ChangeLog da 6ª Edição

Na página 23, trocar o texto:

Exemplo:

```
$ rmdir ~dirtst
```

Por:

Exemplo:

```
$ rmdir ~/dirtst
```

Na página 24, trocar o texto:

```
$ rm phpdir
rm: remove directory `phpdir'? y
rm: cannot remove directory `phpdir': Is a directory
$ rm -rf php
```

Por:

```
$ rm phpdir
rm: remove directory `phpdir'? y
rm: cannot remove directory `phpdir': Is a directory
$ rm -rf phpdir
```

Na página 28, antes de "*Exemplos*:", inserir o seguinte texto:

`-printf formato` Permite que se escolha os campos que serão listados e formata a saída de acordo com o especificado em `formato`.

Na página 29, antes da seção "basename – Devolve o nome de um arquivo", incluir o seguinte texto:

Com o `printf` é possível formatar a saída do comando `find` e especificar os dados desejados. A formatação do `printf` é muito semelhante à do mesmo comando na linguagem C e interpreta caracteres de formatação precedidos por um símbolo de percentual `(%)`. Vejamos seus efeitos sobre a formatação:

Caractere	Significado
%f	Nome do arquivo (caminho completo não aparece)
%F	Indica a qual tipo de file system o arquivo pertence
%g	Grupo ao qual o arquivo pertence
%G	Grupo ao qual o arquivo pertence (GID- Numérico)
%h	Caminho completo do arquivo (tudo menos o nome).
%i	Número do inode do arquivo (em decimal)
%m	Permissão do arquivo (em octal).
%p	Nome do arquivo
%s	Tamanho do arquivo
%u	Nome de usuário (username) do dono do arquivo
%U	Número do usuário (UID) do dono do arquivo

Também é possível formatar datas e horas obedecendo às tabelas a seguir:

Caractere	Significado
%a	Data do último acesso
%c	Data de criação
%t	Data de alteração

Os três caracteres acima produzem uma data semelhante ao do comando `date`. Veja um exemplo:

```
$ find . -name ".b*" -printf '%t %p\n'
Mon Nov 29 11:18:51 2004 ./.bash_logout
Tue Nov  1 09:44:16 2005 ./.bash_profile
Tue Nov  1 09:45:28 2005 ./.bashrc
Fri Dec 23 20:32:31 2005 ./.bash_history
```

Neste exemplo, o `%p` foi o responsável por colocar os nomes dos arquivos. Caso fosse omitido, somente as datas seriam listadas. Observe ainda que ao final foi colocado um `/n`. Sem ele não haveria salto de linha e a listagem acima seria uma grande tripa.

Estas datas também podem ser formatadas, para isso basta passar as letras da tabela anterior para maiúsculas `(%A, %C` e `%T)` e usar um dos formatadores das duas tabelas a seguir:

Tabela de formatação de tempo	
Caractere	**Significado**
`H`	Hora (00..23)
`I`	Hora (01..12)
`k`	Hora (0..23)
`l`	Hora (1..12)
`M`	Minuto (00..59)
`p`	AM or PM
`r`	Horário de 12 horas (hh:mm:ss) seguido de AM ou PM
`S`	Segundos (00 ... 61)
`T`	Horário de 24-horas (hh:mm:ss)
`Z`	Fuso horário (na Cidade Maravilhosa BRST)

Tabela de formatação de datas	
Caractere	**Significado**
`a`	Dia da semana abreviado (Dom...Sab)
`A`	Dia da semana por extenso (Domingo...Sábado)
`b`	Nome do mês abreviado (Jan...Dez)
`B`	Dia do mês por extenso (Janeiro...Dezembro)
`c`	Data e hora completa (Fri Dec 23 15:21:41 2005)
`d`	Dia do mês (01...31)
`D`	Data no formato mm/dd/aa
`h`	Idêntico a b
`j`	Dia seqüencial do ano (001…366)
`m`	Mês (01...12)
`U`	Semana seqüencial do ano. Domingo como 1º dia da semana (00...53)
`w`	Dia seqüencial da semana (0..6)
`W`	Semana seqüencial do ano. segunda-feira como 1º dia da semana (00...53)
`x`	Representação da data no formato do país (definido por $LC_ALL)
`y`	Ano com 2 dígitos (00...99)
`Y`	Ano com 4 dígitos

Para melhorar a situação, vejamos uns exemplos porém vejamos primeiro a quais são os arquivos do diretório corrente que começam por `.b`:

```
$ ls -la .b*
-rw-------    1 d276707  ssup        21419 Dec 26 17:35 .bash_history
-rw-r--r--    1 d276707  ssup           24 Nov 29  2004 .bash_logout
-rw-r--r--    1 d276707  ssup          194 Nov  1 09:44 .bash_profile
-rw-r--r--    1 d276707  ssup          142 Nov  1 09:45 .bashrc
```

Para listar estes arquivos em ordem de tamanho, podemos fazer:

```
$ find . -name ".b*" -printf '%s\t%p\n' | sort -n
24      ./.bash_logout
142     ./.bashrc
194     ./.bash_profile
21419   ./.bash_history
```

No exemplo que acabamos de ver, o `\t` foi substituído por um `<TAB>` na saída de forma a tornar a listagem mais legível.

Para listar os mesmos arquivos classificados por data e hora da última alteração:

```
$ find . -name ".b*" -printf '%TY-%Tm-%Td %TH:%TM:%TS %p\n' | sort
2004-11-29 11:18:51 ./.bash_logout
2005-11-01 09:44:16 ./.bash_profile
2005-11-01 09:45:28 ./.bashrc
2005-12-26 17:35:13 ./.bash_history
```

Na página 52 a ordem está trocada. Primeiro vem a tabela que está na pag 53 e depois o exemplo. No fim da seção e imediatamente antes do comando `cal`, inserir o seguinte texto:

Existe uma forma simples de pegar a data de ontem, lembrando que o "ontem" pode ser no mês anterior (que não sabemos *a priori* a quantidade de dias), no ano anterior ou até no século anterior. Veja só como:

```
$ date --date '1 day ago'
Thu Dec 22 12:01:45 BRST 2005
```

E para proceder ao contrário, isto é, fazer uma viagem ao futuro? Veja o exemplo:

```
$ date --date='1 day'
Sat Dec 24 12:06:29 BRST 2005
```

Como você viu é a mesma coisa, basta tirar o `ago`. E em qual dia da semana cairá o Natal deste ano? Simples:

```
$ date --date='25 Dec' +%a
Sun
```

Mas também podemos misturar alhos com bugalhos, veja só:

```
$ date --date='1 day 1 month 1 year ago'
Mon Nov 22 12:19:58 BRST 2004
$ date --date='1 day 1 month 1 year'
Wed Jan 24 12:20:13 BRST 2007
```

Na página 66, antes da seção "execução em background", inserir o seguinte texto:

Para dar uma parada em um processo use:

```
kill -STOP pid
```

ou

```
kill -19 pid
```

Logo após, para continuar sua execução, use:

```
kill -CONT pid
```

ou

```
kill -18 pid
```

Após a página 68 (imediatamente antes da parte 2 do livro) inserir o capítulo 9 com o título: "Executando tarefas agendadas" e com o seguinte conteúdo:

Aqui veremos como agendar tarefas ou executar *jobs*, visando a performance do sistema como um todo e não a do aplicativo.

Podemos agendar uma determinada tarefa para uma determinada data e horário (como um *reorg* de um banco para ser feito de madrugada) ou ciclicamente (como fazer *backup* semanal toda 6ª Feira ou o *backup* mensal sempre que for o último dia do mês).

Programando tarefas com crontab

`crontab` é o programa para instalar, desinstalar ou listar as tabelas usadas pelo programa (*daemon*) `cron` para que possamos agendar a execução de tarefas administrativas.

Cada usuário pode ter sua própria tabela de `crontab` (normalmente com o nome de `/var/spool/cron/Login_do_Usuário`) ou usar a tabela geral do *root* (normalmente em `/etc/crontab`). A diferença entre as duas formas é somente quanto às permissões de determinadas instruções que são de uso exclusivo do *root*.

Chamamos de `crontab` não só à tabela que possui o agendamento que é consultado pelo programa (*deamon*) `cron`, como também ao programa que mantém o agendamento nesta tabela, já que é ele quem edita a tabela, cria uma nova entrada ou remove outra.

Existem dois arquivos que concedem ou não permissão aos usuários para usar o `crontab`. Eles são o `/etc/cron.allow` e `/etc/cron.deny`. Todos os usuários com l*ogin name* cadastrado no primeiro estão habilitados a usá-lo e os usuários cujos *login names* constem do segundo, estão proibidos de usá-lo.

Veja só uma mensagem característica para pessoas não autorizadas a usar o `cron`:

```
$ crontab -e
You (fulano_de_tal) are not allowed to use this program (crontab)
See crontab(1) for more information
```

Para permitir que todos os usuários tenham acesso ao uso do `crontab`, devemos criar um `/etc/crontab.deny` vazio e remover o `/etc/crontab.allow`.

Com o `crontab` você tem todas as facilidades para agendar as tarefas repetitivas (normalmente tarefas do *admin*) para serem executadas qualquer dia, qualquer hora,

em um determinado momento, ... Enfim, ele provê todas as facilidades necessárias ao agendamento de execução de programas (normalmente *scripts* em *Shell*) tais como *backups*, pesquisa e eliminação de *links* quebrados, ...

As principais opções do comando `crontab` são as seguintes:

Opção	Função
`-r`	Remove o `crontab` do usuário
`-l`	Exibe o conteúdo do `crontab` do usuário
`-e`	Edita o `crontab` atual do usuário

As tabelas `crontab` têm o seguinte "leiaute":

Como pudemos ver as tabelas são definidas por seis campos separados por espaços

Campos da tabela crontab

Valores Possíveis	0-59	0-23	1-31	1-12	0-6	
Função	Minuto	Hora	Dia do Mês	Mês	Dia da Semana	Programa
Campo	1	2	3	4	5	6

em branco (ou `<TAB>`). Vejamos exemplos de linhas da tabela para entender melhor:

```
#M                                  S
#i                                  e
#n     H                            m
#u     o       D       M            a
#t     r       i       e            n
#o     a       a       s            a       Programa
#=======================================
0      0       *       *            *       backup.sh
30     2       *       *            0       bkpsemana.sh
0,30   *       *       *            *       verifica.sh
0      1       30      *            *       limpafs.sh
30     23      31      12           *       encerraano.sh
```

O jogo-da-velha, tralha ou sei-lá-mais-o-quê `(#)` indica ao `cron` que a partir daquele ponto até a linha seguinte, é tudo comentário. Por isso, é comum vermos nos `crontab` de diversas instalações, um "cabeçalho" como o que está no exemplo. Linha a linha vamos ver os significados:

1ª Linha - Todo dia às 00:00 h execute o programa `backup.sh;`

2ª Linha - Às 02:30 h de todo Domingo execute o programa `bkpsemana.sh;`

3ª Linha - Todas as horas exatas e meias horas execute o programa `verifica.sh`;

4ª Linha - Todo os dias 30 (de todos os meses) à 01:00 h execute o programa `limpafs.sh`;

5ª Linha - No dia 31/12 às 23:30 h execute o programa `encerraano.sh`.

O comando at

Este comando permite que se programe datas e horas para execução de tarefas (normalmente *scripts* em *Shell*).

O comando nos permite fazer especificações de tempo bastante complexas. Ele aceita horas no formato HH:MM para executar um *job.*

Você também pode especificar `midnight` (meia-noite), `noon` (meio-dia) ou `teatime` (hora do chá - 16:00 h) e você também pode colocar um sufixo de `AM` ou `PM` indicando se o horário estipulado é antes ou após o meio-dia.

Se o horário estipulado já passou, o *job* será executado no mesmo horário, porém no próximo dia que atenda à especificação.

Você também pode estipular a data que o *job* será executado, dando a data no formato nome-do-mês e dia, com ano opcional, ou no formato MMDDAA ou MM/DD/AA ou DD.MM.AA.

Outra forma de especificação é usando `now + unidades_de_tempo`, onde `unidades_de_tempo` pode ser: `minutes`, `hours`, `days`, ou `weeks`. Para finalizar você ainda pode sufixar um horário com as palavras `today` e `tomorrow` para especificar que o horário estipulado é para hoje ou amanhã.

Exemplos:

Para os exemplos a seguir, vejamos primeiramente a hora e a data de hoje:

```
$ date
Fri Jan  6 12:27:43 BRST 2006
```

Para executar `job.sh` às 16:00h daqui a três dias, faça:

```
$ at 4 pm + 3days
at> job.sh
at> <EOT>
job 2 at 2006-01-09 16:00
```

Calma, vou explicar! O `at` tem o seu próprio *prompt* e quando passamos o comando, ele manda o seu *prompt* para que passemos a tarefa que será executada. Para terminar a passagem de tarefas, fazemos um `<CTRL>+D`, quando o `at` nos devolve um `<EOT>` (***E**nd **O**f **T**ext*) para a tela e, na linha seguinte, o número do *job* `(job 2)` com sua programação de execução.

Para executar o mesmo *job* às 10:00h de 5 de abril, faça:

```
$ at 10am Apr 5
at> job.sh
at> <EOT>
job 3 at 2006-04-05 10:00
```

E se a execução fosse amanhã neste mesmo horário:

```
$ at now + 1 day
at> job.sh
```

```
at> <EOT>
job 4 at 2006-01-07 12:27
```

Para listar os *jobs* enfileirados para execução, faça:

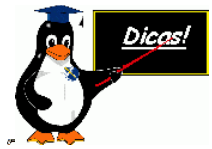
```
$ at -l
2       2006-01-09 16:00 a jneves
3       2006-04-05 10:00 a jneves
4       2006-01-07 12:27 a jneves
```

Para descontinuar o *job* número `2`, tirando-o da fila, faça:

```
$ atrm 2
$ atq
3       2006-04-05 10:00 a jneves
4       2006-01-07 12:27 a jneves
```

Repare:

1. O `atrm` removeu o *job* número `2` da fila sem a menor cerimônia. Não pediu sequer confirmação.
2. Para listar os *jobs* em execução, foi usado o `atq`, que tem exatamente o mesmo comportamento do `at -l`.



Como vimos no `crontab`, o `at` também possui os arquivos `/etc/at.allow` e `/etc/at.deny`, que servem para definir os usuários que podem enfileirar *jobs* e obedecem às mesmas regras citadas para o `/etc/cron.allow` e `/etc/cron.deny`.

O comando batch

Para quem tem que executar tarefas pesadas como uma classificação de um grande arquivo, ou um programa de cálculo de uma grande folha de pagamento, e não está afim de "sentar" o servidor, é que existe o comando batch. Os jobs que são comandados pelo batch ficam em segundo plano (background) esperando o sistema ficar "leve".

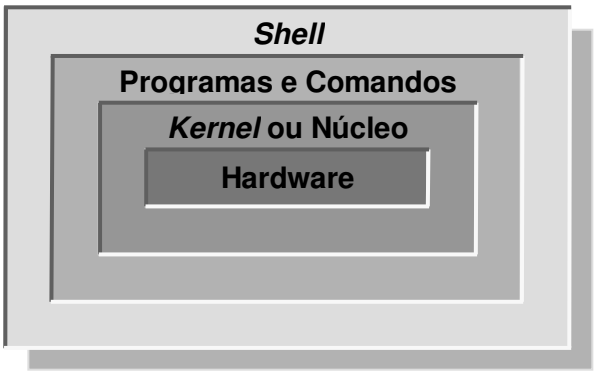
Exemplo:

```
$ batch
at> sort bigfile -o bigfile.sorted
at> <EOT>
job 8 at 2006-01-06 15:04
```

Como você viu pelo *prompt*, o `batch` é parte integrante do `at` e obedece às mesmas regras e arquivos de configuração.

No exemplo citado, o sistema imediatamente tentará iniciar o `sort`, mas só o fará efetivamente quando a carga do sistema estiver abaixo de 0.8. Caso o `sort` esteja em andamento e entre outro processo pesado, novamente ele é suspenso aguardando que a carga total baixe. A citada carga está no arquivo `/proc/loadavg`.

Trocar diagrama da página 75 por este:



Na página 77 tirar a nota de rodapé e incluir o seguinte quadro imediatamente antes da seção "Por que *Shell*?"

ATENÇÃO!

Nos exemplos a seguir, como em todos deste livro, as linhas que serão digitadas por você estão em negrito e o cifrão (**$**) no início é usado para indicar o *prompt default*.

Na pagina 126, substituir o parágrafo:

" Ué!! Não é o cifrão precedente que caracteriza uma variável? Sim, porém em todos os sabores UNIX que testei, sob bash ou ksh, ambas as formas de construção produzem uma boa aritmética. Ah, já ia me esquecendo! Os operadores usados pela expressão acima são os mesmo usados na instrução expr."

pelo seguinte texto:

Ué!! Não é o cifrão precedente que caracteriza uma variável? Sim, porém em todos os sabores *UNIX* que testei, sob *bash* ou *ksh*, ambas as formas de construção produzem uma boa aritmética.

Preste a atenção nesta seqüência:

```
$ unset i                                   $i mooooreu!
$ echo $((i++))
0
$ echo $i
1
$ echo $((++i))
2
$ echo $i
2
```

Repare que apesar da variável não estar definida, pois foi feito um `unset` nela, nenhum dos comandos acusou erro, porque como estamos usando construções aritméticas, sempre que uma variável não existe, é inicializada com zero `(0)`.

Repare que o `i++` produziu zero `(0)`. Isto ocorre porque este tipo de construção chama-se pós-incrementação, isto é, primeiramente o comando é executado e só então a variável é incrementada. No caso do `++i`, foi feita uma pré-incrementação: primeiro incrementou e somente após o comando foi executado.

Também são válidos:

```
$ echo $((i+=3))
5
$ echo $i
5
$ echo $((i*=3))
15
$ echo $i
15
$ echo $((i%=2))
1
$ echo $i
1
```

Estas 3 operações seriam o mesmo que:

```
i=$((i+3))
i=$((i*3))
i=$((i%2))
```

E isto seria válido para todos os operadores aritméticos o que em resumo produziria a tabela a seguir:

Expansão Aritmética

Expressão	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

O auge desta forma de construção com duplo parênteses é o seguinte:

```
$ echo $var
50
$ var=$((var>40 ? var-40 : var+40))
$ echo $var
10
$ var=$((var>40 ? var-40 : var+40))
$ echo $var
50
```

Este tipo de construção deve ser lido da seguinte forma: caso a variável `var` seja maior que `40 (var>40)`, então `(?)` faça `var` igual a `var` menos `40 (var-40)`, senão `(:)` faça `var` igual a `var` mais `40 (var+40)`. O que quis dizer é que os caracteres ponto-de-interrogação `(?)` e dois-pontos `(:)`, fazem o papel de "então" e "senão", servindo desta forma para montar uma operação aritmética condicional.

Da mesma forma que usamos a expressão `$((...))` para fazer operações aritméticas, também poderíamos usar a intrínseca (*built-in*) `let` ou construção do tipo `$[...]`.

Os operadores são os mesmos para estas três formas de construção, o que varia um pouco é a operação aritmética condicional com o uso do `let`. Vejamos como seria:

```
$ echo $var
50
$ let var='var>40 ? var-40 : var+40'
$ echo $var
10
$ let var='var>40 ? var-40 : var+40'
$ echo $var
50
```

Se você quiser trabalhar com bases diferentes da decimal, basta usar o formato:

```
base#numero
```

Onde `base` é um número decimal entre 2 e 64 representando o sistema de numeração, e `numero` é um número no sistema definido por `base`. Se `base#` for omitida, então 10 é assumida como *default*. Os algarismos maiores que 9 são representados por letras minúsculas, maiúsculas, `@` e `_`, nesta ordem.

Se `base` for menor ou igual a 36 maiúsculas ou minúsculas podem ser usadas indiferentemente para definir algarismos maiores que 10 (não está mal escrito, os algarismos do sistema hexadecimal, por exemplo, variam entre 0 (zero) e F). Vejamos como isso funciona:

```
$ echo $[2#11]
3
$ echo $((16#a))
10
$ echo $((16#A))
10
$ echo $((2#11 + 16#a))
13
$ echo $[64#a]
10
$ echo $[64#A]
36
$ echo $((64#@))
62
$ echo $((64#_))
63
```

Nestes exemplos usei as notações `$((...))` e `$[...]` indistintamente, para demonstrar que ambas funcionam.

Ah, já ia me esquecendo! As expressões aritméticas com os formatos `$((...))`, `$[...]` e com o comando `let`, usam os mesmo operadores usados na instrução `expr`, além dos operadores unários `(++, --, +=, *=, ...)` e condicionais que acabamos de ver.

Na página 165, trocar o parágrafo:

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

Por

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

Na página 178, antes da seção "Perguntaram ao mineiro: o que é while? while é while, uai!", inserir o texto:

Lá no início, quando falávamos sobre redirecionamento, explicamos o funcionamento do *here documents* (definido pelo símbolo `<<`, lembra?), porém não abordamos uma variante muito interessante, porque ainda não tínhamos embasamento de *Shell* para entender. Trata-se do *here strings*, que é caracterizado por três sinais de menor `(<<<)`. Sua sintaxe é a seguinte:

```
cmd <<< palavra
```

Onde `palavra` é expandida e supre a entrada do comando `cmd`. Vejamos uma aplicação prática disso usando como exemplo o *script* abaixo.

```
$ cat HereStrings
#!/bin/bash

read Var1 Var2 Var3 <<< "$@"

echo Var1 = $Var1
echo Var2 = $Var2
echo Var3 = $Var3
```

Observe a execução da "criança", passando Pêra, Uva e Maçã como parâmetros:

```
$ HereStrings Pêra Uva Maçã
Var1 = Pêra
Var2 = Uva
Var3 = Maçã
```

Inserir no fim da página 188 (após o parágrafo que começa por "tput cnorm")

- `tput flash` – Dá uma claridade intensa e rápida (*flash*) na tela para chamar a atenção.
- `tput sc (sc → save cursor position)` – Guarda a posição atual do cursor.
- `tput rc (rc → Restore cursor to position)` – Retorna o cursor para a última posição guardada pelo `sc`.

Mover o bloco que vai desde a página 230 seção "Parâmetros" até o quadro "Dicas" (antes da seção "Funções") para a página 221 antes dos "Exercícios".

Logo após a inserção do bloco acima, ainda antes dos "Exercícios", inserir o texto a seguir:

Vetores ou *Arrays*

Os *Shells* mais modernos (o que já sabemos não ser o caso do *Bourne Shell*) suportam vetores (*arrays*) unidimensionais e cada um de seus elementos deverá ser inicializado com a notação `vet[nn]=valor`, onde `vet` é o nome do vetor, `nn` seu índice e `val` é o valor que se está atribuindo àquele elemento do *array*.

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[10]}
Juliana
$ echo ${Familia[18]}

$ 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`.

Vamos criar outro vetor, usando a notação sintática do *Bash*:

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

Êpa! Por este 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, desta 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 esta notação! Caso este vetor já possuísse outros elementos definidos, os valores e os índices antigos seriam perdidos e após a atribuição só restariam os elementos recém criados.

ATENÇÃO!

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 as estas 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
```

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. Desta 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 estas 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 a arroba `(@)`, 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 `$@`.

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 este tipo de construção lhe devolve a quantidade de elementos de um vetor, e não o seu maior índice. Veja este exemplo com o *array* `Veículos`, que como vimos nos exemplos anteriores, tem o índice `[9]` em seu último elemento:

```
$ echo ${#Veiculos[*]}
3
$ echo ${#Veiculos[@]}
3
```

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

```
$ 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:

```
$ array="${Frutas[@]}"
$ echo "${array[4]}"

$ echo "$array"
abacaxi banana laranja tangerina fruta do conde fruta pão
```

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

```
$ array=("${Frutas[@]}")
```

```
$ echo "${array[4]}"
fruta do conde
$ echo "${array[5]}"
fruta pão
```

Como nós vimos na seção anterior (Construção com Parâmetros e Variáveis) os dois-pontos `(:)` servem para especificar uma zona de corte em uma variável. Relembrando:

```
$ var=0123456789
$ echo ${var:1:3}
123
$ echo ${var:3}
3456789
```

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}
banana laranja tangerina
$ echo ${Frutas[@]:4}
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 na seção "Construção com Parâmetros e Variáveis", porém adaptando as construções ao uso de vetores. Garanto-lhe que você entenderá tudo muito facilmente devido à semelhança entre tratamento de vetores e de variáveis.

Na página 230, no final da seção "para evitar trapalhadas use o trap", inserir o texto a seguir:

Se você deseja dar uma saída personalizada na sua sessão, basta colocar a linha a seguir no seu `.bash_profile` ou `.bashrc`.

```
trap 'echo -n "Fim da sessão $$ em "; date ; sleep 2' 0
```

ou:

```
trap 'echo -n "Fim da sessão $$ em "; date ; sleep 2' EXIT
```

Procedendo assim, quando as suas sessões encerrarem, você ganhará por dois segundos uma mensagem do tipo:

```
Fim da sessão 1234 em Thu Dec 29 13:49:23 BRST 2005
```

Incluir a seção a seguir na página 247 antes da seção "Fatiando opções"

script também é um comando

Atualmente está sendo dada uma grande tônica na área de segurança. O que quero mostrar agora não é sobre segurança propriamente dita, mas pode dar uma boa ajuda neste campo.

Algumas vezes o administrador está desconfiado de alguém ou é obrigado a abrir uma conta em seu computador para um consultor externo e, ciente de sua responsabilidade, fica preocupado imaginando o que esta pessoa pode estar fazendo no seu reinado.

Se for este o seu caso, existe uma saída simples e rápida para ser implementada para tirar esta pulga de trás da sua orelha. É o comando `script`, cuja função é colocar tudo o que acontece/aparece na tela em um arquivo.

Quando você entra com o comando `script`, recebe como resposta `Script started, file is typescript` para informar que a instrução está em execução e a saída da tela está sendo copiada para o arquivo `typescript`. Você ficará monitorando tudo daquela estação até que seja executado um comando `exit` ou um `<CTRL>+D` (que é representado na tela por um `exit`) naquela estação, quando então o arquivo gerado pode ser analisado.

Veja no exemplo a seguir:

```
$ script
Script started, file is typescript
$ who
d244775  pts/14       Dec 23 10:18 (10.0.133.116)
d276707  pts/17       Dec 23 11:09 (10.0.132.90)
d276707  pts/18       Dec 23 12:06 (10.0.132.90)
d276707  pts/0        Dec 23 13:47 (10.0.132.90)
$ exit
Script done, file is typescript
$ cat typescript
Script started on Fri Dec 23 13:51:16 2005
$ who
d244775  pts/14       Dec 23 10:18 (10.0.133.116)
d276707  pts/17       Dec 23 11:09 (10.0.132.90)
d276707  pts/18       Dec 23 12:06 (10.0.132.90)
d276707  pts/0        Dec 23 13:47 (10.0.132.90)
$ exit

Script done on Fri Dec 23 13:51:30 2005
```

Caso seja do seu interesse armazenar estes comandos em um arquivo que não seja o `typescript` basta executá-lo especificando o arquivo, da seguinte maneira:

```
$ script arq.cmds
```

Suponha que o `script` esteja sendo automaticamente comandado a partir do `.bash_profile` e neste caso, toda vez que o usuário se conectar ao computador, o arquivo de saída será zerado e regravado. Para evitar que os dados sejam perdidos, use o comando da seguinte maneira:

```
$ script -a arq.cmds
```

Com o uso da opção `-a`, o comando anexa (*append*) no fim de `arq.cmds` o conteúdo da nova seção, sem destruir o que os *logins* anteriores geraram.

Um outro uso bacana do comando é quando você está no telefone dando suporte a um usuário remoto e deseja acompanhar o que ele está fazendo. Para podermos fazer isso é necessário usarmos a opção `-f` (*flush*) do comando que manda em tempo real para o arquivo de saída tudo que está ocorrendo na tela do usuário para o qual você está dando suporte. Existem duas formas distintas, para pegar, também em tempo real, o conteúdo do arquivo que está sendo gerado. A mais simples é quando o usuário faz:

```
$ script -f arq.cmds
```

E, no seu terminal, você faz:

```
$ tail -f arq.cmds
```

Na outra forma, usando *named pipes*, primeiramente você deve criar um arquivo deste tipo. E logo após mandar listar seu conteúdo, da seguinte forma:

```
$ mkfifo paipe
$ cat paipe
```

O usuário deve então usar o *named pipe* `paipe` como saída do comando.

```
$ script -f paipe
```