh
#!/bin/bash
#  Prg de teste para escrever
#+ 10 linhas e parar para ler
#+ Versão 3

clear
while read Num
do
    let ContLin++                           # Contando...
    echo "$Num"
    ((ContLin % (`tput lines` - 3) == 0)) &&
        {
        read -n1 -p"Tecle Algo " < /dev/tty # para ler qq caractere
        clear                               # limpa a tela apos leitura
        }
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 3, isto é, se a tela tem 25 linhas, listará 22 registros e parará para leitura. No comando **read** também foi feita uma alteração, inserido um **-n1** para ler somente um caractere sem ser necessariamente um **<ENTER>** e a opção **-p** para dar a mensagem.

Digamos que eu queira listar o primeiro e o terceiro campos de **/etc/passwd** (*user name* e *user id* respectivamente), cujo separador é dois pontos (**:**) contando cada registro listado (esse contador é só para você não fazer **cut -f1,3 -d: /etc/passwd** ?. O fragmento de código seria assim:

```
while IFS=: read Uname Lixo Uid Lixo
do
    echo $((++Cont)) $Uid $Uname
done
```

Essa porcariazinha é simples, porém carregada de macetes:

1. Antes do **read** mudamos a variável **$IFS** que desta forma só será igual a dois pontos (**:**) durante o **read**;
2. O segundo campo e do quarto em diante não me interessavam, por isso usei o mesmo nome de variável para não alocar memória atoa;
3. Após ler **$Uid** o resto não me interessava, mas mesmo assim os li para **$Lixo**, senão iriam ficar juntos ao *user id*. O último leva o resto todo, lembra?

— Bem meu amigo, por hoje é só porque acho que você já está de saco cheio e se você não estiver eu estou... Mas já que você está tão empolgado com o *Shell*, vou te deixar um exercício de aprendizagem para você melhorar a sua CDteca que é bastante simples. Reescreva o seu programa que cadastra CDs para montar toda a tela com um único **echo** e depois vá posicionando à frente de cada campo para receber os valores que serão teclados pelo operador.

14. Funções Internas e Externas

O exercício que passei na última aula é um pouco grande, mas veja minha proposta de resolução:

```
$ cat musinc5
#!/bin/bash
# Cadastra CDs (versao 5)
#
clear
LinhaMesg=$((`tput lines` - 3)) # Linha que msgs serão dadas para operador
TotCols=$(tput cols)            # Qtd colunas da tela para enquadrar msgs
echo "
                                  Inclusao de Músicas
                                  ======== == =======

                    Título do Álbum:
                                         | Este campo, preenchido auto-
                    Faixa:               <  maticamente foi criado somente
                                         | para orientar o preenchimento
                    Nome da Música:

                    Intérprete:"            # Tela montada com um único echo

while true
do
    tput cup 5 38; tput el    # Posiciona e limpa linha
    read Album
    [ ! "$Album" ] &&         # Operador deu <ENTER>
        {
        Msg="Deseja Terminar? (S/n)"
        TamMsg=${#Msg}
        Col=$(((TotCols - TamMsg) / 2))        # Centra msg na linha
        tput cup $LinhaMesg $Col
        echo "$Msg"
        tput cup $LinhaMesg $((Col + TamMsg + 1))
        read -n1 SN
        tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
        [ $SN = "N" -o $SN = "n" ] && continue # $SN é igual a N ou (-o) n?
        clear; exit                            # Fim da execução
        }
    grep "^$Album\^" musicas > /dev/null &&
        {
        Msg="Este álbum já está cadastrado"
        TamMsg=${#Msg}
        Col=$(((TotCols - TamMsg) / 2))       # Centra msg na linha
        tput cup $LinhaMesg $Col
        echo "$Msg"
        read -n1
        tput cup $LinhaMesg $Col; tput el     # Apaga msg da tela
        continue                              # Volta para ler outro álbum
```

```
        }
    Reg="$Album^"                       # $Reg receberá os dados para gravação
    oArtista=                           # Variavel que guarda artista anterior
    while true
    do
        ((Faixa++))
        tput cup 7 38
        echo $Faixa
        tput cup 9 38                   # Posiciona para ler musica
        read Musica
        [ "$Musica" ] ||                # Se o operador tiver dado <ENTER>...
            {
            Msg="Fim de Álbum? (S/n)"
            TamMsg=${#Msg}
            Col=$(((TotCols - TamMsg) / 2))       # Centra msg na linha
            tput cup $LinhaMesg $Col
            echo "$Msg"
            tput cup $LinhaMesg $((Col + TamMsg + 1))
            read -n1 SN
            tput cup $LinhaMesg $Col; tput el    # Apaga msg da tela
            [ "$SN" = N -o "$SN" = n ]&&continue # $SN é igual a N ou (-o) n?
            break                               # Sai do loop para gravar
            }
        tput cup 11 38                          # Posiciona para ler Artista
        [ "$oArtista" ]&& echo -n "($oArtista) " # Artista anterior é default
        read Artista
        [ "$Artista" ] && oArtista="$Artista"
        Reg="$Reg$oArtista~$Musica:"          # Montando registro
        tput cup 9 38; tput el                # Apaga Musica da tela
        tput cup 11 38; tput el               # Apaga Artista da tela
    done
    echo "$Reg" >> musicas                    # Grava registro no fim do arquivo
    sort musicas -o musicas                   # Classifica o arquivo
done
```

É, o programa tá legal, tá todo estruturadinho, mas assim mesmo gostaria de alguns poucos comentários:

Só para relembrar, as seguintes construções:

O **test** feito assim:

```
[ ! $Album ] && CMD
```

ou assim:

```
[ $Musica ] || CMD
```

representam exatamente a mesma coisa, isto é, no caso da primeira, testamos se a variável **$Album** não (**!**) tem nada dentro, então (**&&**) **CMD** será executado. Na segunda, testamos se **$Musica** tem dado, senão (**||**) **CMD** será executado. Estas são formas *booleanas* de usarmos o comando **test**, que creio que dificultam um pouco o entendimento dos iniciantes. Para ficar mais legível prefiro **[-z $Var]** que retornará verdadeiro se **$Var** não existir ou se estiver vazia, o oposto disso seria **[-n $Var]**.

Assim sendo, para facilitar a leitura, poderíamos substituir **[! $Album] &&** por **[-z $Album] &&** (a tendencia é usar **[[-z $Album]] &&**) e **[$Musica] ||** por **[-n $Musica] &&** (idem **[[-n $Musica]] &&**)

Mas, veja só, se ele está grandinho é porque ainda não dei algumas dicas. Repare que a maior parte do *script* é para dar mensagens centradas na penúltima linha da tela. Observe ainda que algumas mensagens pedem um **S** ou um **N** e outras são só de advertência. Seria o caso típico do uso de funções, que seriam escritas somente uma vez e chamada a execução de diversos pontos do *script*. Vou montar duas funções para resolver estes casos e vamos incorporá-las ao seu programa para ver o resultado final.

14.1. Funções Internas

Chamamos de função interna aquela que é declarada dentro do próprio *script* e normalmente, mas não obrigatoriamente, no seu início.

Veja a função **Pergunta** que receberá 3 parâmetros:

1. O texto da pergunta que será feita e será posicionada 3 linhas acima do fim da tela;
2. A resposta que será tratada como valor padrão (*default*);
3. A outra resposta possível.

```
Pergunta ()
{
  #  A função recebe 3 parâmetros na seguinte ordem:
  #+ $1 - Mensagem a ser exibida na tela;
  #+ $2 - Valor a ser aceito com resposta default;
  #+ $3 - O outro valor aceito.
  #  Supondo que $1=Aceita, $2=s e $3=N, a linha a
  #+ seguir colocaria em Msg o valor "Aceita? (S/n)"
  local Msg="$1 ($(tr a-z A-Z <<< $2)/$(tr A-Z a-z <<< $3))? "
  local TamMsg=${#Msg}
  local LinhaMesg=$(($(tput lines) - 3)) # 3 linhas acima do fim da tela
  local TotCols=$(tput cols)
  local Col=$(((TotCols - TamMsg) / 2))  # Centra Msg na linha
  tput sc                                # Salva posição do cursor no corpo do script
  tput cup $LinhaMesg $Col
  read -p "$Msg" -n1 SN
  [ ! $SN ] && SN=$2                     # Se vazia coloca default em SN
  echo $SN | tr A-Z a-z                  # A saída de SN será em minúscula
  tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
  tput rc                                # Restaura cursor para posição em que estava
  return                                 # Sai da função com $? igual a zero
}
```

Como podemos ver, uma função é definida quando fazemos **nome_da_função ()** e todo o seu corpo está entre chaves (**{}**). Assim como quando falamos sobre passagem de parâmetros para *scripts*, as funções os recebem da mesma forma, isto é, são parâmetros posicionais (**$1**, **$2**, ..., **$n**) e todas as regras que se aplicam a passagem de parâmetros para programas, também valem para funções, mas é muito importante realçar que os parâmetros passados para um *script* não se confundem com aqueles que este passou para suas funções. Isso significa, por exemplo, que o **$1** de um *script* é diferente do **$1** de uma de suas funções.

Alias, pode-se dizer que o uso de funções só difere em 2 pontos de um corpo de programa propriamente dito:

1. Variáveis podem ser criadas como local, isto é, têm efeito somente dentro da função. Fora dela seus valores podem ser diferentes. É isso que permite que os parâmetros posicionais da função também possam ser chamados de **$1**, **$2**, ..., **$n**. Eles são do tipo local.
2. Em um programa, podemos fazer algo do tipo:

```
funcao || echo Erro na função funcao
```

Isto é, uma função pode passar um código de retorno (**$?**) que possa ser testado ao seu término. Mas quem passa esse código é o comando **exit**. Quando usado, o comando **exit** encerra o programa e muitas das vezes, queremos dizer que a execução da função não foi bem sucedida, porém desejamos continuar a execução do programa. É exatamente para isso que existe o comando **return**, que passa o código de retorno (entre **0** e **255**), exatamente como o **exit**, porém encerra somente a função e não o programa.

Repare que as variáveis **$Msg**, **$TamMsg** e **$Col** são de uso restrito desta rotina, e por isso foram criadas como **local**. A finalidade disso é simplesmente para economizar memória, já que ao sair da rotina, elas serão devidamente detonadas da partição e caso não tivesse usado este artifício, permaneceriam residentes.

A linha de código que cria **local Msg**, concatena ao texto recebido (**$1**), um abre parênteses, a resposta *default* (**$2**) em caixa alta, uma barra, a outra resposta (**$3**) em caixa baixa e finaliza fechando o parênteses, colocando um ponto de interrogação e um espaço em branco para separar o texto da resposta. Uso esta convenção para, ao mesmo tempo, mostrar as opções disponíveis e realçar a resposta oferecida como *default* em caixa alta.

Quase ao fim da rotina, a resposta recebida (**$SN**) é passada para caixa baixa de forma que no corpo do programa não se precise fazer este teste.

Veja agora como ficaria a função para dar uma mensagem na tela:

```
function MandaMsg
    {
    #  A função recebe somente um parâmetro
    #+ com a mensagem que se deseja exibir,
    #+ para não obrigar ao programador passar
    #+ a msq entre aspas, usaremos $* (todos
    #+ os parâmetro, lembra?) e não $1.
    (( $# >=1 )) || {
        echo "Você esqueceu de passar a msg"
        return 1
    }
    local Msg="$*"
    local TamMsg=${#Msg}
    local Col=$(((TotCols - TamMsg) / 2)) # Centra msg na linha
    tput sc                               # Salva a posição que o cursor está
    tput cup $LinhaMesg $Col
    read -p "$Msg" -n 1
    tput cup $LinhaMesg $Col; tput el     # Apaga msg da tela
    tput rc                               # Devolve o cursor para a
                                          # posição anterior à função
    return                                # Sai da função
    }
```

Esta é uma outra forma de definir uma função: não a chamamos como no exemplo anterior usando uma construção com a sintaxe **`nome_da_função ()`**, mas sim como **`function nome_da_função`**. Quanto ao mais, nada difere da anterior, exceto que, como consta dos comentários, usamos a variável **`$*`** que como já sabemos é o conjunto de todos os parâmetros passados, para que o programador não precise usar aspas envolvendo a mensagem que deseja passar para a função.

Inclui um teste desnecessário (já que só eu usarei esta função) avisando que faltou parâmetro, mas foi só para você ver o uso de um **`return`** com erro (**`1`**), que poderia ser testado no corpo do programa que chamasse esta função.

Como os códigos destas funções são muito semelhantes, as duas funções poderiam ser englobadas em somente uma, desde que se convencionasse que o texto sempre seria passado entre aspas ou apóstrofos (lembre-se que o usuário da função é somente você). Sendo assim, se a função recebesse somente um parâmetro, se procederia como em **`MandaMsg`**, caso contrário com em **`Pergunta`**. Porque você não tenta fazer isso?

Para terminar com este blá-blá-blá vamos ver então as alterações que o programa necessita quando usamos o conceito de funções:

```
$ cat musinc6
#!/bin/bash
# Cadastra CDs (versao 6)
#

# Área de variáveis globais
LinhaMesg=$((`tput lines` - 3))  # Linha que msgs serão passadas para operador
TotCols=$(tput cols)             # Qtd colunas da tela para enquadrar msgs

# Área de funções
Pergunta ()
{
    #  A função recebe 3 parâmetros na seguinte ordem:
    #+ $1 - Mensagem a ser exibida na tela;
    #+ $2 - Valor a ser aceito com resposta default;
    #+ $3 - O outro valor aceito.
    #+ Supondo que $1=Aceita?, $2=s e $3=N, a linha a
    #+ seguir colocaria em Msg o valor "Aceita? (S/n)"
    local Msg="$1 ($(tr a-z A-Z <<< $2)/$(tr A-Z a-z <<< $3))? "
    local TamMsg=${#Msg}
    local LinhaMesg=$(($(tput lines) - 3))
    local TotCols=$(tput cols)
    local Col=$(((TotCols - TamMsg) / 2))  # Centra Msg na linha
    tput sc                                # Salva a posição que o cursor está
    tput cup $LinhaMesg $Col
    read -p "$Msg" -n1 SN
    [ ! $SN ] && SN=$2                     # Se vazia coloca default em SN
    echo $SN | tr A-Z a-z                  # A saída de SN será em minúscula
    tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
    tput rc                                # Restaura curso para posição em que estava
    return                                 # Sai da função
}
```

```
function MandaMsg
    {
    #  A função recebe somente um parâmetro
    #+ com a mensagem que se deseja exibir,
    #+ para não obrigar ao programador passar
    #+ a msq entre aspas, usaremos $* (todos
    #+ os parâmetro, lembra?) e não $1.
    local Msg="$*"
    local TamMsg=${#Msg}
    local Col=$(((TotCols - TamMsg) / 2)) # Centra msg na linha
    tput sc                               # Salva a posição que o cursor está
    tput cup $LinhaMesg $Col
    read -p "$Msg" -n 1
    tput cup $LinhaMesg $Col; tput el # Apaga msg da tela
    tput rc                           # Devolve o cursor para
                                      # a posição anterior à função
    return                            # Sai da função
    }
```

```
# O corpo do programa propriamente dito começa aqui
clear
echo "
                                    Inclusao de Músicas
                                    ======== == =======
                    Título do Álbum:
                                         | Este campo foi
                    Faixa:              <  criado somente para
                                         | orientar o preenchimento
                    Nome da Música:
                    Intérprete:"          # Tela montada com um único echo
while true
do
    tput cup 5 38; tput el                # Posiciona e limpa linha
    read Album
    [ ! "$Album" ] &&                     # Operador deu <ENTER>
        {
        Pergunta "Deseja Terminar" s n
        [ $SN = "n" ] && continue         # Agora só testo a caixa baixa
        clear; exit                       # Fim da execução
        }
    grep -iq "^$Album\^" musicas 2> /dev/null &&
        {
        MandaMsg Este álbum já está cadastrado
        continue                        # Volta para ler outro álbum
        }
    Reg="$Album^"                       # $Reg receberá os dados de gravação
    oArtista=                           # Guardará artista anterior
    while true
    do
        ((Faixa++))
        tput cup 7 38
        echo $Faixa
        tput cup 9 38                   # Posiciona para ler musica
        read Musica
        [ "$Musica" ] ||                # Se o operador tiver dado <ENTER>...
            {
            Pergunta "Fim de Álbum?" s n
            [ "$SN" = n ] && continue      # Agora só testo a caixa baixa
            break                          # Sai do loop para gravar dados
            }
        tput cup 11 38                          # Posiciona para ler Artista
        [ "$oArtista" ]&& echo -n "($oArtista) " # Artista anterior é default
        read Artista
        [ "$Artista" ] && oArtista="$Artista"
        Reg="$Reg$oArtista~$Musica:"              # Montando registro
        tput cup 9 38; tput el                    # Apaga Musica da tela
        tput cup 11 38; tput el                   # Apaga Artista da tela
    done
    echo "$Reg" >> musicas                # Grava registro no fim do arquivo
    sort musicas -o musicas               # Classifica o arquivo
done
```

Repare que a estruturação do *script* está conforme o gráfico a seguir:

Esta estruturação é devido ao *Shell* ser uma linguagem interpretada e desta forma o programa é lido da esquerda para a direita e de cima para baixo e uma variável para ser vista simultaneamente pelo *script* e suas funções deve ser declarada (ou inicializada) antes de qualquer coisa. As funções por sua vez devem ser declaradas antes do corpo do programa propriamente dito porque no ponto em que o programador mencionou seu nome, o interpretador *Shell* já o havia antes localizado e registrado que era uma função.

Uma coisa bacana no uso de funções é fazê-las o mais genéricas possível de forma que elas sirvam para outras aplicações, sem necessidade de serem reescritas. Essas duas que acabamos de ver têm uso generalizado, pois é difícil um *script* que tenha uma entrada de dados pelo teclado que não use uma rotina do tipo da **MandaMsg** ou não interage com o operador por algo semelhante à **Pergunta**.

Conselho de amigo: crie um arquivo e anexe a este arquivo cada função nova que você criar. Ao final de um tempo, você terá uma bela biblioteca de funções que lhe poupará muito tempo de programação, como você verá na próxima aula.

14.2. Funções Externas

A grande maioria das linguagens modernas tem macetes que permitem que você tire proveito de uma rotina desenvolvida para atender circunstâncias de um determinado programa, trazendo uma cópia dela para dentro de outro programa, sem precisar reescrevê-la. Na linguagem C você faz um **#include**, no python precisamos de um **import** e por aí vai... Em *Shell*, para fazer isso, precisamos do **source** (que, como veremos, também é conhecido por **.** (ponto))

14.2.1. O comando source

Vê se você nota algo de diferente na saída do **ls** a seguir:

```
$ ls -la .bash_profile
-rw-r--r--  1 Julio  unknown   4511 Mar 18 17:45 .bash_profile
```

Não olhe a resposta não, volte a prestar atenção! Bem, já que você está mesmo sem saco de pensar e prefere ler a resposta, vou te dar uma dica: acho que você sabe que o **.bash_profile** é um dos programas que são automaticamente "executados" quando você se *loga* (ARRGGHH! Odeio este termo). Agora que te dei esta dica olhe novamente para a saída do **ls** e me diga o que há de diferente nela.

Como eu disse o **.bash_profile** é "executado" em tempo de *logon* e repare que não tem nenhum direito de execução (ele é **-rw-r--r--**). Isso se dá porque se você o executasse como qualquer outro *script* careta, quando terminasse sua execução todo o ambiente por ele gerado morreria junto com o *Shell* sob o qual ele foi executado (você se lembra que todos os *scripts* são executados em subshells, né?).

Pois é. É para coisas assim que existe o comando **source**, também conhecido por **.** (ponto). Este comando faz com que não seja criado um novo *Shell* (um subshell) para executar o programa que lhe é passado como parâmetro.

Melhor um exemplo que 453 palavras. Veja este scriptizinho a seguir:

```
$ cat script_bobo
cd ..
ls
```

Ele simplesmente deveria ir para o diretório acima do diretório atual. Vamos executar uns comandos envolvendo o **script_bobo** e vamos analisar os resultados:

```
$ pwd
/home/jneves
$ script_bobo
jneves    juliana   paula     silvie
$ pwd
/home/jneves
```

Se eu mandei ele subir um diretório, porque não subiu? Subiu sim! O subshell que foi criado para executar o *script*, tanto subiu que listou os diretórios dos quatro usuários abaixo do **/home**, só que assim que o *script* acabou, o subshell foi para o beleleu e com ele todo o ambiente criado. Olha agora como a coisa muda:

```
$ source script_bobo
jneves    juliana   paula     silvie
$ pwd
/home
$ cd -
/home/jneves
$ . script_bobo
jneves    juliana   paula     silvie
$ pwd
/home
```

Ahh! Agora sim! Sendo passado como parâmetro do comando **source** ou **.** (ponto), o *script* foi executado no *Shell* corrente deixando neste, todo o ambiente criado. Agora damos um *rewind* para o início da explicação sobre este comando. Lá falamos do **.bash_profile**, e a esta altura você já deve saber que a sua incumbência é, logo após o *login*, deixar o ambiente de trabalho preparado para o usuário, e agora entendemos que é por isso mesmo que ele é executado usando este artifício.

E agora você deve estar se perguntando se é só para isso que este comando serve, e eu lhe digo que sim, mas isso nos traz um monte de vantagens e uma das mais usadas é tratar funções como rotinas externas. Veja uma outra forma de fazer o nosso programa para incluir CDs no arquivo musicas:

```
$ cat musinc7
#!/bin/bash
# Cadastra CDs (versao 7)
#

# Área de variáveis globais
LinhaMesg=$((`tput lines` - 3))  # Linha que msgs serão dadas para operador
TotCols=$(tput cols)             # Qtd colunas da tela para enquadrar msgs

# O corpo do programa propriamente dito começa aqui
clear
echo "
                                  Inclusao de Músicas
                                  ======== == =======


                    Título do Álbum:
                                        | Este campo foi
                    Faixa:              <  criado somente para
```

```
                                        | orientar o preenchimento
                    Nome da Música:

                    Intérprete:"       # Tela montada com um único echo
while true
do
    tput cup 5 38; tput el              # Posiciona e limpa linha
    read Album
    [ ! "$Album" ] &&                   # Operador deu
        {
        source pergunta.func "Deseja Terminar" s n
        [ $SN = "n" ] && continue       # Agora só testo a caixa baixa
        clear; exit                     # Fim da execução
        }
    grep -iq "^$Album\^" musicas 2> /dev/null &&
        {
        . mandamsg.func Este álbum já está cadastrado
        continue                        # Volta para ler outro álbum
        }
    Reg="$Album^"                       # $Reg receberá os dados de gravação
    oArtista=                           # Guardará artista anterior
    while true
    do
        ((Faixa++))
        tput cup 7 38
        echo $Faixa
        tput cup 9 38                   # Posiciona para ler musica
        read Musica
        [ "$Musica" ] ||                # Se o operador tiver dado ...
            {
            . pergunta.func "Fim de Álbum?" s n
            [ "$SN" = n ] && continue   # Agora só testo a caixa baixa
            break                       # Sai do loop para gravar dados
            }
        tput cup 11 38                          # Posiciona para ler Artista
        [ "$oArtista" ] && echo -n "($oArtista) "  # Artista anter. é default
        read Artista
        [ "$Artista" ] && oArtista="$Artista"
        Reg="$Reg$oArtista~$Musica:"              # Montando registro
        tput cup 9 38; tput el                    # Apaga Musica da tela
        tput cup 11 38; tput el                   # Apaga Artista da tela
    done
    echo "$Reg" >> musicas                # Grava registro no fim do arquivo
    sort musicas -o musicas               # Classifica o arquivo
done
```

Agora o programa deu uma boa encolhida e as chamadas de função foram trocadas por arquivos externos chamados **pergunta.func** e **mandamsg.func**, que assim podem ser chamados por qualquer outro programa, desta forma reutilizando o seu código.

Por motivos meramente didáticos as execuções de **pergunta.func** e **mandamsg.func** estão sendo comandadas por **source** e por **.** (ponto) indiscriminadamente, embora prefira o **source** por ser mais visível desta forma dando maior legibilidade ao código e facilitando sua posterior manutenção.

Veja agora como ficaram estes dois arquivos:

```
$ cat pergunta.func
#  A função recebe 3 parâmetros na seguinte ordem:
#  $1 - Mensagem a ser dada na tela
#  $2 - Valor a ser aceito com resposta default
#  $3 - O outro valor aceito
#  Supondo que $1=Aceita?, $2=s e $3=n, a linha
#  abaixo colocaria em Msg o valor "Aceita? (S/n)"
Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
TamMsg=${#Msg}
Col=$(((TotCols - TamMsg) / 2))     # Centra msg na linha
tput cup $LinhaMesg $Col
echo "$Msg"
tput cup $LinhaMesg $((Col + TamMsg + 1))
read -n1 SN
[ ! $SN ] && SN=$2                  # Se vazia coloca default em SN
echo $SN | tr A-Z a-z               # A saída de SN será em minúscula
tput cup $LinhaMesg $Col; tput el   # Apaga msg da tela

$ cat mandamsg.func
# A função recebe somente um parâmetro
# com a mensagem que se deseja exibir,
# para não obrigar ao programador passar
# a msq entre aspas, usaremos $* (todos
# os parâmetro, lembra?) e não $1.
Msg="$*"
TamMsg=${#Msg}
Col=$(((TotCols - TamMsg) / 2))     # Centra msg na linha
tput cup $LinhaMesg $Col
echo "$Msg"
read -n1
tput cup $LinhaMesg $Col; tput el   # Apaga msg da tela
```

Em ambos os arquivos, fiz somente duas mudanças que veremos nas observações a seguir, porém tenho mais três a fazer:

1. As variáveis não estão sendo mais declaradas como **local**, porque esta é uma diretiva que só pode ser usada no corpo de funções e portanto estas variáveis permanecem no ambiente do *Shell*, poluindo-o;
2. O comando **return** não está mais presente mas poderia estar sem alterar em nada a lógica, uma vez que ele só serviria para indicar um eventual erro via um código de retorno previamente estabelecido (por exemplo **return 1**, **return 2**, ...), sendo que o **return** e **return 0** são idênticos e significam rotina executada sem erros;
3. O comando que estamos acostumados a usar para gerar código de retorno é o **exit**, mas a saída de uma rotina externa não pode ser feita desta forma, porque por estar sendo executada no mesmo *Shell* que o *script* chamador, o **exit** simplesmente encerraria este *Shell*, terminando a execução de todo o *script*;
4. De onde surgiu a variável **LinhaMesg**? Ela veio do **musinc7**, porque ela havia sido declarada antes da chamada das rotinas (nunca esquecendo que o *Shell* que está interpretando o *script* e estas rotinas é o mesmo);
5. Se você decidir usar rotinas externas, não se avexe, abunde os comentários (principalmente sobre a passagem dos parâmetros) para facilitar a manutenção e seu uso por outros programas no futuro.

Essa teoria é muito bonitinha, mas no duro, o que se usa é um arquivo, digamos: **func.sh** no qual **Pergunta**, **MandaMsg** e todas as funçoes que se usa habitualmente, estão declaradas como função, desta forma, usando **local**, **return**, ... enfim, tudo que tem direito.

No início de um aplicativo que você precise usar, digamos, **Pergunta** deveria ser inserida a seguinte linha:

```
source  func.sh    # Ou . func.sh (esse foi o nome que inventamos)
```

E, quando necessário, você mandaria:

```
Pergunta "Deseja continuar" s n
```

Bem, agora você já tem mais um monte de novidade para melhorar os *scripts* que fizemos você certamente se lembra do programa **listartista** no qual você passava o nome de um artista como parâmetro e ele devolvia as suas músicas mas, lembrando ou não, ele era assim:

```
$ cat listartista
#!/bin/bash
# Dado um artista, mostra as suas musicas
# versao 2

if  [ $# -eq 0 ]
then
    echo Voce deveria ter passado pelo menos um parametro
    exit 1
fi

IFS="
:"
for ArtMus in $(cut -f2 -d^ musicas)
do
    echo "$ArtMus" | grep -i "^$*~" > /dev/null && echo $ArtMus | cut -f2 -d~
done
```

Então para firmar os conceitos que passei, faça-o com a tela formatada, em *loop*, de forma que ele só termine quando receber um **<ENTER>** puro no nome do artista. Ahhh! Quando a listagem atingir a antepenúltima linha da tela, o programa deverá dar uma parada para que o operador possa lê-las, isto é, suponha que a tela tenha 25 linhas. A cada 22 músicas listadas (quantidade de linhas menos 3) o programa aguardará que o operador tecle algo para então prosseguir. Eventuais mensagens de erro devem ser passadas usando a rotina **mandamsg.func** que acabamos de desenvolver.

15. Escrevendo bonitinho (ou formatando a saída)

Sumário

15. Escrevendo bonitinho (ou formatando a saída)

- 15.1. Envenenando a escrita
- 15.2. Caracteres para especificação de formato
- 15.3. Sequencias de "escape" no padrão da linguagem C
- 15.4. Exemplos

15. Escrevendo bonitinho (ou formatando a saída)

Na aula "Usando a tela - Formatando com o comando tput" vimos como o **tput** pode formatar a entrada para leitura de dados. Agora vamos analisar o exercício proposto e em seguida falar sobre formatação da saída com o comando **printf**. Na última aula, eu te pedi para refazer o **listartista** com a tela formatada, em *loop*, de forma que ele só termine quando receber um **<ENTER>** puro, isto é, sem conteúdo, no nome do artista. Eventuais mensagens de erros e perguntas deveriam ser dadas na antepenúltima linha da tela utilizando as rotinas **mandamsg.func** e **pergunta.func** que tínhamos acabado de desenvolver.

Primeiramente eu dei uma encolhida no **mandamsg.func** e no **pergunta.func**, que ficaram assim:

```
$ cat mandamsg.func
# A  função recebe  somente um parâmetro
# com a  mensagem que  se deseja exibir,
# para não obrigar ao programador passar
# a msg  entre aspas, usaremos $* (todos
# os parâmetros, lembra?) e não $1.
Msg="$*"
TamMsg=${#Msg}
Col=$(((TotCols - TamMsg) / 2))     # Centra msg na linha
tput cup $LinhaMesg $Col
read -n1 -p "$Msg "

$ cat pergunta.func
#  A função recebe 3 parâmetros na seguinte ordem:
#  $1 - Mensagem a ser dada na tela
#  $2 - Valor a ser aceito com resposta default
#  $3 - O outro valor aceito
#  Supondo que $1=Aceita?, $2=s e $3=n, a linha
#  abaixo colocaria em Msg o valor "Aceita? (S/n)"
Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
TamMsg=${#Msg}
Col=$(((TotCols - TamMsg) / 2))     # Centra msg na linha
tput cup $LinhaMesg $Col
read -n1 -p "$Msg " SN
[ ! $SN ] && SN=$2                  # Se vazia coloca default em SN
SN=$(echo $SN | tr A-Z a-z)         # A saída de SN será em minúscula
tput cup $LinhaMesg $Col; tput el   # Apaga msg da tela
```

E agora aí vai o grandão:

```
$ cat listartista3
#!/bin/bash
# Dado um artista, mostra as suas músicas
# versao 3

LinhaMesg=$((`tput lines` - 3))  # Linha que msgs serão dadas para operador
TotCols=$(tput cols)             # Qtd colunas da tela para enquadrar msgs

clear
echo "
                              +--------------------------------------------------+
                              |  Lista Todas as Músicas de um Determinado Artista  |
                              |  ----- ----- -- ------- -- -- ----------- -------  |
                              |                                                  |
                              |  Informe o Artista:                              |
                              +--------------------------------------------------+"
while true
do
    tput cup 5 51; tput ech 31  # ech=Erase chars (31 caracteres para não apagar barra vertical)
    read Nome
    if  [ ! "$Nome" ]           # $Nome estah vazio?
    then
        . pergunta.func "Deseja Sair?" s n
        [ $SN = n ] && continue
        break
    fi

    fgrep -iq "^$Nome~" musicas || # fgrep não interpreta ^ como expressão regular
        {
        . mandamsg.func "Não existe música deste artista"
        continue
        }

    tput cup 7 29; echo '|                                                    |'
 LinAtual=8
    IFS="
:"
    for ArtMus in $(cut -f2 -d^ musicas) # Exclui nome do album
    do
        if  echo "$ArtMus" | grep -iq "^$Nome~"
        then
            tput cup $LinAtual 29
            echo -n '|  '
            echo $ArtMus | cut -f2 -d~
            tput cup $LinAtual 82
            echo '|'
            let LinAtual++
            if  [ $LinAtual -eq $LinhaMesg ]
            then
                . mandamsg.func "Tecle Algo para Continuar..."
                tput cup 7 0; tput ed  # Apaga a tela a partir da linha 7
                tput cup 7 29; echo '|                                                |'
                LinAtual=8   ========================================
            fi
        fi
    done
    tput cup $LinAtual 29; echo '|                                                  |'
    tput cup $((++LinAtual)) 29
    read -n1 -p "+-----------Tecle Algo para Nova Consulta------------+"
    tput cup 7 0; tput ed              # Apaga a tela a partir da linha 7
done
```

Assim foi mais trabalhoso mas a apresentação ficou legal exploramos bastante as opções do **tput**. Vamos testar o resultado com um álbum do *Emerson, Lake & Palmer* que tenho cadastrado:

```
+-----------------------------------------------------+
|  Lista Todas as Músicas de um Determinado Artista   |
|  ----- ----- -- ------- -- -- ----------- -------   |
|                                                     |
|  Informe o Artista: Emerson, Lake & Palmer          |
+-----------------------------------------------------+
|                                                     |
|  Jerusalem                                          |
|  Toccata                                            |
|  Still ... You Turn Me On                           |
|  Benny The Bouncer                                  |
|  Karn Evil 9                                        |
|                                                     |
+-----------Tecle Algo para Nova Consulta------------+
```

15.1. Envenenando a escrita

Pronto, finalmente poderemos ver como se envenena a escrita. Agora você já sabe tudo sobre leitura, mas sobre escrita está apenas engatinhando. Já sei que você vai me perguntar:

— Ora, não é com o comando **echo** e com os redirecionamentos de saída que se escreve? É, com estes comandos você escreve 90% das coisas necessárias, porém se precisar de escrever algo formatado eles lhe darão muito trabalho e, mesmo assim, ficará uma coisa mal acabada. Para formatar a saída veremos agora uma instrução muito interessante e que é de execução muito veloz, porque, assim como o **echo**, é um comando intrínseco (*builtin*) do *Shell* - é o **printf**. Sua sintaxe é a seguinte:

```
printf [-v VAR] FMT [ARG...]
```

Onde:

- **FMT** - é uma cadeia de caracteres que contem 3 tipos de objeto:
  1. caracteres simples;
  2. caracteres para especificação de formato;
  3. sequência de *escape* no padrão da linguagem C.
- **ARG** - é a cadeia a ser impressa sob o controle do formato **FMT**.
- **VAR** - é o nome da variável que armazenará a saída do comando, caso a opção **-v** seja usada

15.2. Caracteres para especificação de formato

Cada um dos caracteres utilizados para especificação de formato é precedido pelo caractere **%**, de acordo com a tabela:

Caracteres para especificação de formato	
Letra	**O argumento (ARG) será impresso como**
c	Simples caractere
d	Número no sistema decimal
e	Notação científica exponencial
f	Número com ponto decimal (*float*)
g	O menor entre os formatos **%e** e **%f** com supressão dos zeros não significativos
o	Número no sistema octal
s	Cadeia de caracteres
x	Número no sistema hexadecimal [0-9A-Z]
%	Imprime um **%**. Não existe nenhuma conversão

15.3. Sequencias de "escape" no padrão da linguagem C

As sequências de *escape* padrão da linguagem C são sempre precedidas por uma contra-barra (****) e as que são reconhecidas pelo comando **printf** são:

Sequencias de *escape* no padrão da linguagem C	
Sequência	**Efeito**
a	Soa o *beep*
b	Volta uma posição (*backspace*)
f	Salta para a próxima página lógica (*form feed*)
n	Salta para o início da linha seguinte (*line feed*)
r	Volta para o início da linha corrente (*carriage return*)
t	Avança para a próxima marca de tabulação

15.4. Exemplos

Não acabou por aí não! Tem muito mais coisa sobre a instrução, mas como é muito cheio de detalhes e, portanto, chato para explicar e, pior ainda para ler ou estudar, vamos passar direto aos exemplos com seus comentários, que não estou aqui para encher o saco de ninguém.

```
$ printf "%c" "1 caracter"
1$              # Errado! Só listou 1 caractere e não saltou linha ao final
$ printf "%c\n" "1 caracter"
1               # Saltou linha mas ainda não listou a cadeia inteira
$ printf "%c caractere\n" 1
1 caractere     # Esta é a forma correta o %c recebeu o 1
$ a=2
$ printf "%c caracteres\n" $a
2 caracteres    # O %c recebeu o valor da variável $a
$ printf "%10c caracteres\n" $a
2 caracteres
$ printf "%10c\n" $a caracteres
         2
         c
```

Repare que nos dois últimos exemplos, em virtude do **%c**, só foi listado um caractere de cada cadeia. O **10** à frente do **c**, não significa 10 caracteres. Um número seguindo o sinal de percentagem (**%**) significa o tamanho que a cadeia terá após a execução do comando.

E tome de exemplo:

```
$ printf "%d\n" 32
32
$ printf "%10d\n" 32
        32              # Preenche com 8 brancos à esquerda e não com zeros
$ printf "%04d\n" 32
0032                    # 04 após % significa 4 dígitos com zeros à esquerda
$ printf "%e\n" $(echo "scale=2 ; 100/6" | bc)
1.666000e+01            # O default do %e é 6 decimais
$ printf "%.2e\n" `echo "scale=2 ; 100/6" | bc`
1.67e+01                # O .2 especificou duas decimais
$ printf "%f\n" 32.3
32.300000               # O default do %f é 6 decimais
$ printf "%.2f\n" 32.3
32.30                   # O .2 especificou duas decimais
$ printf "%.3f\n" `echo "scale=2 ; 100/6" | bc`
33.330                  # O bc devolveu 2 decimais. o printf colocou 0 à direita
```

Observação sobre os números reais (decimais) gerados pelo **bc** nos exemplos acima: O **bc** sempre usará um ponto (**.**) como separador de decimais e em alguns sistemas o **printf** só reconhece a vírgula neste papel. Para evitar erros, prefacie o **printf** com a cadeia **LC_ALL=C** que ambos os comandos serão executados no local *default* (**LC** é derivado de *locale*, **ALL** é tudo e **C** é o padrão)

Veja só:

```
$ printf "%e\n" $(echo "scale=2 ; 100/6" | bc)
bash: printf: 16.66: número inválido
0,000000e+00
$ LC_ALL=C printf "%e\n" $(echo "scale=2 ; 100/6" | bc)
1.666000e+01
```

Continuando com os exemplos:

```
$ printf "%o\n" 10
12                      # Converteu o 10 para octal
$ printf "%03o\n" 27
033                     # Assim a conversão fica com mais jeito de octal, né?
$ printf "%s\n" Peteleca
Peteleca
$ printf "%15s\n" Peteleca
       Peteleca         # Peteleca + brancos à esquerda com total de 15 caracteres
$ printf "%-15sNeves\n" Peteleca
Peteleca       Neves    # O menos (-) encheu com brancos à direita de Peteleca
$ printf "%.3s\n" Peteleca
Pet                     # 3 trunca as 3 primeiras
$ printf "%10.3sa\n" Peteleca

       Peta             # Pet com 10 caracteres concatenado com a (após o s)

$ printf "EXEMPLO %x\n" 45232
EXEMPLO b0b0            # Transformou para hexa mas os zeros não combinam
$ printf "EXEMPLO %X\n" 45232
EXEMPLO B0B0            # Assim disfarçou melhor (repare o X maiúsculo)
$ printf "%X %XL%X\n" 49354 192 10
C0CA C0LA
```

O último exemplo não é marketing e é bastante completo, vou comentá-lo passo-a-passo:

O primeiro **%X** converteu **49354** em hexadecimal resultando **C0CA** (leia-se "cê", "zero", "cê" e "a");

Em seguida veio um espaço em branco seguido por outro **%XL**. O **%X** converteu o **192** dando como resultado **C0** que com o **L** fez **C0L;**

E finalmente o último **%X** transformou o **10** em **A**.

Olha que interessante esse caso: quero traçar uma linha com 20 caracteres. Então eu faço:

```
$ printf "%20s" | tr ' ' -   # Não saltei linha (\n), então prompt sairá colado
--------------------$
```

Como disse para o **printf** que a saída teria 20 caracteres e não especifiquei qual, o comando mandou 20 espaços em branco para o **tr**, que os substituiu por traços (**-**).

Até aí tudo bem, mas e se eu quisesse passar uma linha tracejada de ponta a ponta do terminal? Simples, muda somente a quantidade de caracteres. Uma linha tem **tput cols** colunas, então basta trocar 20 por este comando, priorizando sua execução. Faça assim:

```
$ printf "%$(tput cols)s" ' ' | tr ' ' -
--------------------------------------------------------------------------------
```

Vamos mudar um pouco agora. Vamos fazer uma linha que vá de ponta a ponta do terminal, mas agora eu quero que ela seja cheia, e não tracejada. Faça assim:

```
$ printf  "\e(0\x71\e(B"
─
```

Isso não é um traço nem um sublinha, é uma sequência de *escape* que configura um segmento semigráfico de reta. Então você fica imaginando que bastaria ir no **tr**, e trocar o traço (**-**) por **\e(0\x71\e(B**, mas a coisa não é bem assim, porque o **tr** troca um caractere por outro e esta sequência de *escape* tem um monte de caracteres. Então vamos fazer assim:

```
$ printf -v traco "\e(0\x71\e(B"
$ echo $traco
─
```

A opção **-v traco** jogou a saída para a variável **traco** e agora podemos fazer o **tr**, já que a troca será um para um. Assim:

```
$ printf "%$(tput cols)s" ' ' | tr ' ' $traco
```

Epa! Não é bem assim! Você viu que caracteres estranhos o **tr** mandou para a saída? Ele não entendeu o UTF-8... Então só nos resta trocar o **tr** por um **sed**, que nunca nos deixa na mão.

```
$ printf "%$(tput cols)s" ' ' | sed "s/ /$traco/g"
─────────────────────────────────────────────────────────────────────────────
```

Mas agora, se você quiser fazer em uma só linha, como tudo em *Shell*, basta fazer:

```
 $ printf "%$(tput cols)s" ' ' | sed "s/ /$(printf '\e(0\x71\e(B')/g"
```

Conforme vocês podem notar, a instrução **printf** é bastante completa e aconselho a todos que a usem bastante para gravar na cabeça esses caracteres de formatação, porque são muito úteis.

Creio que quando resolvi explicar o **printf** através de exemplos, acertei em cheio pois não saberia como enumerar tantas regrinhas sem tornar a leitura enfadonha.

Sumário

16. Variáveis e parâmetros

- 16.1. Principais Variáveis do Shell
- 16.2. Expansão de Parâmetros
- 16.3. Vetores ou Arrays
  - 16.3.1. Um pouco de manipulação de vetores
  - 16.3.2. Vetores associativos
  - 16.3.3. Lendo um arquivo para um vetor (comando mapfile)

16. Variáveis e parâmetros

Agora começaremos a ver os principais conceitos aplicados às variáveis no *Shell* com ênfase em:

1. Variáveis do *Shell*;
2. Expansão de parâmetros e
3. Tratamento de Vetores (*Arrays*);

16.1. Principais Variáveis do Shell

O **Bash** possui diversas variáveis que servem para dar informações sobre o ambiente ou alterá-lo. Seu número é muito grande e não pretendo mostrar todas, mas uma pequena parte, abordando aquelas podem lhe ajudar na elaboração de *scripts*. Então aí vão as principais:

Principais variáveis do Bash	
Variável	**Conteúdo**
CDPATH	Contém os caminhos que serão pesquisados para tentar localizar um diretório especificado. Apesar desta variável ser pouco conhecida, seu uso deve ser incentivado por poupar muito trabalho, principalmente em instalações com estrutura de diretórios com muitos níveis.
HISTSIZE	Limita o número de instruções que cabem dentro do arquivo de histórico de comandos (normalmente **.bash_history** mas efetivamente é o que está armazenado na variável **$HISTFILE**). Seu valor *default* é 500.
HOSTNAME	O nome do *host* corrente (que também pode ser obtido com o comando **uname -n**).
LANG	Usada para determinar a língua falada no país (mais especificamente categoria do *locale*).
LINENO	O número da linha do *script* ou da função que está sendo executada, seu uso principal é para dar mensagens de erro acompanhado das variáveis **$0** (nome do programa) e **$FUNCNAME** (nome da função em execução)
LOGNAME	Armazena o nome de *login* do usuário.
MAILCHECK	Especifica, em segundos, a frequência que o *Shell* verificará a presença de correspondências nos arquivos indicados pelas variáveis **$MAILPATH** ou **$MAIL**. O tempo padrão é 60 segundos. Uma vez este tempo expirado, o *Shell* fará esta verificação antes de exibir o próximo *prompt* primário (definido em **$PS1**). Se esta variável estiver sem valor ou com um valor menor ou igual a zero, a verificação de novas correspondências não será efetuada.
PATH	Caminhos que serão pesquisados para tentar localizar um arquivo especificado. Como cada *script* é um arquivo, caso use o diretório corrente (**.**) na sua variável **$PATH**, você não necessitará de usar o **./scrp** para que **scrp** seja executado. Basta fazer **scrp**.
PIPESTATUS	É uma variável do tipo vetor (*array*) que contém uma lista de valores de código de retorno do último *pipeline* executado, isto é, um *array* que abriga cada um dos **$?** de cada instrução do último *pipeline*.
PROMPT_COMMAND	Se esta variável receber uma instrução, toda vez que você der um **<ENTER>** direto no *prompt* principal (**$PS1**), este comando será executado. É útil quando se está repetindo muito uma determinada instrução.
PS1	É o *prompt* principal. Aqui usamos os seus *defaults*: **$** para usuário comum e **#** para **root**, mas é muito frequente que ele esteja customizado. Uma curiosidade é que existe até concurso de quem programa o **$PS1** mais criativo. (clique para dar uma googlada).
PS2	Também chamado *prompt* de continuação, é aquele sinal de maior (**>**) que aparece após um **<ENTER>** sem o comando ter sido encerrado.
PWD	Possui o caminho completo (**$PATH**) do diretório corrente. Tem o mesmo efeito do comando **pwd**.
RANDOM	Cada vez que esta variável é acessada, devolve um número inteiro, que é um randômico entre 0 e 32767.
REPLY	Use esta variável para recuperar o último campo lido, caso ele não tenha nenhuma variável associada.
SECONDS	Esta variável contém a quantidade de segundos que o *Shell* corrente está de pé. Use-a somente para esnobar usuários daquilo que chamam de sistema operacional, mas necessita de *boots* frequentes. ?
TMOUT	Se tiver um valor maior do que zero, este valor será tomado como o padrão de *timeout* do comando **read**. No *prompt*, este valor é interpretado como o tempo de espera por uma ação antes de expirar a sessão. Supondo que a variável contenha 30, o *Shell* dará *logout* após 30 segundos de *prompt* sem nenhuma ação.

Outro vetor do sistema muito importante para os programadores é o **BASH_REMATCH**, que vimos nas explicações do comando **test**.

- CDPATH

```
$ echo $CDPATH
.:..:~:/usr/local
$ pwd
/home/jneves/LM
$ cd bin
$ pwd
/usr/local/bin
```

Como **/usr/local** estava na minha variável **$CDPATH**, e não existia o diretório **bin** em nenhum dos seus antecessores (**.**, **..** e **~**), o **cd** foi executado para **/usr/local/bin**.

- LANG

```
$ date
Thu Apr 14 11:54:13 BRT 2017
$ LANG=pt_BR date
Qui Abr 14 11:55:14 BRT 2017
```

Com a especificação da variável **LANG=pt_BR** (português do Brasil), a data passou a ser informada no padrão brasileiro. É interessante observarmos que não foi usado ponto-e-vírgula (**;**) para separar a atribuição de **LANG** do comando **date**, desta forma alterando o valor da variável somente durante a execução do comando.

- PIPESTATUS

```
$ who
jneves   pts/0        Apr 11 16:26 (10.2.4.144)
jneves   pts/1        Apr 12 12:04 (10.2.4.144)

$ who | grep ^botelho
$ echo ${PIPESTATUS[*]}
0 1
```

Neste exemplo mostramos que o usuário **botelho** não estava "logado", em seguida executamos um *pipeline* que procurava por ele. Usa-se a notação **[*]** em um *array* para listar todos os seus elementos, e desta forma vimos que a primeira instrução (**who**) foi bem sucedida (código de retorno 0) e a seguinte (**grep**), não (código de retorno 1).

- **RANDOM**

Para gerar randomicamente um inteiro entre 0 e 100, fazemos:

```
$ echo $((RANDOM%101))
73
```

Ou seja, pegamos o resto da divisão por 101 do número randômico gerado, porque o resto da divisão de qualquer número por 101 varia entre 0 e 100.

Usando o mesmo raciocínio, para gerar entre 5 e 100, manda-se gerar até 96, ou seja de 0 a 95 e depois soma-se 5 a este randômico gerado:

```
$ echo $((RANDOM%96+5))
49
```

- **REPLY**

```
$ read -p "Digite S ou N: "
Digite S ou N: N
$ echo $REPLY
N
```

Eu sou do tempo que memória era um bem precioso que custava muuuuito caro. Então para pegar um **S** ou um **N**, não costumo alocar um espaço especial e assim sendo, pego o que foi digitado na variável **$REPLY**.

16.2. Expansão de Parâmetros

Bem, muito do que vimos até agora são comandos externos ao *Shell*. Eles quebram o maior galho, facilitam a visualização, manutenção e depuração do código, mas não são tão eficientes quanto os intrínsecos (*built-ins*). Quando o nosso problema for performance, devemos dar preferência ao uso dos intrínsecos e a partir de agora vou te mostrar algumas técnicas para o teu programa pisar no acelerador.

Na tabela e exemplos a seguir, veremos uma série de construções chamadas **Expansão** (ou Substituição) **de Parâmetros** (*Parameter Expansion* ou *Parameter Substitution*), que substituem instruções como o **cut**, o **expr**, o **tr**, o **sed** e outras, de forma mais ágil. Já tínhamos até visto alguns como:

1. **${NUM}** - Para acessar parâmetros de ordem superior a 9. Se fizermos: **$ echo $13**, será listado o 1° parâmetro, com um 3 colado. Para listar o 13° parâmetro é necessário que se faça **echo ${13}**. Em função disso é que ele foi batizado de Expansão de Parâmetros pois o uso desta técnica permitiu que se endereçasse de forma direta, mais de 9 parâmetros.
2. **${#VAR}** - Devolve tamanho de **$VAR**
3. **${VAR:POS:TAM}** - Extrai de **$VAR** desde a posição **$POS** (origem zero) com tamanho **$TAM**
4. **${VAR:POS}** - Extrai de **$VAR** a partir de **POS**. Origem zero

Expansão de parâmetros	
Expressão	**Resultado esperado**
${VAR:-CAD}	Se **$VAR** não tem valor, o resultado da expressão é a cadeia **CAD**
${VAR:+CAD}	Se **$VAR** tem valor, o resultado da expressão é a cadeia **CAD**
${#VAR}	Tamanho de **$VAR**
${VAR:POS}	Extrai de **$VAR** a partir de **POS**. Origem zero
${VAR:POS:TAM}	Extrai de **$VAR** a partir de **POS** com tamanho igual a **TAM**. Origem zero
${VAR#PADR}	Corta a menor ocorrência de **$VAR** à esquerda da expressão que case com o padrão **PADR**, sendo este formado por metacaracteres de expansão de arquivos
${VAR##PADR}	Corta a maior ocorrência de **$VAR** à esquerda da expressão que case com o padrão **PADR**, sendo este formado por metacaracteres de expansão de arquivos
${VAR%PADR}	Corta a menor ocorrência de **$VAR** à direita da expressão que case com o padrão **PADR**, sendo este formado por metacaracteres de expansão de arquivos
${VAR%%PADR}	Corta a maior ocorrência de **$VAR** à direita da expressão que case com o padrão **PADR**, sendo este formado por metacaracteres de expansão de arquivos
${VAR/PADR/CAD}	Troca em **$VAR** a primeira ocorrência de **PADR** por **CAD**. O padrão **PADR** pode ser formado por metacaracteres de expansão de arquivos
${VAR//PADR/CAD}	Troca em **$VAR** todas as ocorrências de **PADR** por **CAD**. O padrão **PADR** pode ser formado por metacaracteres de expansão de arquivos
${VAR/#PADR/CAD}	Se **PADR** combina com o início de **$VAR**, então é trocado por **CAD**. O padrão **PADR** pode ser formado por metacaracteres de expansão de arquivos
${VAR/%PADR/CAD}	Se **PADR** combina com o fim de **$VAR**, então é trocado por **CAD**. O padrão **PADR** pode ser formado por metacaracteres de expansão de arquivos
${!VAR}	Lista valor da variável apontada por **$VAR** (indireção)
${VAR^}	Coloca a primeira letra de **$VAR** em maiúscula
${VAR^^}	Coloca todas as letras de **$VAR** em maiúscula
${VAR,}	Coloca a primeira letra de **$VAR** em minúscula
${VAR,,}	Coloca todas as letras de **$VAR** em minúscula
${VAR~}	Troca a caixa da primeira letra de **$VAR**
${VAR~~}	Inverte a caixa de todas as letras de **$VAR**

- **Para aceitar valores padrão (*default*)**

  Se em uma pergunta o **S** é oferecido como valor *default* (padrão) e a saída vai para a variável **$SN**, após ler o valor podemos fazer:

```
SN=${SN:-S}
```

  Desta forma se o operador deu um simples **<ENTER>** para confirmar que aceitou o valor *default*, após executar esta instrução, a variável terá o valor **S**, caso contrário, terá o valor digitado.

  Mais um exemplo dessa expansão de parâmetros. Suponha que o seu *script* vá ler o nome do usuário para fazer *login* em uma máquina remota. Você, querendo facilitar a vida do cara, oferece como valor *default* o *login name* dele na máquina local, que está contido na variável **$LOGNAME** e que tem grande chance de ser o mesmo da remota.

  Essa leitura poderia ser feita conforme o fragmento de código a seguir:

```
read -p "Login na máquina remota ($LOGNAME): " ln
# $LOGNAME sendo oferecido como default
ln=${ln:-$LOGNAME}
```

  Caso a variável **$ln** esteja vazia, ao executar a expansão de parâmetros o valor do *login name* remoto receberá o mesmo valor do *login* local.

- **O oposto disso, mas também interessante**

```
$ Var1=10; unset Var2; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10
$ Var1=10; Var2=; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10
$ Var1=10; Var2=20; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10 e Var2 tem 20
```

  Como você pôde ver, enquanto inexistiu **$Var2** ou seu valor foi nulo, não houve a expansão do parâmetro. Isso aconteceu somente após a variável ser valorada.

- **Para sabermos o tamanho de uma cadeia:**

```
$ cadeia=0123
$ echo ${#cadeia}
4
```

- **Para extrair de uma cadeia da posição um até o final fazemos:**

```
$ cadeia=abcdef
$ echo ${cadeia:1}
bcdef
```

  Repare que a origem é zero e não um.

- **Na mesma variável `$cadeia` do exemplo acima, para extrair 3 caracteres a partir da posição 2:**

```
$ echo ${cadeia:2:3}
cde
```

Repare que novamente que a origem da contagem é zero e não um.

- **Podemos também extrair do fim para o princípio**

```
$ TimeBom=Flamengo
$ echo ${TimeBom: -5}
mengo
$ echo ${TimeBom:(-5)}
mengo
```

Nessa extração invertida, o espaço em branco ou os parênteses são obrigatórios para que o sinal de menos (`-`) não cole nos dois-pontos (`:`). Caso isso ocorresse, ficaria semelhante à expansão que vimos há pouco e que substitui uma variável vazia ou nula por um valor padrão (*default*), ou seja, a famosa `${VAR:-PADR}`. Veja:

```
$ echo ${TimeBom:-5}
Flamengo
```

- **Para suprimir tudo à esquerda da primeira ocorrência de uma cadeia**

```
$ cadeia="Casa da Mãe Joana"
$ echo ${cadeia#*' '}
da Mãe Joana
$ echo Morada ${cadeia#*' '}
Morada da Mãe Joana
```

Neste exemplo foi suprimido à esquerda tudo que casasse com a menor ocorrência da expressão `*' '`, ou seja, tudo até o primeiro espaço em branco.

Estes exemplos também poderiam ser escritos sem protegermos o espaço da interpretação do *Shell* (mas prefiro protegê-lo para facilitar a legibilidade do código), veja:

```
$ echo ${cadeia#* }
da Mãe Joana
$ echo Espelunca ${cadeia#* }
Espelunca da Mãe Joana
```

Repare que conforme descrevemos na tabela resumo das expansões de parâmetros o padrão representado por `PADR` permite o uso de metacaracteres de expansão de arquivos (`*`, `?`, `[...]` e `[!...]`).

- **Para suprimir tudo à esquerda da última ocorrência de uma cadeia**

  Utilizando o mesmo valor da variável **$cadeia**, observe como faríamos para termos somente **Joana**:

```
$ echo ${cadeia##*' '}
Joana
$ echo Vou levar um papo com a ${cadeia##*' '}
Vou levar um papo com a Joana
```

  Desta vez suprimimos à esquerda de **$cadeia** a maior ocorrência do padrão **PADR**. Assim como no caso anterior, o uso de metacaracteres é permitido.

  Outro exemplo mais útil: para que não apareça o caminho (*path*) completo do seu programa (que, como já sabemos está contido na variável **$0**) em uma mensagem de erro, inicie o seu texto da seguinte forma:

```
echo Uso: ${0##*/} texto da mensagem de erro
```

  Onde o zero (**0**) é a variável que tem o caminho (relativo ou absoluto) de seu programa. Assim sendo, o que nos interessa é apagar à esquerda o maior casamento com o padrão ***/**, isto é, tudo (*****) até a última barra (**/**), sobrando somente o último pedaço, que é o nome do arquivo.

- **Para suprimir à direita da ocorrência de uma cadeia**

  O uso do percentual (**%**) é como se olhássemos o jogo-da-velha (**#**) no espelho, isto é, são simétricos. Então vejamos um exemplo para provar isso:

```
$ echo $cadeia
Casa da Mãe Joana
$ echo ${cadeia%' '*}
Casa da Mãe
$ echo ${cadeia%%' '*}
Casa
```

- **Para trocar a primeira ocorrência de uma subcadeia em uma cadeia por outra:**

```
$ echo $cadeia
Casa da Mãe Joana
$ echo ${cadeia/Mãe/Vovó}
Casa da Vovó Joana
$ echo ${cadeia/Mãe /}
Casa da Joana
```

Neste caso preste a atenção quando for usar metacaracteres, eles são gulosos! Eles sempre combinarão com a maior possibilidade, veja o exemplo a seguir onde a intenção era trocar **Casa da Mãe Joana** por **Residência da mãe Joana**:

```
$ echo $cadeia
Casa da Mãe Joana
$ echo ${cadeia/*a /Residência }
Residência Mãe Joana
```

A ideia era pegar tudo até o primeiro **a** (a seguido de espaço em branco), para trocar **Casa** por **Residência** mas o que foi trocado foi tudo até o último **a** (**a** seguido de espaço em branco, que foi encontrado em **da**), Isto poderia ser resolvido de diversas maneiras, veja algumas:

```
$ echo ${cadeia/*sa/Residência}
Residência da Mãe Joana
$ echo ${cadeia/????/Lar}
Lar da Mãe Joana
```

- **Trocando todas as ocorrências de uma subcadeia por outra.**
  - **$ echo ${cadeia//a/ha}**
  - **Chasha dha Mãe Johanha**

Trocamos todos as letras **a** por **ha**, só para demonstrar que a troca não tem de ser de um para um. Veja agora um *script* para tirar espaços em branco dos nomes dos arquivos.

```
$ cat TiraBranco.sh
#!/bin/bash
#  Renomeia arquivos com espaços nem branco
#+ no nome, trocando-os por sublinhado (_).
Erro=0
for Arq in *' '*      # Expande para todos os arquivos com espaço em branco no nome
do
[ -f ${Arq// /_} ] && {
    echo $Arq não foi renomeado
    Erro=1
    continue
    }
mv "$Arq" "${Arq// /_}"
done 2> /dev/null     #  Caso não exista arquivo com brancos o for dá erro
exit $Erro
```

Um macete muito interessante: no exemplo que acabamos de ver, se quiséssemos testar se existiam arquivos com espaços no nome e fizéssemos:

`[[ -f *' '* ]]` ou `[ -f *' '* ]`

E se existisse mais de um arquivo que atendesse a esta característica, você ganharia um erro porque o comando `test -f` é unário, isto é, ele só verifica a existencia de um único arquivo. Para contornar isso, poderíamos fazer:

```
ls *' '* > /dev/null 2> /dev/null || echo Não há arquivo com espaço no nome.
```

Experimente!

- **Trocando uma subcadeia no início de uma variável**

```
$ echo $Passaro
quero quero
$ echo "Como diz o sulista - "${Passaro/#quero/não}
Como diz o sulista - não quero
```

- **Trocando uma subcadeia no fim de uma variável**

```
$ echo "Como diz o nordestino - "${Passaro/%quero/não}
Como diz o nordestino - quero não
```

- **Para listar o valor de uma variável apontada por outra**

Digamos que o conteúdo da variável `$a` seja `b` e da variável `$b` seja `5`. Veja como listar o valor de `$b` a partir de `$a`:

```
$ a=b
$ b=5
$ echo ${!a}
5
```

- **Mudando caixa (alta ou baixa) de letras**

Essas expansões modificam a caixa (alta ou baixa) das letras do texto que está sendo expandido. Quando usamos circunflexo (^), a expansão é feita para maiúsculas, quando usamos vírgula (,), a expansão é feita para minúsculas e quando usamos til (~) a expansão troca a caixa das letras.

```
$ Nome="botelho"
$ echo ${Nome^}              # 1° letra em maiúscula
Botelho
$ echo ${Nome^^}     # Todas as letras em maiúsculas
BOTELHO
$ Nome="botelho carvalho"
$ echo ${Nome^}              # 1° letra em maiúscula
Botelho carvalho
$ Coisa="AAAbbb cccDDD"
$ echo ${Coisa~}     # Troca case da 1° letra
aAAbbb CccDDD
$ echo ${Coisa~~}    # Troca case de todas as letras
aaaBBB CCCddd
```

Um fragmento de *script* que pode facilitar a sua vida:

```
read -p "Deseja continuar (S/n)? " # S maiúsculo (default)
[[ ${REPLY^} == N ]] && exit
```

Esta forma evita testarmos se a resposta dada foi um **N** (maiúsculo) ou um **n** (minúsculo).

No rWindows, além dos vírus e da instabilidade, também são frequentes nomes de arquivos com espaços em branco e quase todos em maiúsculas. Já vimos um exemplo de como trocar os espaços em branco por sublinha (_), no próximo veremos como passá-los para minúsculas (se você recebe muitos arquivos daquela coisa, o melhor é criar um *script* que seja a junção desses dois).

```
$ cat trocacase.sh
#!/bin/bash
#  Se o nome do arquivo tiver pelo menos uma
#+ letra maiúscula, troca-a para minúscula

for Arq in *[A-Z]*   #  Arquivos com pelo menos 1 maiúscula
do
if  [ -f "${Arq,,}" ]         #  Arq em minúsculas já existe?
then
    echo ${Arq,,} já existe
else
mv "$Arq" "${Arq,,}"
fi
done
```

Uma coisa muito importante, sob o Bash acontece um caso, para mim sem explicação: A ordem dos caracteres em expansões do tipo **[...]** é determinada pela variável **LC_COLLATE** e a sequência *default* dos caracteres é diferente da que usamos. Então é necessário que se faça:

```
export LC_COLLATE=C
```

Você pode botar isso em cada script mas eu prefiro usar esta expressão no **/etc/profile** das máquinas que administro pois atende a todos *scripts* e usuários.

16.3. Vetores ou Arrays

Um vetor ou *array* é um método para se tratar diversas informações (normalmente do mesmo tipo) sob um único nome, que é o nome do vetor. Poderíamos ter, por exemplo, um vetor chamado **cervejas**, que armazenasse **Skol**, **Antártica**, **Polar**, **Serra Malte**. Para acessarmos esses dados, precisamos de um índice.

Assim, **NomeDoVetor[Indice]** contém um valor, ou seja: **cervejas[0]** contém **Skol** e **cerveja[2]** contém **Polar**.

Existem 4 formas de se declarar um vetor:

```
vet=(EL0 EL1 ... ELn)
```

Cria o vetor **vet**, inicializando-o com os elementos **EL1**, **EL2**, ... **ELn**. Se for usado como **vet=()**, o vetor **vet** será inicializado vazio.

Declarando-se desta forma, se **vet** já existir, perderá **todos** os antigos valores, recebendo os novos.

```
vet[N]=VAL
```

Cria o elemento índice **N** do vetor **vet** com o valor **VAL**. Se não existir, o vetor **vet** será criado.

```
declare -a vet
```

Cria o vetor **vet** vazio. Caso **vet** já exista, se manterá inalterado.

```
declare -A vet
```

Cria um vetor associativo (aquele cujos índices não são numéricos). Essa é a única forma de declarar um vetor associativo, que só é suportado a partir da versão 4.0 do Bash.

Para se verificar o conteúdo de um elemento de um vetor, devemos usar a notação **${vet[NN]}**.

Os elementos de um vetor não precisam ser contínuos. Veja:

```
$ Familia[10]=Silvina
$ Familia[22]=Juliana
$ Familia[40]=Paula
$ Familia[51]=Julio
$ echo ${Familia[22]}
Juliana
$ echo ${Familia[18]}  # Não existe
$ echo ${Familia[40]}
Paula
```

Como você pode observar, foi criado um vetor com índices esparsos e quando se pretendeu listar um inexistente, simplesmente o retorno foi nulo.

O Bash suporta a notação **vet=(val1 val2 ... valn)** para definir os valores dos **n** primeiros elementos do vetor **vet**, o UNIX não.
Vamos criar outro vetor, usando a notação sintática do Bash:

```
$ Frutas=(abacaxi banana laranja tangerina)
$ echo ${Frutas[1]}
banana
```

Êpa! Por esse último exemplo pudemos notar que a indexação de um vetor começa em zero e não em um, isto é, para listar abacaxi deveríamos ter feito:

```
$ echo ${Frutas[0]}
abacaxi
```

Mas como dá para perceber, dessa forma só conseguiremos gerar vetores densos, mas usando a mesma notação, ainda somente sob o Bash, poderíamos gerar vetores esparsos da seguinte forma:

```
$ Veiculos=([2]=jegue [5]=cavalo [9]=patinete)
```

Cuidado ao usar essa notação! Caso esse vetor já possuísse outros elementos definidos, os valores e os índices antigos seriam perdidos e após a atribuição só restariam os elementos recém criados.

Não mostrei isso no comando **read**, mas a sua opção **-a** lê direto para dentro de um vetor. Vejamos como isso funciona:

```
 $ read -a Animais <<< "cachorro gato cavalo"  # Usando Here Strings
```

Vamos ver se isso funcionou:

```
$ for i in 0 1 2
> do
>     echo ${Animais[$i]}
> done
cachorro
gato
cavalo
```

Ou, ainda, montando vetores a partir da leitura de arquivos:

```
$ cat nums
1 2 3
2 4 6
3 6 9
4 8 12
5 10 15

$ while read -a vet
> do
>     echo -e ${vet[0]}:${vet[1]}:${vet[2]}
> done < nums
1:2:3
2:4:6
3:6:9
4:8:12
5:10:15
```

Vamos voltar às frutas e acrescentar ao vetor a **fruta do conde** e a **fruta pão:**

```
$ Frutas[4]="fruta do conde"
$ Frutas[5]="fruta pão"
```

Para listar essas duas inclusões que acabamos de fazer em **Frutas**, repare que usarei expressões aritméticas sem problema algum:

```
$ echo ${Frutas[10-6]}
fruta do conde
$ echo ${Frutas[10/2]}
fruta pão
$ echo ${Frutas[2*2]}
fruta do conde
$ echo ${Frutas[0*3]}
abacaxi
```

16.3.1. Um pouco de manipulação de vetores

De forma idêntica ao que vimos em passagem de parâmetros, o asterisco (*) e a arroba (@) servem para listar todos. Dessa forma, para listar todos os elementos de um vetor podemos fazer:

```
$ echo ${Frutas[*]}
 abacaxi banana laranja tangerina fruta do conde fruta pão
```

ou:

```
$ echo ${Frutas[@]}
abacaxi banana laranja tangerina fruta do conde fruta pão
```

E qual será a diferença entre as duas formas de uso? Bem, como poucos exemplos valem mais que muito blá-blá-blá, vamos listar todas as frutas, uma em cada linha:

```
$ for fruta in ${Frutas[*]}
> do
>     echo $fruta
> done
abacaxi
banana
laranja
tangerina
fruta
do
conde
fruta
pão
```

Ops, não era isso que eu queria! Repare que a **fruta do conde** e a **fruta pão** ficaram quebradas. Vamos tentar usando arroba (@):

```
$ for fruta in ${Frutas[@]}
> do
>     echo $fruta
> done
abacaxi
banana
laranja
tangerina
fruta
do
conde
fruta
pão
```

Hiii, deu a mesma resposta! Ahh, já sei! O Bash está vendo o espaço em branco entre as palavras de **fruta do conde** e **fruta pão** como um separador de campos (veja o que foi dito anteriormente sobre a variável **$IFS**) e parte as frutas em pedaços. Como já sabemos, devemos usar aspas para proteger essas frutas da curiosidade do *Shell*. Então vamos tentar novamente:

```
$ for fruta in "${Frutas[*]"}
> do
>     echo $fruta
> done
abacaxi banana laranja tangerina fruta do conde fruta pão
```

Epa, piorou! Então vamos continuar tentando:

```
$ for fruta in "${Frutas[@]}"
> do
>     echo $fruta
> done
abacaxi
banana
laranja
tangerina
fruta do conde
fruta pão
```

Voilà! Agora funcionou! Então é isso, quando usamos **"$@"** (entre aspas), ela não parte o elemento do vetor em listagens como a que fizemos. Isto também é válido quando estamos falando de passagem de parâmetro e da substituição de **"$@"**.

Existe uma outra forma de fazer o mesmo. Para mostrá-la vamos montar um vetor esparso, formado por animais de nomes compostos:

```
$ Animais=([2]="Mico Leão" [5]="Galinha d'Angola" [8]="Gato Pardo")
```

Agora veja como podemos listar os índices:

```
$ echo ${!Animais[*]}
2 5 8
$ echo ${!Animais[@]}
2 5 8
```

Essas construções listam os índices do vetor, sem diferença alguma na resposta. Assim sendo, podemos escolher uma delas para mostrar também todos os elementos de **Animais**, da seguinte forma:

```
$ for Ind in ${!Animais[@]}   # Ind recebe cada um dos índices
> do
> echo ${Animais[Ind]}
> done
Mico Leão
Galinha d'Angola
Gato Pardo
```

Para obtermos a quantidade de elementos de um vetor, ainda semelhantemente à passagem de parâmetros, fazemos:

```
$ echo ${#Frutas[*]}
6
```

ou

```
$ echo ${#Frutas[@]}
6
```

Repare, no entanto, que esse tipo de construção lhe devolve a quantidade de elementos de um vetor, e não o seu maior índice. Veja este exemplo com o vetor **Veículos**, que, como vimos nos exemplos anteriores, tem o índice **[9]** em seu último elemento:

```
$ echo ${!Veiculos[@]}        # Só para relembrar os índices de Veiculos
2 5 9
$ echo ${#Veiculos[*]}
3
$ echo ${#Veiculos[@]}
3
```

Por outro lado, se especificarmos o índice, essa expressão devolverá a quantidade de caracteres daquele determinado elemento do vetor.

```
$ echo ${Frutas[1]}
banana
$ echo ${#Frutas[1]}
6
```

Vamos entender como se copia um vetor inteiro para outro. A esta altura dos acontecimentos, já sabemos que, como existem elementos do vetor **Frutas** compostos por várias palavras separadas por espaços em branco, devemos nos referir a todos os elementos indexando com arroba **[@]**. Vamos então ver como copiar:

```
$ Vetor="${Frutas[@]}"
$ echo "${Vetor[4]}"
$ echo "$Vetor"
abacaxi banana laranja tangerina fruta do conde fruta pão
```

O que aconteceu nesse caso foi que eu criei uma variável chamada **$Vetor** com o conteúdo de todos os elementos do vetor **Frutas**. Como sob o Bash eu posso definir um *array* colocando os valores de seus elementos entre parênteses, deveria então ter feito:

```
$ Vetor=("${Frutas[@]}")
$ echo "${Vetor[4]}"
fruta do conde
$ echo "${Vetor[5]}"
fruta pão
```

Você se lembra que a substituição de processos que usa dois pontos (**:**) serve para especificar uma zona de corte em uma variável. Vamos relembrar:

```
$ var=1234567890
$ echo ${var:1:3}       # O número 1 significa 2° caractere. Origem zero, não esqueça
234
$ echo ${var:4}         # A partir do 5° caractere até o fim
567890
```

Em vetores, o seu comportamento é similar, porém agem sobre os seus elementos e não sobre seus caracteres como em variáveis, vide o exemplo anterior. Vamos exemplificar para entender:

```
$ echo ${Frutas[@]:1:3}   # A partir do 2° elemento, listar 3 elementos
banana laranja tangerina
$ echo ${Frutas[@]:4}     # A partir do 5° caractere até o fim
fruta do conde fruta pão
```

Se fosse especificado um elemento, este seria visto como uma variável.

```
$ echo ${Frutas[0]:1:4}
baca
```

Experimente agora para ver o que acontece na prática o que vimos nas aulas sobre variáveis e sobre expansão de parâmetros, porém adaptando as construções ao uso de vetores. Garanto que você entenderá tudo muito facilmente devido à semelhança entre tratamento de vetores e de variáveis.

Primeiro vou te dar um exemplo do que acabei de falar, depois você inventa uns *scripts* para testar outras expansões de parâmetros como as provocadas por `%`, `%%`, `#`, `##`, `...`, ok?

Então vamos criar o vetor **`Frase`**:

```
$ Frase=(Alshançar o shéu é sensashional. Um sushesso\!)
$ echo ${Frase[*]//sh/c}
Alcançar o céu é sensacional. Um sucesso!
```

Para coroar isso tudo, num determinado dia estávamos diversos "shelleiros" batendo papo, quando chegou um colega perguntando como ele poderia contar a quantidade de cada anilha que teria de usar para fazer uma cabeação.

Anilha são aqueles pequenos anéis numerados que você vê nos cabos de rede, que servem para identificá-los. Em outras palavras, o problema era dizer quantas vezes ele usaria cada algarismo em um dado intervalo entre os números. Vejamos a proposta de solução:

```
$ cat anilhas.sh
#!/bin/bash
Tudo=$(eval echo {$1..$2})              # Recebe os num. entre $1 e $2
for ((i=0; i<${#Tudo}; i++))
{
[ ${Tudo:i:1} ] || continue          # Espaço entre 2 números
let Algarismo[${Tudo:i:1}]++         # Incrementa vetor do algarismo
}
for ((i=0; i<=9; i++))
{
printf "Algarismo %d = %2d\n" \
    $i ${Algarismo[$i]:-0}           # Se o elemento for vazio, lista zero
}

$ anilhas.sh 234 252  # Para numerar cabos desde 234 até 252
Algarismo 0 =  2
Algarismo 1 =  2
Algarismo 2 = 21
Algarismo 3 =  7
Algarismo 4 = 12
Algarismo 5 =  5
Algarismo 6 =  2
Algarismo 7 =  2
Algarismo 8 =  2
Algarismo 9 =  2
```

Me divirto muito escrevendo um *script* como esse, no qual foi empregado somente Bash puro. Além de divertido, é antes de mais nada rápido.

16.3.2. Vetores associativos

A partir do Bash 4.0, passou a existir o vetor associativo. Chama-se vetor associativo, aqueles cujos índices são alfabéticos. As regras que valem para os vetores inteiros, valem também para os associativos, porém antes de valorar estes últimos, é obrigatório declará-los.

Exemplo

```
$ declare -A Animais # Obrigatório para vetor associativo
$ Animais[cavalo]=doméstico
$ Animais[zebra]=selvagem
$ Animais[gato]=doméstico
$ Animais[tigre]=selvagem
```

É impossível gerar todos os elementos de uma só vez, como nos vetores inteiros. Assim sendo, não funciona a sintaxe:

```
Animais = ([cavalo]=doméstico [zebra]=selvagem [gato]=doméstico [tigre]=selvagem)
$ echo ${Animais[@]}
doméstico selvagem doméstico selvagem
$ echo ${!Animais[@]}
gato zebra cavalo tigre
```

Repare que os valores não são ordenados, ficam armazenados na ordem que são criados, diferentemente dos vetores inteiros que ficam em ordem numérica.

Supondo que esse vetor tivesse centenas de elementos, para listar separadamente os domésticos dos selvagens, poderíamos fazer um script assim:

```
$ cat animal.sh
#!/bin/bash
# Separa animais selvagens e domésticos
declare -A Animais
Animais[cavalo]=doméstico               # Criando vetor para teste
Animais[zebra]=selvagem                 # Criando vetor para teste
Animais[gato]=doméstico                 # Criando vetor para teste
Animais[tigre]=selvagem                 # Criando vetor para teste
Animais[urso pardo]=selvagem            # Criando vetor para teste
for Animal in "${!Animais[@]}"          # Percorrendo vetor pelo índice
do
if [[ "${Animais[$Animal]}" == selvagem ]]
then
    Sel=("${Sel[@]}" "$Animal")     # Concatenado novo animal no vetor selvagens
else
    Dom=("${Dom[@]}" "$Animal")     # Concatenado novo animal no vetor domésticos
fi
done
#  Operador condicional, usado para descobrir qual
#+ vetor tem mais elementos. Veja detalhes na seção
#+ O interpretador aritmético do Shell
Maior=$[${#Dom[@]}>${#Sel[@]}?${#Dom[@]}:${#Sel[@]}]
clear
tput bold                                       # Cabeçalho
printf "%-15s%-15s\n" Domésticos Selvagens      # Cabeçalho
tput sgr0                                       # Cabeçalho

for ((i=0; i<$Maior; i++))
{
    tput cup $[1+i] 0; echo ${Dom[i]}
    tput cup $[1+i] 14; echo ${Sel[i]}
}
```

Gostaria de chamar a sua atenção para um detalhe: neste script me referi a um elemento de vetor associativo empregando `${Animais[$Animal]}` ao passo que me referi a um elemento de um vetor inteiro usando `${Sel[i]}`. Ou seja, quando usamos uma variável como índice de um vetor inteiro, não precisamos prefixá-la com um cifrão (`$`), ao passo que no vetor associativo, o cifrão (`$`) é obrigatório.

16.3.3. Lendo um arquivo para um vetor (comando mapfile)

Ainda falando do Bash 4.0, eis que ele surge com uma outra novidade: o comando intrínseco (*builtin*) **mapfile**, cuja finalidade é jogar um arquivo de texto inteiro para dentro de um vetor, sem *loop* ou substituição de comando

— EPA! Isso deve ser muito rápido!

— E é. Faça os testes e comprove!

Exemplo

```
$ cat frutas
abacate
maçã
morango
pera
tangerina
uva

$ mapfile vet < frutas  # Mandando frutas para vetor vet
$ echo ${vet[@]}        # Listando todos elementos de vet
abacate maçã morango pera tangerina uva
```

Obteríamos resultado idêntico se fizéssemos:

```
$ vet=($(cat frutas))
```

Porém, isso seria mais lento, porque a substituição de comando é executada em um *subshell*.

Uma outra forma de fazer isso que logo vem à cabeça é ler o arquivo com a opção **-a** do comando **read**. Vamos ver como seria o comportamento disso:

```
$ read -a vet < frutas
$ echo ${vet[@]}
abacate
```

Como deu para perceber, foi lido somente o primeiro registro de frutas.

Agora já chega, o papo hoje foi muito chato porque foi muita decoreba, mas o principal é você ter entendido o que te falei e, quando precisar, consulte estas anotações e guarde-as para consultas futuras.

No exercício que deixarei, vou te dar a maior moleza e só vou cobrar o seguinte: pegue a rotina **pergunta.func**, (a que na qual falamos no início do nosso bate-papo de hoje) e otimize-a para que a variável **$SN** receba o valor *default* por expansão de parâmetros, como vimos.

Sumário

17. Etcétera

- 17.1. O comando eval
- 17.2. Sinais de Processos
  - 17.2.1. Sinais assassinos
  - 17.2.2. O trap não atrapalha
- 17.3. Comando getopts
- 17.4. Named Pipes
- 17.5. Sincronização de processos.
- 17.6. Bloqueio de arquivos
- 17.7. Substituição de processos

17. Etcétera

A partir de agora, vou te mostrar uns comandos muito importantes mas que não se encaixam em nenhum dos contextos que vimos até agora. São os famosos "outros".

— E aê amigo, te dei a maior moleza, né? Um exerciciozinho muito simples...

— É mais nos testes que eu fiz, e de acordo com o que você ensinou sobre substituição de parâmetros, achei que deveria fazer outras alterações nas funções que desenvolvemos para torná-las de uso geral como você me disse que todas as funções deveriam ser, quer ver?

— Claro né mané, se te pedi para fazer é porque estou afim de te ver aprender, mas peraí, dá um tempo! Vai, mostra aí o que você fez.

— Bem, além do que você pediu, eu reparei que o programa que chamava a função, teria de ter previamente definidas a linha em que seria dada a mensagem e a quantidade de colunas. O que fiz foi incluir duas linhas - nas quais empreguei substituição de parâmetros - que caso uma destas variáveis não fosse informada, a própria função atribuiria. A linha de mensagem seria três linhas acima do fim da tela e o total de colunas seria obtido pelo comando **tput cols**. Veja como ficou:

```
$ cat pergunta.func
#  A funcao recebe 3 parametros na seguinte ordem:
#+ $1 - Mensagem a ser dada na tela
#+ $2 - Valor a ser aceito com resposta default
#+ $3 - O outro valor aceito
#+ Supondo que $1=Aceita?, $2=s e $3=n, a linha
#+ abaixo colocaria em Msg o valor "Aceita? (S/n)"
TotCols=${TotCols:-$(tput cols)} # Se nao estava definido, agora esta
LinhaMesg=${LinhaMesg:-$(($(tput lines)-3))}     # Idem
Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
TamMsg=${#Msg}
Col=$(((TotCols - TamMsg) / 2))   # Para centrar Msg na linha
tput cup $LinhaMesg $Col
read -n1 -p "$Msg " SN
SN=${SN:-$2}                      # Se vazia coloca default em SN
SN=$(echo $SN | tr A-Z a-z) # A saida de SN serah em minuscula
tput cup $LinhaMesg $Col; tput el # Apaga msg da tela
```

— Gostei, você já se antecipou ao que eu ia pedir. Só pra gente encerrar este papo de substituição de parâmetros, repare que a legibilidade está horrorível, mas a performance, isto é, velocidade de execução, está ótima. Como funções são como cuecas, já que cada um usa as suas, e quase não dão manutenção, eu sempre opto pela performance.

— Hoje vamos sair daquela chatura que foi o nosso último papo e vamos voltar à lógica saindo da decoreba, mas volto a te lembrar, tudo que eu te mostrei da outra é válido e quebra um galhão, guarde aqueles guardanapos que rabiscamos que, mais cedo ou mais tarde, vão te ser muito úteis.

17.1. O comando eval

— Vou te dar um problema que eu duvido que você resolva:

```
$ var1=3
$ var2=var1
```

— Te dei estas duas variáveis, e quero que você me diga como eu posso, só me referindo a **$var2**, listar o valor de **$var1** (**3**).

— A isso é mole, é só fazer:

```
 echo $`echo $var2`
```

— Repare que eu coloquei o **echo $var2** entre crases (`), que desta forma terá prioridade de execução e resultará em **var1**, montando **echo $var1** que produzirá **3**.

— A é? Então execute para ver se está correto.

```
$ echo $`echo $var2`
$var1
```

— Ué! Que foi que houve? O meu raciocínio me parecia bastante lógico...

— O seu raciocínio realmente foi lógico, o problema é que você esqueceu de uma das primeiras coisas que te falei aqui nas nossas aulas e vou repetir. O *Shell* usa a seguinte ordem para resolver uma linha de comandos:

- Resolve os redirecionamentos;
- Substitui as variáveis pelos seus valores;
- Resolve e substitui os meta caracteres;
- Resolve aliases, verifica se programa existe, se você tem direito de executá-lo, ...
- Passa a linha já toda esmiuçada para execução.

Desta forma, quando chegou na fase de resolução de variáveis, que como eu disse é anterior à execução, a única variável existente era **$var2** e por isso a tua solução produziu como saída **$var1**. O comando **echo** identificou isso como uma cadeia e não como uma variável.

Problemas deste tipo são relativamente frequentes e é para isso que existe a instrução **eval**, cuja sintaxe é:

```
eval CMD
```

Onde **CMD** é uma linha de comando qualquer que você poderia inclusive executar direto no *prompt* do terminal. Quando você põe o **eval** na frente, no entanto, o que ocorre é que o *Shell* vê a linha passada e resolve todos os parâmetros possíveis de **CMD** e passa tudo já resolvido para o **eval**. o que esta instrução faz é passar **CMD** para execução, submetendo-o assim a uma nova interpretação do *Shell*, dando então na prática duas passadas (interpretadas) em **CMD**.

Desta forma se executássemos o comando que você propôs colocando o **eval** à sua frente, teríamos a saída esperada, veja:

```
$ eval echo $`echo $var2`
3
```

Este exemplo também poderia ter sido feito da seguinte maneira:

```
$ eval echo \$$var2
3
```

Na primeira passada a contrabarra (****) seria retirada e **$var2** seria resolvido produzindo **var1**, para a segunda passada teria sobrado **echo $var1**, que produziria o resultado esperado.

Agora vou colocar um comando dentro de **var2:**

```
$ var2=ls
```

Vou executar:

```
$ $var2
10porpag1.sh   alo2.sh      listamusica   logaute.sh
10porpag2.sh   confuso      listartista   mandamsg.func
10porpag3.sh   contpal.sh   listartista3  monbg.sh
alo1.sh        incusu       logado
```

Agora vamos colocar em **var2** o seguinte: **ls $var1;** e em **var1** vamos colocar **l***, vejamos:

```
$ var2='ls $var1'
$ var1='l*'
$ $var2
ls: $var1: No such file or directory
$ eval $var2
listamusica    listartista  listartista3  logado
logaute.sh
```

Novamente, no tempo de substituição das variáveis, **$var1** ainda não havia se apresentado ao *Shell* para ser resolvida, desta forma só nos resta executar o comando **eval** para dar as duas passadas necessárias.

Uma vez um colega de uma excelente lista sobre *Shell Script*, colocou uma dúvida: queria fazer um menu que numerasse e listasse todos os arquivos com extensão **.sh** e quando o operador escolhesse uma opção, o programa correspondente seria executado. A minha proposta foi a seguinte:

```
$ cat fazmenu
#!/bin/bash
#
# Lista numerando os programas com extensão .sh no
# diretório corrente e executa o escolhido pelo operador
#
function Erro
{
    echo "
    ***************ERRO***************
    Uso: $0 NNN, onde NNN é uma das opções apresentadas"
    exit 1
}
clear; i=1
printf "%11s\t%s\n\n" Opção Programa
CASE='case $opt in'
for arq in *.sh
do
    printf "\t%03d\t%s\n" $i $arq
    CASE="$CASE
        "$(printf "%03d)\t %s;;" $i $arq)
    i=$((i+1))
done
CASE="$CASE
          *)     Erro;;
esac"
read -n3 -p "Informe a opção desejada: " opt
printf -v opt '%03d' $opt          # Preenche $opt com zeros à esquerda
echo
eval "$CASE"
```

Parece complicado porque usei muito **printf** para formatação da tela, mas é bastante simples, vamos entendê-lo: o primeiro **printf** foi colocado para fazer o cabeçalho e logo em seguida comecei a montar dinamicamente a variável **$CASE**, na qual ao final será feito um **eval** para execução do programa escolhido. Repare no entanto que dentro do loop do **for** existem dois **printf**: o primeiro serve para formatar a tela e o segundo para montar o **case** (se antes do comando **read** você colocar uma linha **echo "$CASE"**, verá que o comando **case** montado dentro da variável está todo indentado. Frescura, né? ☺. Na saída do **for**, foi adicionada uma linha à variável **$CASE**, para no caso de se fazer uma opção inválida, ser executada a função **Erro** avisando ao operador e abortando o programa.

Vamos executá-lo para ver a saída gerada:

```
$ fazmenu.sh
Opcao Programa
001 10porpag1.sh
002 10porpag2.sh
003 10porpag3.sh
004 alo1.sh
005 alo2.sh
006 contpal.sh
007 fazmenu.sh
008 logaute.sh
009 monbg.sh
010 readpipe.sh
011 redirread.sh
Informe a opção desejada:
```

Neste programa seria interessante darmos uma opção de término, e para isso seria necessário a inclusão de uma linha após o *loop* de montagem da tela e alterarmos a linha na qual fazemos a atribuição final do valor da variável **$CASE**. Vejamos como ele ficaria:

```
$ cat fazmenu
#!/bin/bash
#
# Lista numerando os programas com extensão .sh no
# diretório corrente e executa o escolhido pelo operador
#
function Erro
{
    echo "
    ************************ERRO************************
    Uso: $0 NNN, onde NNN é uma das opções apresentadas"
    exit 1
}
clear; i=1
printf "%11s\t%s\n\n" Opção Programa
CASE='case $opt in'
for arq in *.sh
do
    printf "\t%03d\t%s\n" $i $arq
    CASE="$CASE
        "$(printf "%03d)\t %s;;" $i $arq)
    i=$((i+1))
done
printf "\t%d\t%s\n\n" 999 "Fim do programa" # Linha incluida
CASE="$CASE
        999)     exit;;                    # Linha incluida
          *)     Erro;;
esac"
read -n3 -p "Informe a opção desejada: " opt
printf -v opt '%03d' $opt                  # Preenche $opt com zeros à esquerda
echo
eval "$CASE"
```

17.2. Sinais de Processos

Existe no Linux uma coisa chamada sinal (*signal*). Existem diversos sinais que podem ser mandados para (ou gerados por) processos em execução. Vamos de agora em diante dar uma olhadinha nos sinais mandados para os processos e mais à frente vamos dar uma passada rápida pelos sinais gerados por processos. Você pode ver todos estes sinais com o comando **kill -l**

17.2.1. Sinais assassinos

Para mandar um sinal a um processo, usamos normalmente o comando **kill**, cuja sintaxe é:

```
kill -SIG PID
```

Onde **PID** é o identificador do processo (*Process IDentification* ou *Process ID*). Além do comando **kill**, algumas sequências de teclas também podem gerar um sinal **SIG** A tabela a seguir mostra os sinais mais importantes para monitorarmos:

Sinais Mais Importantes		
Sinal		**Gerado por:**
0	**EXIT**	Fim normal do programa
1	**SIGHUP**	Quando recebe um **kill -HUP**
2	**SIGINT**	Interrupção pelo teclado (**<CTRL+C>**)
3	**SIGQUIT**	Interrupção pelo teclado (**<CTRL+\>**)
15	**SIGTERM**	Quando recebe um **kill** ou **kill -TERM**

Além destes sinais, existe o famigerado **-9** ou **SIGKILL** que, para o processo que o está recebendo, equivale a meter o dedo no botão de desliga do computador, o que seria altamente indesejável, já que muitos programas necessitam "limpar o meio de campo" ao seu término. Se o seu final ocorrer de forma prevista, ou seja se tiver um término normal, é muito fácil de fazer esta limpeza, porém se o seu programa tiver um fim brusco muita coisa pode ocorrer:

- É possível que em um determinado espaço de tempo, o seu computador acabe ficando cheio de arquivos de trabalho inúteis
- Seu processador poderá ficar atolado de processos *zombies* e *defuncts* gerados por processos filhos que perderam os pais;
- É necessário liberar *sockets* abertos para não deixar os clientes congelados;
- Seus bancos de dados poderão ficar corrompidos porque sistemas gerenciadores de bancos de dados necessitam de um tempo para gravar seus *buffers* em disco (*commit*).

Enfim, existem mil razões para não usar um **kill** com o sinal **-9** e para monitorar fins anormais de programas.

17.2.2. O trap não atrapalha

Para fazer a monitoração descrita acima existe o comando **trap** cuja sintaxe é:

```
trap "CMD1; CMD2; CMDn" S1 S2 ... SN
```

ou

```
trap 'CMD1; CMD2; CMDn' S1 S2 ... SN
```

Onde os comandos **CMD1**, **CMD2**,...**CMDn** serão executados caso o programa receba os sinais **S1**, **S2**, ..., **SN**.

As aspas (") ou os apóstrofos (') só são necessários caso o **trap** possua mais de um comando **CMD** associado. Cada um dos **CMD** pode ser também uma função interna, uma externa ou outro *script*.

Para entender o uso de aspas (") e apóstrofos (') vamos recorrer a um exemplo que trata um fragmento de um *script* que faz um **ftp** para uma máquina remota (**$RemoComp**), na qual o usuário é **$Fulano**, sua senha é **$Segredo** e vai transmitir o arquivo contido em **$Arq**. Suponha ainda que estas quatro variáveis foram recebidas em uma rotina anterior de leitura e que este *script* é muito usado por diversas pessoas da instalação. Vejamos este trecho de código:

```
ftp -ivn $RemoComp << FimFTP >> /tmp/$$ 2>> /tmp/$$
    user $Fulano $Segredo
    binary
    get $Arq
FimFTP
```

Repare que, tanto as saídas do diálogo do **ftp**, como os erros encontrados, estão sendo redirecionados para **/tmp/$$**, o que é uma construção bastante normal para arquivos temporários usados em *scripts* com mais de um usuário, porque **$$** é a variável que contém o número do processo (**PID**), que é único, e com este tipo de construção evita-se que dois ou mais usuários disputem a posse e os direitos sobre o arquivo.

Caso este **ftp** seja interrompido por um **kill** ou um **<CTRL+C>**, certamente deixará lixo no disco. É exatamente esta a forma como mais se usa o comando **trap**. Como isto é trecho de um *script*, devemos, logo no seu início, como um de seus primeiros comandos, fazer:

```
trap "rm -f /tmp/$$ ; exit" 0 1 2 3 15
```

Desta forma, caso houvesse uma interrupção brusca (sinais **1**, **2**, **3** ou **15**) antes do programa encerrar (no **exit** dentro do comando **trap**), ou um fim normal (sinal **0**), o arquivo **/tmp/$$** seria removido.

Caso na linha de comandos do **trap** não houvesse a instrução **exit**, ao final da execução desta linha o fluxo do programa retornaria ao ponto em que estava quando recebeu o sinal que originou a execução deste **trap**.

Este **trap** poderia ser subdividido, ficando da seguinte forma:

```
trap "rm -f /tmp/$$" 0
trap "exit" 1 2 3 15
```

Assim ao receber um dos sinais o programa terminaria, e ao terminar, geraria um sinal **0**, que removeria o arquivo. Caso seu fim seja normal, o sinal também será gerado e o **rm** será executado.

Note também que o *Shell* pesquisa a linha de comandos uma vez quanto o **trap** é interpretado (e é por isso que é usual colocá-lo no início do programa) e novamente quando um dos sinais listados é recebido. Então, no último exemplo, o valor de **$$** será substituído no momento que o comando **trap** foi lido da primeira vez, já que as aspas (**"**) não protegem o cifrão (**$**) da interpretação do *Shell*.

Se você desejasse que a substituição fosse realizada somente quando recebesse o sinal, o comando deveria ser colocado entre apóstrofos (**'**). Assim, na primeira interpretação do **trap**, o *Shell* não veria o cifrão (**$**), porém os apóstrofos (**'**) seriam removidos e finalmente o *Shell* poderia substituir o valor da variável. Neste caso, a linha ficaria da seguinte maneira:

```
trap 'rm -f /tmp/$$ ; exit' 0 1 2 3 15
```

Suponha dois casos: você tem dois *scripts* que chamaremos de **script1**, cuja primeira linha será um **trap** e **script2**, sendo este último colocado em execução pelo primeiro, e por serem dois processos, terão dois **PID** distintos.

1° Caso: O **ftp** encontra-se em **script1**

Neste caso, o argumento do comando **trap** deveria vir entre aspas (**"**) porque caso ocorresse uma interrupção (**<CTRL+C>** ou **<CTRL+\>**) no **script2**, a linha só seria interpretada neste momento e o **PID** do **script2** seria diferente do encontrado em **/tmp/$$** (não esqueça que **$$** é a variável que contém o **PID** do processo ativo);

2° Caso: O **ftp** acima encontra-se em **script2**

Neste caso, o argumento do comando **trap** deveria estar entre apóstrofos (**'**), pois caso a interrupção se desse durante a execução de **script1**, o arquivo não teria sido criado. Caso ocorresse durante a execução de **script2**, o valor de **$$** seria o **PID** deste processo, que coincidiria com o de **/tmp/$$**.

O comando **trap**, quando executado sem argumentos, lista os sinais que estão sendo monitorados no ambiente, bem como a linha de comando que será executada quando tais sinais forem recebidos.

Assim como o comando **kill -l**, o **trap -l** também gera uma lista com todos os sinais aceitos

Se a linha de comandos do **trap** for nula (vazia), isto significa que os sinais especificados devem ser ignorados quando recebidos. Por exemplo, o comando:

```
trap "" 2
```

Especifica que o sinal de interrupção (**<CTRL+C>**) deve ser ignorado. No caso citado, quando não se deseja que sua execução seja interrompida. A sintaxe do último exemplo diz para o *Shell*: ao receber **<CTRL>+C** (sinal 2) não faça nada (**""**)

Este último exemplo é um erro comum, quando se deseja que um determinado sinal volte ao seu estado padrão (*default*), mas está errado. Para restaurar um sinal, o **2**, por exemplo, devemos fazer:

```
trap 2
```

Se você ignora um sinal, todos os *Subshells* irão ignorar este sinal. Portanto, se você especifica qual ação deve ser tomada quando receber um sinal, então todos os *Subshells* irão também tomar a ação quando receberem este sinal, ou seja, os sinais são automaticamente exportados. Para o sinal que temos mostrado (sinal **2**), isto significa que os *Subshells* serão encerrados.

Outra forma de restaurar um sinal ao seu *default* é fazendo:

```
trap - SINAL
```

Em *Korn Shell* (**ksh**) não existe a opção **-s** do comando **read** para ler uma senha. O que costumamos fazer é usar o comando **stty** com a opção **-echo** que inibe a escrita na tela até que se encontre um **stty echo** para restaurar esta escrita. Então, se estivéssemos usando o interpretador **ksh**, a leitura da senha teria que ser feita da seguinte forma:

```
echo -n "Senha: "
stty -echo
read Senha
stty echo
```

O problema neste tipo de construção é que caso o operador não soubesse a senha, ele provavelmente daria um **<CTRL+C>** ou um **<CTRL+\>** durante a instrução **read** para descontinuar o programa e, caso ele agisse desta forma, o que quer que ele escrevesse, não apareceria na tela do seu terminal. Para evitar que isso aconteça, o melhor a fazer é:

```
echo -n "Senha: "
trap "stty echo
      exit" 2 3
stty -echo
read Senha
stty echo
trap 2 3
```

Para terminar este assunto, abra uma console gráfica e escreva no *prompt* de comando o seguinte:

```
$ trap "echo Mudou o tamanho da janela" 28
```

Em seguida, pegue o *mouse* (arghh!!) e arraste-o de forma a variar o tamanho da janela corrente. Surpreso? É o *Shell* orientado a eventos...

Mais unzinho porque não pude resistir. Agora escreva assim:

```
$ trap "echo já era" 17
```

Em seguida faça:

```
$ sleep 3 &
```

Você acabou de criar um *Subshell* que irá dormir durante três segundos em *background*. Ao fim deste tempo, você receberá a mensagem **já era**, porque o sinal **17** é emitido a cada vez que um *Subshell* termina a sua execução.

Para devolver estes sinais aos seus defaults, faça:

```
$ trap 17 28
```

Ou

```
$ trap - 17 28
```

Acabamos de ver mais dois sinais que não são tão importantes como os que vimos anteriormente, mas vou registrá-los na tabela a seguir:

Sinais Não Muito Importantes		
Sinal		**Gerado por:**
17	SIGCHLD	Fim de um processo filho
28	SIGWINCH	Mudança no tamanho da janela gráfica

17.3. Comando getopts

O comando **getopts** recupera as opções e seus argumentos de uma lista de parâmetros de acordo com a sintaxe POSIX.2, isto é, letras (ou números) após um sinal de menos (**-**) seguidas ou não de um argumento; no caso de somente letras (ou números) elas podem ser agrupadas. Você deve usar este comando para "fatiar" opções e argumentos passados para o seu *script*.

Sintaxe:

```
getopts OPCS VAR
```

A **OPCS** deve explicitar uma cadeia de caracteres com todas as opções reconhecidas pelo *script*, assim se ele reconhece as opções **-a**, **-b** e **-c**, **OPCS** deve ser **abc**. Se você deseja que uma opção seja seguida por um argumento, ponha dois-pontos (**:**) depois da letra, como em **a:bc**. Isto diz ao **getopts** que a opção **-a** tem a forma:

```
 -a ARGUMENTO
```

Normalmente um ou mais espaços em branco separam o parâmetro da opção; no entanto, **getopts** também manipula parâmetros que vêm colados à opção como em:

```
 -aARGUMENTO
```

Fique atento! **OPCS** não pode conter interrogação (**?**).

O **VAR** constante da linha de sintaxe acima, define uma variável que cada vez que o comando **getopts** for executado, receberá a próxima opção dos parâmetros posicionais e a colocará na variável **VAR**.

O **getopts** coloca uma interrogação (**?**) na variável definida em **VAR** se achar uma opção não definida em **OPCS** ou se não achar o argumento esperado para uma determinada opção (e é por isso que **VAR** não pode conter ponto de interrogação (**?**)).

Como já sabemos, cada opção passada por uma linha de comandos tem um índice numérico, assim, a primeira opção estará contida em **$1**, a segunda em **$2**, e assim por diante. Quando o **getopts** obtém uma opção, ele armazena o índice do próximo parâmetro a ser processado na variável **OPTIND**.

Quando uma opção tem um argumento associado (indicado pelo **:** na **OPCS**), **getopts** armazena o argumento na variável **OPTARG**. Se uma opção não possui argumento ou o argumento esperado não foi encontrado, a variável **OPTARG** será "matada" (**unset**).

O comando encerra sua execução quando:

- Encontra um parâmetro que não começa por menos (**-**);
- O parâmetro especial **-** marca o fim das opções;
- Quando encontra um erro (por exemplo, uma opção não reconhecida).

O exemplo abaixo é meramente didático, servindo para mostrar, em um pequeno fragmento de código, o uso pleno do comando.

```
$ cat getoptst.sh
#!/bin/sh

#  Execute assim:
#
#       getopts.sh -h -Pimpressora arq1 arq2
#
#  e note que as informacoes de todas as opcoes sao exibidas
#
#  A cadeia 'P:h' diz que a opcao -P eh uma opcao complexa
#+ e requer um argumento, e que -h eh uma opcao simples
#+ que nao requer argumentos.

while getopts 'P:h' OPT_LETRA
do
    echo "getopts fez a variavel OPT_LETRA igual a '$OPT_LETRA'
       OPTARG eh '$OPTARG'"
done
used_up=$((OPTIND - 1))
echo "Dispensando os primeiros \$OPTIND-1 = $used_up argumentos"
shift $used_up
echo "O que sobrou da linha de comandos foi '$*'"
```

Para entendê-lo melhor, vamos executá-lo como está sugerido em seu cabeçalho:

```
$ getopts.sh -h -Pimpressora arq1 arq2
getopts fez a variavel OPT_LETRA igual a 'h'
       OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
       OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 2 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

Desta forma, sem ter muito trabalho, separei todas as opções com seus respectivos argumentos, deixando somente os parâmetros que foram passados pelo operador para posterior tratamento.

Se fosse um programa profissional, não mandaria esses dados para a tela, mas sim para um vetor.

Repare que se tivéssemos escrito a linha de comando com o argumento (**impressora**) separado da opção (**-P**), o resultado seria exatamente o mesmo, exceto pelo **$OPTIND**, já que neste caso ele identifica um conjunto de três opções/argumentos e no anterior somente dois. Veja só:

```
$ getopts.sh -h -P impressora arq1 arq2
getopts fez a variavel OPT_LETRA igual a 'h'
       OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
       OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 3 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

Repare, no exemplo a seguir, que se passarmos uma opção inválida, a variável **$OPT_LETRA** receberá um ponto-de-interrogação (**?**) e a **$OPTARG** será "apagada" (**unset**).

```
$ getopts.sh -f -Pimpressora arq1 arq2 # A opção -f não é valida
Illegal option -f
getopts fez a variavel OPT_LETRA igual a '?'
       OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
       OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 2 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

— Me diz uma coisa: você não poderia ter usado um **case** para evitar o **getopts?**

— Poderia sim, mas para que? Os comandos estão aí para serem usados... O exemplo dado foi didático, mas imagine um programa que aceitasse muitas opções e seus parâmetros poderiam ou não estar colados às opções, suas opções também poderiam ou não estar coladas, ia ser um **case** infernal e com **getopts** é só seguir os passos acima.

— Para praticar o que falamos, vou te deixar um problema para resolver: quando você varia o tamanho de uma tela, no seu centro não aparece dinamicamente em vídeo reverso a quantidade de linhas e colunas? Então! Eu quero que você reproduza isso usando a linguagem *Shell*.

```
$ cat tamtela.sh
#!/bin/bash
#
# Coloca no centro da tela, em video reverso,
# a quantidade de colunas e linhas
# quando o tamanho da tela eh alterado.
#
trap Muda 28    # 28 = sinal gerado pela mudanca no tamanho
                # da tela e Muda eh a funcao que fara isso.

Bold=$(tput bold)     # Modo de enfase
Rev=$(tput rev)       # Modo de video reverso
Norm=$(tput sgr0)     # Restaura a tela ao padrao default

Muda ()
{
    clear
    Cols=$(tput cols)
    Lins=$(tput lines)
    tput cup $(($Lins / 2)) $(((Cols - 7) / 2)) # Centro da tela
    echo $Bold$Rev$Cols X $Lins$Norm
}

clear
read -n1 -p "Mude o tamanho da tela ou tecle algo para terminar "
```

17.4. Named Pipes

Um outro tipo de *pipe* é o *named pipe*, que também é chamado de **FIFO**. **FIFO** é um acrônimo de *First In First Out* que se refere à propriedade em que a ordem dos bytes entrando no *pipe* é a mesma que a da saída. O *name* em *named pipe* é, na verdade, o nome de um arquivo. Os arquivos tipo *named pipes* são exibidos pelo comando **ls** como qualquer outro, com poucas diferenças, veja:

```
$ ls -l pipe1
prw-r--r--   1   julio   dipao   0 Jan 22 23:11 pipe1|
```

O **p** na coluna mais à esquerda indica que **fifo1** é um *named pipe*. O resto dos bits de controle de permissões, quem pode ler ou gravar o *pipe*, funcionam como um arquivo normal. Nos sistemas mais modernos uma barra vertical (**|**) é colocada ao fim do nome do arquivo, é outra dica, e nos sistemas LINUX, onde a opção de cor está habilitada, o nome do arquivo é escrito em vermelho por *default*.

Nos sistemas mais antigos, os *named pipes* são criados pelo programa **mknod**, normalmente situado no diretório **/etc**.

Nos sistemas mais modernos, a mesma tarefa é feita pelo **mkfifo**. O programa **mkfifo** recebe um ou mais nomes como argumento e cria *pipes* com estes nomes. Por exemplo, para criar um *named pipe* com o nome **pipe1**, faça:

```
$ mkfifo pipe1
```

Como sempre, a melhor forma de mostrar como algo funciona é dando exemplos. Suponha que nós tenhamos criado o *named pipe* mostrado anteriormente. Vamos agora trabalhar com duas sessões ou duas consoles virtuais ou uma de cada. Em uma delas faça:

```
$ ls -l > pipe1
```

e em outra faça:

```
$ cat < pipe1
```

Voilá! A saída do comando executado na primeira console foi exibida na segunda. Note que a ordem em que os comandos ocorreram não importa.

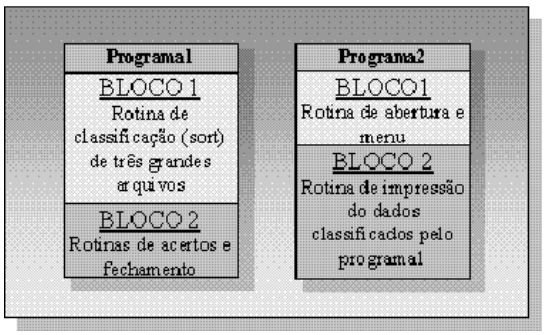
Se você prestou atenção, reparou que o primeiro comando executado, parecia ter "pendurado, congelado". Isto acontece porque a outra ponta do *pipe* ainda não estava conectada, e então o sistema operacional suspendeu o primeiro processo até que o segundo "abrisse" o *pipe*. Para que um processo que usa *pipe* não fique em modo de *wait*, é necessário que em uma ponta do *pipe* tenha um processo "tagarela" e na outra um "ouvinte" e no exemplo que demos, o **ls** era o "falador" e o **cat** era o "orelhão".

Uma aplicação muito útil dos *named pipes* é permitir que programas sem nenhuma relação possam se comunicar. Os *named pipes* também são usados para sincronizar processos, já que em um determinado ponto você pode colocar um processo para "ouvir" ou para "falar" em um determinado *named pipe* e ele daí só sairá, se outro processo "falar" ou "ouvir" aquele *pipe*.

Você já viu que o uso desta ferramenta é ótimo para sincronizar processos e para fazer bloqueio em arquivos de forma a evitar perda/corrupção de informações devido a atualizações simultâneas (concorrência). Vejamos exemplos para ilustrar estes casos.

17.5. Sincronização de processos.

Suponha que você dispare paralelamente dois programas (processos) cujos diagramas de blocos de suas rotinas são como a figura a seguir:



Os dois processos são disparados em paralelo e no **BLOCO1** do **Programa1** as três classificações são disparadas da seguinte maneira:

```
for Arq in BigFile1 BigFile2 BigFile3
do
    if  sort $Arq
    then
        Manda=va
    else
        Manda=pare
        break
    fi
done
echo $Manda > pipe1
[ $Manda = pare ] &&
    {
    echo Erro durante a classificação dos arquivos
    exit 1
    }
...
```

Assim sendo, o comando **if** testa cada classificação que está sendo efetuada. Caso ocorra qualquer problema, as classificações seguintes serão abortadas, uma mensagem contendo a cadeia **pare** é enviada pelo **pipe1** e **programa1** é descontinuado com um fim anormal.

Enquanto o **Programa1** executava o seu primeiro bloco (as classificações) o **Programa2** executava o seu **BLOCO1**, processando as suas rotinas de abertura e menu paralelamente ao **Programa1**, ganhando desta forma um bom intervalo de tempo.

O fragmento de código do **Programa2** a seguir, mostra a transição do seu **BLOCO1** para o **BLOCO2**:

```
OK=`cat pipe1`
if  [ $OK = va ]
then
    ...
    Rotina de impressão
    ...
else    #  Recebeu "pare" em OK
    exit 1
fi
```

Após a execução de seu primeiro bloco, o **Programa2** passará a "ouvir" o **pipe1**, ficando parado até que as classificações do **Programa1** terminem, testando a seguir a mensagem passada pelo **pipe1** para decidir se os arquivos estão íntegros para serem impressos, ou se o programa deverá ser descontinuado. Desta forma é possível disparar programas de forma assíncrona e sincronizá-los quando necessário, ganhando bastante tempo de processamento.

17.6. Bloqueio de arquivos

Suponha que você escreveu uma **CGI** (*Common Gateway Interface*) em *Shell* para contar quantos hits recebe uma determinada URL e a rotina de contagem está da seguinte maneira:

```
    Hits="$(cat page.hits 2> /dev/null)" || Hits=0
    echo $((Hits++)) > page.hits
```

Desta forma se a página receber dois ou mais acessos concorrentes, um ou mais poderá(ão) ser perdido(s), basta que o segundo acesso seja feito após a leitura da arquivo **page.hits** e antes da sua gravação, isto é, basta que o segundo acesso seja feito após o primeiro ter executado a primeira linha do *script* e antes de executar a segunda.

Então o que fazer? Para resolver o problema de concorrência vamos utilizar um *named pipe*. Criamos o seguinte *script* que será o *daemon* que receberá todos os pedidos para incrementar o contador. Note que ele vai ser usado por qualquer página no nosso site que precise de um contador.

```
$ cat contahits.sh
#!/bin/bash
PIPE="/tmp/pipe_contador" # arquivo named pipe
# dir onde serao colocados os arquivos contadores de cada pagina
DIR="/var/www/contador"

[ -p "$PIPE" ] || mkfifo "$PIPE"

while :
do
    for URL in $(cat < $PIPE)
    do
        FILE="$DIR/$(echo $URL | sed 's,.*/,,')"
        # OBS1: no sed acima, como precisava procurar
        #       uma barra,usamos vírgula como separador.
        # OBS2: quando rodar como daemon comente a proxima linha
        echo "arquivo = $FILE"

        n="$(cat $FILE 2> /dev/null)" || n=0
        echo $((n=n+1)) > "$FILE"
    done
done
```

Como só este *script* altera os arquivos, não existe problema de concorrência.

Este *script* será um *daemon*, isto é, rodará em *background*. Quando uma página sofrer um acesso, ela escreverá a sua URL no arquivo de *pipe*. Para testar, execute este comando:

```
    echo "teste_pagina.html" > /tmp/pipe_contador
```

Para evitar erros, em cada página que quisermos adicionar o contador acrescentamos a seguinte linha:

```
<!--#exec cmd="echo $REQUEST_URI > /tmp/pipe_contador"-->
```

Note que a variável **$REQUEST_URI** contém o nome do arquivo que o navegador (*browser*) requisitou.

Este último exemplo, é fruto de uma idéia que troquei com o amigo e mestre em *Shell*, Thobias Salazar Trevisan, que escreveu o *script* e colocou-o em sua excelente URL. Aconselho a todos que querem aprender *Shell* a dar uma olhada nela (Dê uma olhada e inclua-a nos favoritos).

Ahhh! Você pensa que o assunto sobre *named pipes* está esgotado? Enganou-se. Vou mostrar um uso diferente a partir de agora.

17.7. Substituição de processos

Acabei de mostrar um monte de dicas sobre *named pipes*, agora vou mostrar que o *Shell* também usa os *named pipes* de uma maneira bastante singular, que é a substituição de processos (*process substitution*). Uma substituição de processos ocorre quando você põe um comando ou um *pipeline* de comandos entre parênteses e um **<** ou um **>** grudado na frente do parêntese da esquerda. Por exemplo, teclando-se o comando:

```
$ cat <(ls -l)
```

Resultará no comando **ls -l** executado em um *subshell* como é normal (por estar entre parênteses), porém redirecionará a saída para um *named pipe* temporário, que o *Shell* cria, nomeia e depois remove. Então o **cat** terá um nome de arquivo válido para ler (que será este *named pipe* e cujo dispositivo lógico associado é **/dev/fd/63**), e teremos a mesma saída que a gerada pela listagem do **ls -l**, porém dando um ou mais passos que o usual, isto é, mais onerosa para o computador.

Como poderemos constatar isso? Fácil... Veja o comando a seguir:

```
$ ls -l >(cat)
l-wx------ 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É... Realmente é um *named pipe*.

Você deve estar pensando que isto é uma maluquice de nerd, né? Então suponha que você tenha 2 diretórios: **dir** e **dir.bkp** e deseja saber se os dois estão iguais (aquela velha dúvida: será que meu backup está atualizado?). Basta comparar os dados dos arquivos dos diretórios com o comando **cmp**, fazendo:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) || echo backup furado
```

ou, melhor ainda:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) >/dev/null || echo backup furado
```

Da forma acima, a comparação foi efetuada em todas as linhas de todos os arquivos de ambos os diretórios. Para acelerar o processo, poderíamos comparar somente a listagem longa de ambos os diretórios, pois qualquer modificação que um arquivo sofra, é mostrada na data/hora de alteração e/ou no tamanho do arquivo. Veja como ficaria:

```
$ cmp <(ls -l dir) <(ls -l dir.bkp) >/dev/null || echo backup furado
```

Este é um exemplo meramente didático, mas são tantos os comandos que produzem mais de uma linha de saída, que serve como guia para outros. Eu quero gerar uma listagem dos meus arquivos, numerando-os e ao final dar o total de arquivos do diretório corrente:

```
while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done < <(ls)
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Tá legal, eu sei que existem outras formas de executar a mesma tarefa. Usando o comando **while**, a forma mais comum de resolver esse problema seria:

```
ls | while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Quando executasse o *script*, pareceria estar tudo certo, porém no comando **echo** após o **done**, você verá que o valor de **$i** foi perdido. Isso deve-se ao fato desta variável estar sendo incrementada em um *subshell* criado pelo *pipe* **(|)** e que terminou no comando **done**, levando com ele todas as variáveis criadas no seu interior e as alterações feitas em todas as variáveis, inclusive as criadas externamente.

Somente para te mostrar que uma variável criada fora do *subshell* e alterada em seu interior perde as alterações feitas ao seu final, execute o *script* a seguir:

```
#!/bin/bash
LIST=""                  # Criada no shell principal
ls | while read FILE     # Inicio do subshell
do
    LIST="$FILE $LIST"   # Alterada dentro do subshell
done                     # Fim do subshell
echo :$LIST:
```

Ao final da execução você verá que aparecerão apenas dois dois-pontos (**::**). Mas no início deste exemplo eu disse que era meramente didático porque existem formas melhores de fazer a mesma tarefa. Veja só estas duas:

```
$ ls | cat -n -
```

ou então, usando a própria substituição de processos:

```
$ cat -n <(ls)
```

Um último exemplo: você deseja comparar **arq1** e **arq2** usando o comando **comm**, mas este comando necessita que os arquivos estejam classificados. Então a melhor forma de proceder é:

```
$ comm <(sort arq1) <(sort arq2)
```

Esta forma evita que você faça as seguintes operações:

```
$ sort arq1 > /tmp/sort1
$ sort arq2 > /tmp/sort2
$ comm /tmp/sort1 /tmp/sort2
$ rm -f /tmp/sort1 /tmp/sort2
```

18. Automatização de tarefas com crontab e cron

Sumário

18. Automatização de tarefas com cron e crontab

18.1. O formato do arquivo crontab

18.2. Edição da crontab

18.3. Referências

18. Automatização de tarefas com cron e crontab

Um dos assuntos mais citados na enquete que conduzimos sobre assuntos possíveis para o nosso curso EAD de programação shell, depois de expressões regulares, foi a automatização de tarefas.

Sem sombra de dúvida, um complemento fundamental para shell scripts, é a possibilidade de agendamento de tarefas. Com este recurso, propiciado pelos aplicativos **cron** e **crontab**, a tarefa de administração de sistemas fica grandemente facilitada. Eu mantenho cinco portais (Dicas-L, Contando Histórias, Aprendendo Inglês, Inglês Instrumental e o portal do Instituto de Desenvolvimento do Potencial Humano). Todas as tarefas de manutenção destes portais são feitas por meio de shell scripts, com execução automatizada por meio do **cron**. O envio das mensagens, a publicação das páginas, o backup do banco de dados, a cópia de segurança de todo o portal para outro servidor, a verificação da taxa de ocupação de espaço em disco, alertas em caso de uso excessivo de CPU e muito mais. Em suma, automatizei tudo que pude, caso contrário seria impossível cuidar de tantos portais.

A automatização de tarefas em sistemas GNU/Linux é feita por meio de dois programas: **crontab** e **cron**. Além destes programas, temos também, para cada usuário, um arquivo chamado **crontab**, que é o local onde são gravadas as diretivas que serão seguidas pelo **cron**. O arquivo **crontab** é editado pelo comando **crontab**. Os arquivos **crontab** dos usuários são gravados no diretório **/var/spool/cron/crontabs**.

18.1. O formato do arquivo crontab

Uma linha do arquivo **crontab** possui o seguinte formato:

```
0 6 * * * comando
```

Os cinco primeiros campos determinam a periodicidade de execução do comando. Abaixo segue uma explicação do significado de cada um destes campos:

Campo	Significado	Valores
1	Minutos	0 a 59
2	Hora	0 a 23
3	Dia do mês	1 a 31
4	Mês	1 a 12
5	Dia da semana	0 a 7, sendo que os números 0 e 7 indicam o domingo
6	Comando a ser executado	qualquer comando válido do sistema

Exemplos

```
0  20 * * 1-5 comando
```

O comando será executado de segunda a sexta (**1-5**), exatamente às 20h.

```
10 10 1 * * comando
```

O comando será executado às 10h10, do dia 1° de todos os meses.

```
0,10,20,30,40,50, * 31 12 * comando
```

No dia 31 de dezembro, o comando será executado a cada dez minutos, o dia inteiro. Sempre que um campo for preenchido com o asterisco, significa que aquele campo assumirá todos os valores possíveis.

Podemos abreviar esta notação, em vez de especificar cada um dos minutos, podemos fazer da seguinte forma:

```
*/10 * 31 12 * comando
```

Os dias da semana podem também ser indicados por sua abreviação: **sun**, **mon**, **tue**, **wed**, **thu**, **fri**, **sat**.

Prosseguindo ...

```
* * * * * comando
```

Esta linha da crontab fará com que o comando seja executado todos os minutos, de todos os dias, de todos os meses. É apenas um exemplo, não faça isto, a não ser que tenha uma razão muito boa para tal.

Isto é o básico, mas o aplicativo **cron** oferece mais alguns recursos muito úteis. A tabela abaixo relaciona mais alguns parâmetros que podemos usar na crontab com seu significado:

Parâmetro	Descrição	Equivale a
@reboot	ocorre ao iniciar o computador	—
@yearly	ocorre 1 vez ao ano	0 0 1 1 *
@annually	o mesmo que @yearly	0 0 1 1 *
@monthly	ocorre 1 vez ao mês	0 0 1 * *
@weekly	ocorre 1 vez na semana	0 0 * * 0
@daily	Executa uma vez ao dia	0 0 * * *
@midnight	mesmo que @daily	0 0 * * *
@hourly	ocorre 1 vez a cada hora	0 * * * *

A diretiva **@reboot** é particularmente útil, pois em cada sistema nós temos tarefas que precisam ser executadas quando da inicialização da máquina e este parâmetro é um atalho muito conveniente para representar estas atividades.
Para utilizar estes parâmetros, bastam dois campos: o parâmetro em si e o nome do comando. Por exemplo, para executar o backup do banco de dados diariamente, acrescente a seguinte linha ao arquivo **crontab**:

```
@daily /usr/local/bin/db_backup.sh
```

O script **db_backup.sh** deve ser criado pelo administrador com as diretivas apropriadas para o backup do banco de dados.

Importante, se o seu computador não estiver no ar quando do horário da execução do comando, o **cron** não fará um novo agendamento, esta situação precisa ser tratada manualmente. A única exceção é a diretiva **@reboot**, que faz com que o comando especificado seja executado quanto a máquina for ligada.

18.2. Edição da crontab

Falta agora descobrir como editar a crontab, o que é bastante simples:

```
$ crontab -e
```

A diretiva **-e** indica ao programa crontab que queremos editar seu conteúdo. O arquivo crontab padrão vem com um conjunto de instruções (em inglês) sobre a sua sintaxe:

```
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
```

Todas as linhas iniciadas por **#** são comentários, e você pode removê-las quando quiser, sem problema algum.

O arquivo crontab será aberto utilizando o editor de sua preferência. Você pode definir esta preferência através da variável de ambiente **EDITOR**:

```
export EDITOR=vi
```

Se você não conhece o **vi**, pode usar o **nano**, que é um editor mais amigável, embora com muito menos recursos.

Caso deseje apenas listar o conteúdo do arquivo **crontab**, digite:

```
$ crontab -l
```

O usuário **root** pode editar diretamente a **crontab** dos usuários do sistema que administra. Para isto basta executar o comando:

```
# crontab -u [nome do usuário] -e
```

IMPORTANTE: não é só colocar o comando na crontab e esquecer da vida, precisamos gerar alertas ou verificar a saída gerada pelo arquivo para ver se tudo deu certo.

Por padrão, o **cron** envia para o usuário um email com o resultado do comando. Através da análise destas mensagens, podemos decidir o que fazer.

Caso queiramos que o email seja enviado para um outro usuário, podemos definir, no arquivo **crontab**, a variável **MAILTO**:

```
MAILTO="admin@mydomain.com"
```

E se eu errar na edição da **crontab**, corro o risco de perder alguma coisa? Fique tranquilo, sempre que você gravar a **crontab**, se houver algum erro de sintaxe, o sistema te avisa, não grava as alterações e te pergunta se você quer refazer a edição:

```
$ crontab -e

crontab: installing new crontab
"/tmp/crontab.KA6PeR/crontab":24: bad minute
errors in crontab file, can't install.
Do you want to retry the same edit? (y/n)
```

Não se esqueça, sempre coloque o caminho completo do comando, visto que a variável **PATH** da execução do comando via **cron** é diferente daquela do seu ambiente.

18.3. Referências

- Unix Crontab
- Artigos Dicas-L, por vários autores
- **man crontab**, para saber mais sobre o comando crontab
- **man 5 crontab**, para saber mais sobre o formato do arquivo **crontab**
- **man cron**, para saber mais sobre o comando **cron**