SHELL SCRIPT
PROFISSIONAL

Aurélio Marinho Jargas

Novatec

Editor: Rubens Prates
Revisão gramatical: Maria Rita Quintella
Capa: Alex Lutkus

ISBN: 978-85-7522-152-5

Histórico de impressões:

Outubro/2011	Quarta reimpressão
Janeiro/2011	Terceira reimpressão
Março/2010	Segunda reimpressão
Outubro/2008	Primeira reimpressão
Abril/2008	Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP, Brasil
Tel.: +55 11 2959-6529
Fax: +55 11 2950-8869
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Jargas, Aurélio Marinho
Shell Script Profissional / Aurélio Marinho Jargas. -- São Paulo : Novatec Editora, 2008.

ISBN 978-85-7522-152-5

1. Shell Script (Programa de computador)
I. Título.

08-01176 CDD-005.369

Índices para catálogo sistemático:

1. Shell Script : Computadores : Programas : Processamento de dados 005.369

Prefácio

Em 1997, fiz meu primeiro script em shell.

Eu tinha 20 anos, era estagiário na Conectiva (hoje Mandriva) e ainda estava aprendendo o que era aquele tal de Linux. Hoje é tudo colorido e bonitinho, mas há uma década usar o Linux significava passar a maior parte do tempo na interface de texto, a tela preta, digitando comandos no terminal. Raramente, quando era preciso rodar algum aplicativo especial, digitava-se `startx` para iniciar o modo gráfico.

Meu trabalho era dar manutenção em servidores Linux, então o domínio da linha de comando era primordial. O Arnaldo Carvalho de Melo (acme) era meu tutor e ensinou o básico sobre o uso do terminal, comandos, opções, redirecionamento e filtragem (pipes). Mas seu ensinamento mais valioso, que até hoje reflete em minha rotina de trabalho, veio de maneira indireta: ele me forçava a ler a documentação disponível no sistema.

A rotina era sempre a mesma: o Arnaldo sentava-se e demonstrava como fazer determinada tarefa, digitando os comandos, para que eu visse e aprendesse. Porém, era como acompanhar uma corrida de Fórmula-1 no meio da reta de chegada. Ele digitava rápido, falava rápido e, com seu vasto conhecimento, transmitia muito mais informações do que minha mente leiga era capaz de absorver naqueles poucos minutos.

Quando ele voltava para sua mesa, então começava meu lento processo de assimilação. Primeiro, eu respirava fundo, sabendo que as próximas horas iriam fatigar os neurônios. Com aquele semblante murcho de quem não sabe nem por onde começar, eu dava um `history` para ver quais eram todos aqueles comandos estranhos que ele havia digitado. E eram muitos.

Para tentar entender cada comando, primeiro eu lia a sua mensagem de ajuda (`--help`) para ter uma ideia geral de sua utilidade. Depois, lia a sua man page, da primeira à última linha, para tentar aprender o que aquele comando fazia. Só então arriscava fazer alguns testes na linha de comando para ver o funcionamento. Experimentava algumas opções, porém com cautela, pensando bem antes de apertar o Enter.

Ler uma man page era uma experiência frustrante. Tudo em inglês, não tinha exemplos e o texto parecia ter sido sadicamente escrito da maneira mais enigmática

possível, confundindo em vez de ajudar. A pior de todas, com larga vantagem no nível de frustração, era a temida `man bash`, com suas intermináveis páginas que eu lia, relia, trelia e não entendia nada. Ah, meu inglês era de iniciante, o que tornava o processo ainda mais cruel.

A Internet não ajudava em quase nada, pois as fontes de informação eram escassas. Só havia algumas poucas páginas em inglês, com raros exemplos. Fóruns? Blogs? Lista de discussão? Passo-a-passo-receita-de-bolo? Esqueça. Era man page e ponto.

Mas com o tempo fui me acostumando com aquela linguagem seca e técnica das man pages e aos poucos os comandos começaram a me obedecer. Os odiosos *command not found* e *syntax error near unexpected token* tornaram-se menos frequentes. Deixei de perder arquivos importantes por ter digitado um comando errado, ou por ter usado > ao invés de >>.

Dica: Leia man pages. É chato, cansativo e confuso, mas compensa. Afinal, quem aprende a dirigir em um jipe velho, dirige qualquer carro.

Quanto mais aprendia sobre os comandos e suas opções, mais conseguia automatizar tarefas rotineiras do servidor, fazendo pequenos scripts.

Códigos simples, com poucos comandos, mas que poupavam tempo e garantiam a padronização na execução. Ainda tenho alguns deles aqui no meu `$HOME`, vejamos: mudar o endereço IP da máquina, remover usuário, instalação do MRTG, gerar arquivo de configuração do DHCPD.

Em alguns meses já eram dezenas de scripts para as mais diversas tarefas. Alguns mais importantes que outros, alguns mais complexos: becape, gravação de CDs, geração de páginas HTML, configuração de serviços, conversores, wrappers e diversas pequenas ferramentas para o controle de qualidade do Conectiva Linux. Que versátil é o shell!

Mas nem tudo era flores. Alguns scripts cresceram além de seu objetivo inicial, ficando maiores e mais complicados. Cada vez era mais difícil encontrar o lugar certo para fazer as alterações, tomando o cuidado de não estragar seu funcionamento. Outros técnicos também participavam do processo de manutenção, adicionando funcionalidades e corrigindo bugs, então era comum olhar um trecho novo do código que demorava um bom tempo até entender o que ele fazia e por que estava ali.

Era preciso amadurecer. Mais do que isso, era preciso fazer um trabalho mais profissional.

Os scripts codificados com pressa e sem muito cuidado com o alinhamento e a escolha de nomes de variáveis estavam se tornando um pesadelo de manutenção. Muito tempo era perdido em análise, até realmente saber onde e o que alterar. Os scripts de "cinco minutinhos" precisavam evoluir para programas de verdade.

Cabeçalhos informativos, código comentado, alinhamento de blocos, espaçamento, nomes descritivos para variáveis e funções, registro de mudanças, versionamento. Estes eram alguns dos componentes necessários para que os scripts confusos fossem transformados em programas de manutenção facilitada, poupando nosso tempo e, consequentemente, dinheiro.

A mudança não foi traumática, e já nos primeiros programas provou-se extremamente bem-sucedida. Era muito mais fácil trabalhar em códigos limpos e bem-comentados. Mesmo programas complexos não intimidavam tanto, pois cada bloco lógico estava visualmente separado e documentado. Bastava ler os comentários para saber em qual trecho do código era necessária a alteração. Quanta diferença!

Dica: Escreva códigos legíveis. Cada minuto adicional investido em limpeza de código e documentação compensa. Não basta saber fazer bons algoritmos, é preciso torná-los acessíveis.

Todo este processo fez-me evoluir como profissional e como programador. Primeiro, aprendi a procurar por conhecimento em vez de códigos prontos para copiar e colar. Depois, aprendi o valor imenso de um código limpo e informativo.

Então, nasceu este livro de shell.

Ao olhar para o Sumário, você verá uma lista de tecnologias e técnicas para tornar seus scripts mais poderosos. Mas não quero que você simplesmente aprenda a fazer um CGI ou consiga entender expressões regulares. Quero que você também evolua. Quero que você faça programas de verdade em vez de meros scripts toscos.

Prepare-se para uma imersão em shell. Ao ler cada capítulo, você fará um mergulho profundo no assunto, aprendendo bem os fundamentos para depois poder aplicá-los com a segurança de quem sabe o que está fazendo. Os códigos não são entregues prontos em uma bandeja. Passo a passo vamos juntos construindo os pedaços dos programas, analisando seu funcionamento, detectando fraquezas e melhorando até chegar a uma versão estável. Você não ficará perdido, pois avançaremos conforme os conceitos são aprendidos.

Imagine-se lendo o manual de seu primeiro carro zero quilômetro, sentado confortavelmente no banco do motorista, sentindo aquele cheirinho bom de carro

novo. Você não tem pressa, pode ficar a noite toda saboreando aquelas páginas, testando todos os botões daquele painel reluzente. Se o manual diz que a luz acenderá ao apertar o botão vermelho, o que você faz imediatamente? Aperta o botão e sorri quando a luz acende.

De maneira similar, estude este livro com muita calma e atenção aos detalhes. Não salte parágrafos, não leia quando estiver com pressa.

Reserve um tempo de qualidade para seus estudos. Com um computador ao alcance das mãos, teste os conceitos durante a leitura. Digite os comandos, faça variações, experimente! Brinque na linha de comando, aprendendo de maneira prazerosa.

Vamos?

Visite o site do livro (www.shellscript.com.br) para baixar os códigos-fonte de todos os programas que estudaremos a seguir.

Capítulo 4

Opções de linha de comando (-f, --foo)

Trazer mais opções e possibilidades para o usuário é algo que faz parte da evolução natural de um programa. Mas não é elegante forçar o usuário a editar o código para alterar o valor de variáveis, cada vez que precisar de um comportamento diferente. Aprenda a fazer seu programa reconhecer opções de linha de comando, curtas e longas, tornando-o mais flexível e amigável ao usuário.

Você já está bem-acostumado a usar opções para as ferramentas do sistema. É um `-i` no `grep` para ignorar a diferença das maiúsculas e minúsculas, é um `-n` no `sort` para ordenar numericamente, é um `-d` e um `-f` no `cut` para extrair campos, é um `-d` no `tr` para apagar caracteres. Usar uma opção é rápido e fácil, basta ler a man page ou o `--help` e adicioná-la no comando.

Agora, responda-me rápido: o que impede que os seus próprios programas também tenham opções de linha de comando? Não seria interessante o seu programa agir como qualquer outra ferramenta do sistema?

Neste capítulo aprenderemos a colocar opções em nossos programas. Com isso, além da melhor integração com o ambiente (sistema operacional), melhoramos a interface com o usuário, que já está acostumado a usar opções e poderá facilmente informar dados e alterar o comportamento padrão do programa.

O formato "padrão" para as opções

Não há uma convenção ou padrão internacional que force um programa a usar este ou aquele formato para ler parâmetros e argumentos do usuário. Ao longo das décadas, alguns formatos foram experimentados, porém hoje podemos constatar que a maioria usa o formato adotado pelos aplicativos GNU. Acompanhe a análise.

Em um sistema Unix, que possui aplicativos com mais idade que muitos leitores deste livro, é variada a forma que os programas esperam receber as opções de linha de comando. Uma grande parte usa o formato de opções curtas (de uma letra) com o hífen, mas não há um padrão. Veja alguns exemplos:

Formato das opções de programas no Unix

Comando	Formato	Exemplos
find	-<*palavra*>	-name,-type
ps	<*letra*>	a u w x
dd	<*palavra*>=	if=, of=, count=

Há também os aplicativos GNU, que de uma forma ou outra estão ligados à FSF (Free Software Foundation) e preferem a licença GPL. Estes são programas com código mais recente, que vieram para tentar substituir os aplicativos do Unix. Estão presentes em todos os cantos de um sistema Linux, alguns exemplares habitam no Mac e também podem ser instalados no Windows por intermédio do Cygwin. Eles seguem um padrão que não é forçado, porém recomendado e adotado pela maioria.

Formato das opções de aplicativos GNU

Formato	Exemplos	Descrição
`-<letra>`	`-h`, `-V`	Opções curtas, de uma letra
`--<palavra>`	`--help`, `--version`	Opções longas, de uma ou mais palavras

Por ser um formato mais consistente e hoje amplamente utilizado, acostume-se a usá-lo. Este é um padrão inclusive para outras linguagens de programação como Python e Perl (por meio dos módulos getopt).

Para pensar na cama: Os aplicativos GNU são considerados mais modernos e geralmente possuem mais opções e funcionalidades do que seus parentes do Unix. Essa abundância de opções pode ser benéfica ou não, dependendo do seu ponto de vista. O GNU sort deveria mesmo ter a opção `-u` sendo que já existe o `uniq` para fazer isso? E o GNU `ls` com suas mais de 80 opções, incluindo a pérola `--dereference-command-line-symlink-to-dir`, não seria um exagero? Onde fica o limite do "faça apenas UMA tarefa e a faça bem-feita"? Quantidade reflete qualidade? Evolução ou colesterol?

Opções clássicas para usar em seu programa

Há algumas opções clássicas que são usadas pela grande maioria dos programas para um propósito comum. Por exemplo, o `-h` ou `--help` é usado para mostrar uma tela de ajuda. Você já está acostumado a usar esta opção em várias ferramentas do sistema, então, quando for implementar uma tela de ajuda em seu programa, não invente! É `-h` e `--help`. Não surpreenda o usuário ou force-o a aprender novos conceitos sem necessidade.

Algumas opções estão tão presentes por todo o sistema que é considerado um abuso utilizá-las para outros propósitos que não os já estabelecidos. Algumas exceções são programas antigos do Unix que já usavam estas letras para outras funções. Mas você, como novato na vizinhança, siga as regras e faça seu programa o mais parecido com os já existentes. O uso das seguintes opções é fortemente recomendado (caso aplicável):

Opções estabelecidas

Opção curta	Opção longa	Descrição
`-h`	`--help`	Mostra informações resumidas sobre o uso do programa e sai.
`-V`	`--version`	Mostra a versão do programa e sai (V maiúsculo).
`-v`	`--verbose`	Mostra informações adicionais na saída, informando o usuário sobre o estado atual da execução.
`-q`	`--quiet`	Não mostra nada na saída, é uma execução quieta.
	`--`	Terminador de opções na linha de comando, o que vem depois dele não é considerado opção.

Há outras opções que não são um padrão global, mas que são vistas em vários aplicativos com o mesmo nome e função similar. Então, se aplicável ao seu programa, é aconselhável que você use estes nomes em vez de inventar algo novo:

Opções recomendadas

Opção curta	Opção longa	Descrição	Exemplo
`-c`	`--chars`	Algo com caracteres	`cut -c, od -c, wc -c`
`-d`	`--delimiter`	Caractere(s) usado(s) como delimitador, separador	`cut -d, paste -d`
`-f`	`--file`	Nome do arquivo a ser manipulado	`grep -f, sed -f`
`-i`	`--ignore-case`	Trata letras maiúsculas e minúsculas como iguais	`grep -i, diff -i`
`-n`	`--number`	Algo com números	`cat -n, grep -n, head -n, xargs -n`
`-o`	`--output`	Nome do arquivo de saída	`sort -o, gcc -o`
`-w`	`--word`	Algo com palavras	`grep -w, wc -w`

Como adicionar opções a um programa

Chega de teoria, não é mesmo? Está na hora de fazer a parte divertida desse negócio: programar. Vamos acompanhar passo a passo a inclusão de várias opções a um programa já existente. Por falar nele, aqui está:

usuarios.sh

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Aurélio, Novembro de 2007
#

cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Este é um programa bem simples que mostra uma listagem dos usuários do sistema, no formato "login TAB nome completo". Seu código resume-se a uma única linha com um `cut` que extrai os campos desejados do arquivo de usuários (`/etc/passwd`) e um `tr` filtra esta saída transformando todos os dois-pontos em TABs. Veja um exemplo de sua execução:

```
$ ./usuarios.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

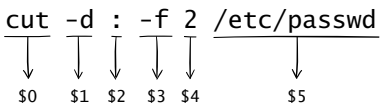
Realmente simples, não? Nas próximas páginas, esta pequena gema de uma linha crescerá para suportar opções e aumentar um pouco a sua gama de funcionalidades. Mas, claro, sem perder o foco inicial: mostrar a lista de usuários. Nada de ler e-mail ou trazer um Wiki embutido ;)

Adicionando as opções -h, -V, --help e --version

Nossa primeira tarefa será adicionar as duas opções mais clássicas de qualquer programa: a `-h` para ajuda e a `-V` para obter a versão. Alguma ideia de como fazer isso? Vou dar um tempinho para você pensar. Como pegar as opções que o usuário digitou? Como reconhecer e processar estas opções? E se o usuário passar uma opção inválida, o que acontece?

E aí, muitas ideias? Vamos começar simples, fazendo a opção `-h` para mostrar uma tela de ajuda para o programa. O primeiro passo é saber exatamente quais foram as opções que o usuário digitou, e, caso tenha sido a `-h`, processá-la.

Para saber o que o usuário digitou na linha de comando, basta conferir o valor das variáveis posicionais `$1`, `$2`, `$3` e amigos. Em `$0` fica o nome do programa, em `$1` fica o primeiro argumento, em `$2` o segundo, e assim por diante.



Parâmetros posicionais $0, $1, $2, ...

Então, se nosso programa foi chamado como `usuarios.sh -h`, a opção estará guardada em `$1`. Aí ficou fácil. Basta conferir se o valor de `$1` é `-h`, em caso afirmativo, mostramos a tela de ajuda e saímos do programa. Traduzindo isso para shell fica:

usuarios.sh (v2)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
#
# Aurélio, Novembro de 2007
#

MENSAGEM_USO="
Uso: $0 [-h]

  -h      Mostra esta tela de ajuda e sai
"

# Tratamento das opções de linha de comando
if test "$1" = "-h"
then
   echo "$MENSAGEM_USO"
   exit 0
fi

# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Lá nos cabeçalhos deixamos claro que esta é a segunda versão do programa e que ela traz de novidade a opção `-h`. A mensagem de uso é guardada em uma variável, usando o `$0` para obter o nome do programa. Isso é bom porque podemos mudar o nome do programa sem precisar nos preocupar em mudar a mensagem de ajuda junto. Em seguida, um `if` testa o conteúdo de `$1`, se for o `-h` que queremos, mostra a ajuda e sai com código de retorno zero, que significa: tudo certo.

```
$ ./usuarios-2.sh -h
Uso: ./usuarios-2.sh [-h]
-h            Mostra esta tela de ajuda e sai
```

```
$ ./usuarios-2.sh -X
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

Funciona! Quando passamos o `-h`, foi mostrada somente a mensagem de ajuda e o programa foi terminado. Já quando passamos a opção `-X`, ela foi ignorada e o programa continuou sua execução normal. Como é fácil adicionar opções a um programa em shell!

Na mensagem de ajuda, o `[-h]` entre colchetes indica que este parâmetro é opcional, ou seja, você pode usá-lo, mas não é obrigatório.

De maneira similar, vamos adicionar a opção `-V` para mostrar a versão do programa. Novamente testaremos o conteúdo de `$1`, então basta adicionar mais um teste ao `if`, desta vez procurando por `-V`:

```
# Tratamento das opções de linha de comando
if test "$1" = "-h"
then
    echo "$MENSAGEM_USO"
    exit 0
elif test "$1" = "-V"
then
    # mostra a versão
    ...
fi
```

E a cada nova opção, o `if` vai crescer mais. Mmmm, espera. Em vez de fazer um `if` monstruoso, cheio de braços, aqui é bem melhor usar o `case`. De brinde ainda ganhamos a opção "`*`" para pegar as opções inválidas. Veja como fica melhor e mais legível:

```
# Tratamento das opções de linha de comando
case "$1" in
    -h)
        echo "$MENSAGEM_USO"
        exit 0
    ;;

    -V)
        # mostra a versão
    ;;

    *)
        # opção inválida
    ;;
esac
```

Para a opção inválida é fácil, basta mostrar uma mensagem na tela informando ao usuário o erro e sair do programa com código de retorno 1. Um `echo` e um `exit` são suficientes. A versão basta mostrar uma única linha com o nome do programa e a sua versão atual, então sai com retorno zero. Mais um `echo` e um `exit`. Parece fácil.

usuarios.sh (v3)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
#
# Aurélio, Novembro de 2007
#

MENSAGEM_USO="
Uso: $0 [-h | -V]

  -h       Mostra esta tela de ajuda e sai
  -V       Mostra a versão do programa e sai
"
```

```
# Tratamento das opções de linha de comando
case "$1" in
    -h)
        echo "$MENSAGEM_USO"
        exit 0
    ;;

    -V)
        echo $0 Versão 3
        exit 0
    ;;

    *)
        echo Opção inválida: $1
        exit 1
    ;;
esac

# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Mais uma vez o cabeçalho foi atualizado para informar as mudanças. Isso deve se tornar um hábito, uma regra que não pode ser quebrada. Acostume-se desde já. Em seguida, a mensagem de uso agora mostra que o programa também possui a opção `-V`. O pipe em `[-h | -V]` informa que você pode usar a opção `-h` ou a opção `-V`, e ambas são opcionais (indicado pelos colchetes). Depois vem o `case` com cada opção bem alinhada, tornando a leitura agradável. No `-V` é mostrada a versão atual e a saída é normal. O asterisco vale para qualquer outra opção fora o `-h` e o `-V`, e além de mostrar a mensagem informando que a opção digitada é inválida, sai com código de erro (1).

```
$ ./usuarios-3.sh -h

Uso: ./usuarios-3.sh [-h | -V]

  -h            Mostra esta tela de ajuda e sai
  -V            Mostra a versão do programa e sai

$ ./usuarios-3.sh -V
./usuarios-3.sh Versão 3
$ ./usuarios-3.sh -X
Opção inválida: -X
$ ./usuarios-3.sh
Opção inválida:
$
```

Tudo funcionando! Ops, quase. Quando não foi passada nenhuma opção, o programa deveria mostrar a lista de usuários normalmente, mas acabou caindo no asterisco do `case`... Primeira melhoria a ser feita: só testar pela opção inválida se houver `$1`; caso contrário, continue.

```
    *)
        if test -n "$1"
        then
            echo Opção inválida: $1
            exit 1
        fi
    ;;
```

Outra alteração interessante seria eliminar o "./" do nome do programa tanto na opção `-h` quanto na `-V`. Ele é mostrado porque o `$0` sempre mostra o nome do programa exatamente como ele foi chamado, inclusive com `PATH`. O comando `basename` vai nos ajudar nesta hora, arrancando o `PATH` e deixando somente o nome do arquivo.

```
MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V]
...
"
```

Já que estamos aqui, com o `-h` e o `-V` funcionando, que tal também adicionar suas opções equivalentes `--help` e `--version`? Vai ser muito mais fácil do que você imagina. Lembre-se de que dentro do `case` é possível especificar mais de uma alternativa para cada bloco? Então, alterando somente duas linhas do programa ganhamos de brinde mais duas opções.

```
case "$1" in
    -h | --help)
        ...
    ;;

    -V | --version)
        ...
    ;;

    *)
        ...
    ;;
esac
```

Uma última alteração seria melhorar este esquema de mostrar a versão do programa. Aquele número ali fixo dentro do case tende a ser esquecido. Você vai modificar o programa, lançar outra versão e não vai se lembrar de aumentar o número da opção -V. Por outro lado, o número da versão está sempre lá no cabeçalho, no registro das mudanças. Será que...

```
$ grep '^# Versão ' usuarios-3.sh
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
$ grep '^# Versão ' usuarios-3.sh | tail -1
# Versão 3: Adicionado suporte à opção -V e opções inválidas
$ grep '^# Versão ' usuarios-3.sh | tail -1 | cut -d : -f 1
# Versão 3
$ grep '^# Versão ' usuarios-3.sh | tail -1 | cut -d : -f 1 | tr -d \#
 Versão 3
$
```

Ei, isso foi legal! Além de extrair a versão do programa automaticamente, ainda nos forçamos a sempre registrar nos cabeçalhos o que mudou na versão nova, para não quebrar o -V. Simples e eficiente. Vejamos como ficou esta versão nova do código.

usuarios.sh (v4)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
#
# Aurélio, Novembro de 2007
#

MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V]
```

```
  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai
"

# Tratamento das opções de linha de comando
case "$1" in
   -h | --help)
      echo "$MENSAGEM_USO"
      exit 0
   ;;

   -V | --version)
      echo -n $(basename "$0")
      # Extrai a versão diretamente dos cabeçalhos do programa
      grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#
      exit 0
   ;;

   *)
      if test -n "$1"
      then
         echo Opção inválida: $1
         exit 1
      fi
   ;;
esac

# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Agora, sim, podemos considerar que o programa ficou estável após a adição de quatro opções de linha de comando, além do tratamento de opções desconhecidas. Antes de continuar adicionando opções novas, é bom conferir se está mesmo tudo em ordem com o código atual.

```
$ ./usuarios-4.sh -h

Uso: usuarios-4.sh [-h | -V]

  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai
```

```
$ ./usuarios-4.sh --help

Uso: usuarios-4.sh [-h | -V]

  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai

$ ./usuarios-4.sh -V
usuarios-4.sh Versão 4
$ ./usuarios-4.sh --version
usuarios-4.sh Versão 4
$ ./usuarios-4.sh --foo
Opção inválida: --foo
$ ./usuarios-4.sh -X
Opção inválida: -X
$ ./usuarios-4.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

Adicionando opções específicas do programa

Colocar as opções clássicas no programa foi fácil, não? Um `case` toma conta de tudo, cada opção tendo seu próprio cantinho dentro do programa. Faz o que tem que fazer e termina com um `exit` de zero ou um para informar se está tudo bem. Se o fluxo de execução não entrou em nenhum destes cantinhos, o programa segue sua rota normal, mostrando a lista de usuários do sistema.

Agora vamos adicionar algumas opções que possuem um comportamento diferente. Elas vão ter seu cantinho dentro do `case`, mas em vez de terminar a execução do programa, vão definir variáveis e no final a lista de usuários também deverá ser mostrada. Isso vai requerer uma alteração estrutural em nosso programa. Mas vamos com calma.

Primeiro, o mais importante é avaliar: que tipo de opção seria útil adicionar ao nosso programa? Como programadores talentosos e criativos que somos, podemos adicionar qualquer coisa ao código, o céu é o limite. Mas será que uma enorme quantidade de opções reflete qualidade? Adicionar toda e qualquer opção que vier à mente é realmente benéfico ao programa e aos usuários? Avalie o seguinte:

- A opção `--FOO` vai trazer benefícios à maioria dos usuários ou apenas a um grupo muito seleto de usuários avançados? Estes usuários avançados já não conseguem se virar sem esta opção?
- A opção `--FOO` vai trazer muita complexidade ao código atual? Sua implementação será muito custosa ao programador?
- A opção `--FOO` vai influir no funcionamento da opção `--BAR` já existente?
- A opção `--FOO` está dentro do escopo do programa? Ela não vai descaracterizar seu programa?
- A opção `--FOO` é auto-explicável? Ou é preciso um parágrafo inteiro para descrever o que ela faz? Se está difícil dar um nome, pode ser um sintoma que ela nem deveria existir em primeiro lugar...
- A opção `--FOO` é realmente necessária? Não é possível fazer a mesma tarefa usando uma combinação das opções já existentes?
- A opção `--FOO` é realmente necessária? Não seria ela apenas "cosmética", não adicionando nenhum real valor ao seu programa?
- A opção `--FOO` é **REALMENTE** necessária? :)

A ênfase na real necessidade de se colocar uma opção é justificada pela tendência que nós, programadores, temos de ir adicionando funcionalidades no código, sem pensar muito nelas. Afinal, programar é divertido! É muito melhor ficar horas programando novidades do que "perder tempo" avaliando se aquilo realmente será útil.

Essa tendência leva a transformar em um monstro indomável aquele programinha rápido e eficiente das primeiras versões. No início, o programa tinha poucas opções, mas as executava instantaneamente e com perfeição. Com o passar do tempo, muitas opções foram adicionadas e hoje ele demora para dar uma resposta, às vezes interrompendo sua execução sem aviso prévio. Deu pau!

Para não ser apenas mais um personagem dessa história que se repete diariamente, faça um favor a si mesmo: leia a lista anterior toda vez que pensar em adicionar uma opção nova ao seu programa. Pense, analise, avalie, questione. Se a opção passar por todas estas barreiras e provar ser realmente útil, implemente-a.

Cara chato, né? Mas se não tiver um chato para te ensinar as coisas chatas (porém importantes), como você vai evoluir como programador? A chatice de hoje é a qualidade de amanhã em seu trabalho. Invista no seu potencial e colha os frutos no futuro!

Agora que acabaram os conselhos da terceira-idade, podemos continuar :)

Vamos decidir quais opções incluir em nosso programa. Não seremos tão rigorosos quanto à utilidade das opções, visto que o objetivo aqui é ser didático. Mas no seu programa, já sabe... Vejamos, o que o `usuarios.sh` faz é listar os usuários do sistema. Quais variações desta listagem poderiam ser interessantes ao usuário? Talvez uma opção para que a listagem apareça ordenada alfabeticamente? Isso pode ser útil.

```
$ ./usuarios-4.sh | sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
$
```

Sabemos que existe o comando sort e que ele ordena as linhas. Mas o usuário não é obrigado a saber disso. Ele no máximo lerá a nossa tela de ajuda e ali saberá das possibilidades de uso do programa. Então, apesar de ter uma implementação trivial, o usuário se beneficiará com esta opção. Nosso programa terá duas opções novas `-s` e `--sort` que serão equivalentes e servirão para que a lista seja ordenada.

Sim, poderia ser `-o` e `--ordena` para que as opções ficassem em português. Mas como já temos `--help` e `--version` em inglês, é preferível manter o padrão do que misturar os idiomas. Outra opção seria deixar tudo em português alterando as opções atuais para `--ajuda` e `--versao` (sem acentos para evitar problemas!). Avalie o perfil de seus usuários e decida qual idioma utilizar.

Para adicionar esta opção nova temos que incluí-la na mensagem de ajuda, e também dentro do case. Alguma variável global será necessária para indicar se a saída será ou não ordenada, e esta opção mudará o valor desta variável. Um esqueleto deste esquema seria assim:

```
ordenar=0                # A saída deverá ser ordenada?

...

# Tratamento das opções de linha de comando
case "$1" in

   -s | --sort)
       ordenar=1
   ;;

   ...
esac

...

if test "$ordenar" = 1
then
   # ordena a listagem
fi
```

No início do programa desligamos a chave de ordenação (`$ordenar`), colocando um valor padrão zero. Então são processadas as opções de linha de comando dentro do case. Caso o usuário tenha passado a opção `--sort`, a chave é ligada. Lá no final do programa há um `if` que testa o valor da chave, se ela estiver ligada a listagem é ordenada.

Este é o jeito limpo de implementar uma opção nova. Tudo o que ela faz é mudar o valor de uma variável de nosso programa. Lá na frente o programa sabe o que fazer com essa variável. O efeito é exatamente o mesmo de o usuário editar o programa e mudar o valor padrão da chave (`ordenar=1`) no início. A vantagem em se usar uma opção na linha de comando é evitar que o usuário edite o código, pois assim corre o risco de bagunçá-lo.

Para implementar a ordenação em si, a primeira ideia que vem à cabeça é fazer tudo dentro do `if`. Bastaria repetir a linha do `cut | tr` colocando um `sort` no final e pronto, ambas as possibilidades estariam satisfeitas. Acompanhe:

```
if test "$ordenar" = 1
then
   cut -d : -f 1,5 /etc/passwd | tr : \\t | sort
else
   cut -d : -f 1,5 /etc/passwd | tr : \\t
fi
```

Mas isso traz vários problemas. O primeiro é a repetição de código, o que nunca é benéfico. Serão dois lugares para mudar caso alguma alteração precise ser feita na dupla `cut | tr`. Esta solução também não irá ajudar quando precisarmos adicionar outras opções que também manipulem a listagem. O melhor é fazer cada tarefa isoladamente, para que a independência entre elas garanta que todas funcionem simultaneamente: primeiro extrai a lista e guarda em uma variável. Depois ordena, se necessário.

```
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)

# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
   lista=$(echo "$lista" | sort)
fi

# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

Assim o código fica maior, porém muito mais flexível e poderoso. A vantagem da separação das tarefas ficará evidente quando adicionarmos mais opções ao programa. Já são muitas mudanças, hora de lançar uma versão nova:

usuarios.sh (v5)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
```

```
# Versão 5: Adicionadas opções -s e --sort
#
# Aurélio, Novembro de 2007
#

ordenar=0            # A saída deverá ser ordenada?

MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V | -s]

  -s, --sort         Ordena a listagem alfabeticamente
  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
"

# Tratamento das opções de linha de comando
case "$1" in

   -s | --sort)
      ordenar=1
   ;;

   -h | --help)
      echo "$MENSAGEM_USO"
      exit 0
   ;;

   -V | --version)
      echo -n $(basename "$0")
      # Extrai a versão diretamente dos cabeçalhos do programa
      grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#
      exit 0
   ;;

   *)
      if test -n "$1"
      then
         echo Opção inválida: $1
         exit 1
      fi
   ;;
esac
```

```
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)

# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
   lista=$(echo "$lista" | sort)
fi

# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

O código está simples de entender, é certo que a opção nova vai funcionar. Mas não custa testar, são apenas alguns segundos investidos. E vá que aparece algum bug alienígena que nossos olhos terráqueos não consigam captar?

```
$ ./usuarios-5.sh --sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
$
```

Ufa, não foi desta vez que os ETs tiraram o nosso sono... Agora que o código do programa está com uma boa estrutura, fica fácil adicionar duas opções novas: uma para inverter a ordem da lista e outra para mostrar a saída em letras maiúsculas. Digamos `--reverse` e `--uppercase`. Como programadores experientes em shell, sabemos que o `tac` inverte as linhas de um texto e que o `tr` pode converter um texto para maiúsculas. Assim, o código para suportar estas opções novas será tão trivial quanto o do `--sort`.

```
$ ./usuarios-5.sh --sort | tac
www     World Wide Web Server
uucp    Unix to Unix Copy Protocol
unknown Unknown User
tokend  Token Daemon
sshd    sshd Privilege separation
root    System Administrator
qtss    QuickTime Streaming Server
postfix Postfix User
mysql   MySQL Server
mailman Mailman user
lp      Printing Services
jabber  Jabber User
eppc    Apple Events User
daemon  System Services
amavisd Amavisd User

$ ./usuarios-5.sh --sort | tac | tr a-z A-Z
WWW     WORLD WIDE WEB SERVER
UUCP    UNIX TO UNIX COPY PROTOCOL
UNKNOWN UNKNOWN USER
TOKEND  TOKEN DAEMON
SSHD    SSHD PRIVILEGE SEPARATION
ROOT    SYSTEM ADMINISTRATOR
QTSS    QUICKTIME STREAMING SERVER
POSTFIX POSTFIX USER
MYSQL   MYSQL SERVER
MAILMAN MAILMAN USER
LP      PRINTING SERVICES
JABBER  JABBER USER
EPPC    APPLE EVENTS USER
DAEMON  SYSTEM SERVICES
AMAVISD AMAVISD USER
$
```

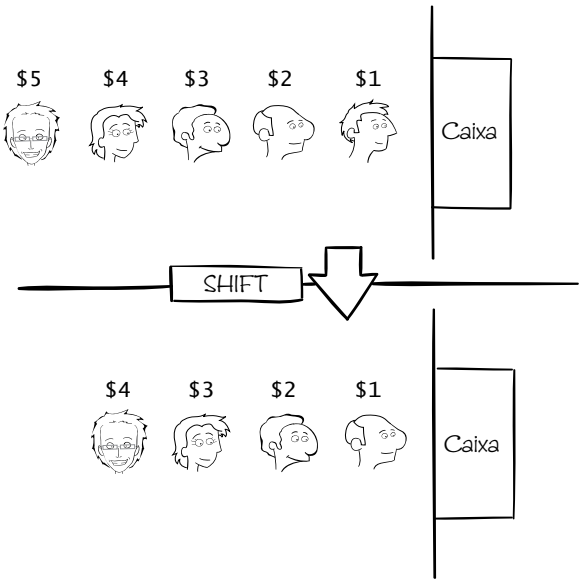
Ah, como é bom programar em shell e ter à mão todas estas ferramentas já prontas que fazem todo o trabalho sujo, não é mesmo? Um único detalhe que está faltando no código de nosso programa é que ele sempre verifica o valor de $1, não estando pronto para receber múltiplas opções. E agora ele precisará disso, pois o usuário pode querer usar todas ao mesmo tempo:

```
usuarios.sh --sort --reverse --uppercase
```

Volte algumas páginas e analise o código-fonte do programa. O que precisa ser feito para que `$2`, `$3` e outros também sejam interpretados pelo `case`? Como processar um número variável de opções de linha de comando?

O segredo da resposta está no comando `shift`. Ele remove o `$1`, fazendo com que todos os parâmetros posicionais andem uma posição na fila. Assim o `$2` vira `$1`, o `$3` vira `$2`, e assim por diante.

Imagine uma fila de banco onde você é o quinto (`$5`) cliente. Quando o primeiro da fila for atendido, a fila anda (`shift`) e você será o quarto (`$4`). Depois o terceiro, e assim por diante, até você ser o primeiro ($1) para finalmente ser atendido. Essa é a maneira shell de fazer loop nos parâmetros posicionais: somente o primeiro da fila é atendido, enquanto os outros esperam. Em outras palavras: lidaremos sempre com o `$1`, usando o `shift` para fazer a fila andar.



Funcionamento do comando shift

Traduzindo este comportamento para códigos, teremos um loop `while` que monitorará o valor de `$1`. Enquanto esta variável não for nula, temos opções de linha de comando para processar. Dentro do loop fica o `case` que já conhecemos. Ele já sabe consultar o valor de `$1` e tomar suas ações. Ele é como se fosse o caixa do banco. Uma vez processada a opção da vez, ela é liberada para que a fila ande (`shift`) e o loop continue até não haver mais o `$1`:

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
```

```
    case "$1" in

        -s | --sort)
            ordenar=1
        ;;

        ...
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done
```

Veja como ficou a versão nova do código com este loop implementado, bem como as opções novas `--reverse` e `--uppercase`. Perceba também que a tela de ajuda mudou um pouco, usando o formato `[OPÇÕES]` para simplificar a sintaxe de uso, indicando que todas as opções seguintes não são obrigatórias (colchetes). Outra mudança dentro do `case`, foi a retirada do teste de existência do parâmetro `$1`, que era feito na opção padrão (asterisco). Ele não é mais necessário visto que o `while` já está fazendo esta verificação.

usuarios.sh (v6)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
# Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
#           leitura de múltiplas opções (loop)
#
# Aurélio, Novembro de 2007
#

ordenar=0           # A saída deverá ser ordenada?
inverter=0          # A saída deverá ser invertida?
maiusculas=0        # A saída deverá ser em maiúsculas?
```

```
MENSAGEM_USO="
Uso: $(basename "$0") [OPÇÕES]

OPÇÕES:
  -r, --reverse      Inverte a listagem
  -s, --sort         Ordena a listagem alfabeticamente
  -u, --uppercase    Mostra a listagem em MAIÚSCULAS

  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
"

# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in

        -s | --sort)
            ordenar=1
        ;;

        -r | --reverse)
            inverter=1
        ;;

        -u | --uppercase)
            maiusculas=1
        ;;

        -h | --help)
            echo "$MENSAGEM_USO"
            exit 0
        ;;

        -V | --version)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#
            exit 0
        ;;
```

```
        *)
            echo Opção inválida: $1
            exit 1
        ;;
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done

# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)

# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
    lista=$(echo "$lista" | sort)
fi

# Inverte a listagem (se necessário)
if test "$inverter" = 1
then
    lista=$(echo "$lista" | tac)
fi

# Converte para maiúsculas (se necessário)
if test "$maiusculas" = 1
then
    lista=$(echo "$lista" | tr a-z A-Z)
fi

# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

Antes de passar para a próxima opção a ser incluída, cabe aqui uma otimização de código que melhorará a sua legibilidade. Alteraremos dois trechos, sem incluir nenhuma funcionalidade nova, apenas o código será reformatado para ficar mais compacto e, ao mesmo tempo, mais legível. Isso não acontece sempre, geralmente compactar significa tornar ruim de ler. Mas como neste caso as linhas são praticamente iguais, mudando apenas uma ou outra palavra, o alinhamento beneficia a leitura:

Código atual	Código compacto
`case "$1" in` `  -s | --sort)` `  ordenar=1` `  ;;` `  -r | --reverse)` `  inverter=1` `  ;;` `  -u | --uppercase)` `  maiusculas=1` `  ;;` `  ...` `esac`	`case "$1" in` `  # Opções que ligam/desligam chaves` `  -s | --sort  ) ordenar=1  ;;` `  -r | --reverse  ) inverter=1  ;;` `  -u | --uppercase) maiusculas=1 ;;` `  ...` `esac`
`# Ordena a listagem (se necessário)` `if test "$ordenar" = 1` `then` `  lista=$(echo "$lista" \| sort)` `fi` `# Inverte a listagem (se necessário)` `if test "$inverter" = 1` `then` `  lista=$(echo "$lista" | tac)` `fi` `# Converte para maiúsculas (se nece...` `if test "$maiusculas" = 1` `then` `  lista=$(echo "$lista" | tr a-z A-Z)` `fi`	`# Ordena, inverte ou converte para maiúsculas (se necessário)` `test "$ordenar"  = 1 && lista=$(echo "$lista" \| sort)` `test "$inverter"  = 1 && lista=$(echo "$lista" | tac)` `test "$maiusculas" = 1 && lista=$(echo "$lista" | tr a-z A-Z)`

Adicionando opções com argumentos

Até agora vimos opções que funcionam como chaves que ligam funcionalidades. Elas não possuem argumentos, sua presença basta para indicar que tal funcionalidade deve ser ligada (ou em alguns casos, desligada).

A última opção que adicionaremos ao nosso programa será diferente. Ela se chamará `--delimiter`, e assim como no `cut`, esta opção indicará qual caractere será usado como delimitador. Em nosso contexto, isso se aplica ao separador entre o login e o nome completo do usuário, que hoje está fixo no TAB. Veja a diferença na saída do programa, com esta opção nova:

```
$ ./usuarios-7.sh | grep root
root    System Administrator
$ ./usuarios-7.sh --delimiter , | grep root
root,System Administrator
$
```

Assim o usuário ganha flexibilidade no formato de saída, podendo adaptar o texto ao seu gosto. A implementação da funcionalidade é tranquila, basta usar uma nova variável que guardará o delimitador. Mas e dentro do `case`, como fazer para obter o argumento da opção?

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in

        -d | --delimiter)
            shift
            delim="$1"
        ;;

        ...
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done
```

Primeiro é feito um `shift` "manual" para que a opção `-d` (ou `--delimiter`) seja descartada, fazendo a fila andar e assim seu argumento fica sendo o `$1`, que é então salvo em `$delim`. Simples, não? Sempre que tiver uma opção que tenha argumentos, use esta técnica. Ah, mas e se o usuário não passou nenhum argumento, como em `usuarios.sh -d`?

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in

        -d | --delimiter)
```

```
            shift
            delim="$1"

            if test -z "$delim"
            then
               echo "Faltou o argumento para a -d"
               exit 1
            fi
        ;;

        ...
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done
```

Agora sim, a exceção foi tratada e o usuário informado sobre seu erro. Chega, né? Quase 100 linhas está bom demais para um programa que inicialmente tinha apenas 10. Mas, em compensação, agora ele possui 12 opções novas (seis curtas e suas alternativas longas) e muita flexibilidade de modificação de sua execução, além de estar mais amigável com o usuário por ter uma tela de ajuda. Veja como ficou a última versão:

usuarios.sh (v7)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
# Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
#           leitura de múltiplas opções (loop)
# Versão 7: Melhorias no código para que fique mais legível,
#           adicionadas opções -d e --delimiter
```

```
#
# Aurélio, Novembro de 2007
#

ordenar=0           # A saída deverá ser ordenada?
inverter=0          # A saída deverá ser invertida?
maiusculas=0        # A saída deverá ser em maiúsculas?
delim='\t'          # Caractere usado como delimitador de saída

MENSAGEM_USO="
Uso: $(basename "$0") [OPÇÕES]

OPÇÕES:
  -d, --delimiter C  Usa o caractere C como delimitador
  -r, --reverse      Inverte a listagem
  -s, --sort         Ordena a listagem alfabeticamente
  -u, --uppercase    Mostra a listagem em MAIÚSCULAS

  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
"

# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in

        # Opções que ligam/desligam chaves
        -s | --sort     ) ordenar=1    ;;
        -r | --reverse  ) inverter=1   ;;
        -u | --uppercase) maiusculas=1 ;;

        -d | --delimiter)
            shift
            delim="$1"

            if test -z "$delim"
            then
                echo "Faltou o argumento para a -d"
                exit 1
            fi
        ;;
```

```
        -h | --help)
            echo "$MENSAGEM_USO"
            exit 0
        ;;

        -V | --version)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#
            exit 0
        ;;

        *)
            echo Opção inválida: $1
            exit 1
        ;;
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done

# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)

# Ordena, inverte ou converte para maiúsculas (se necessário)
test "$ordenar"    = 1 && lista=$(echo "$lista" | sort)
test "$inverter"   = 1 && lista=$(echo "$lista" | tac)
test "$maiusculas" = 1 && lista=$(echo "$lista" | tr a-z A-Z)

# Mostra o resultado para o usuário
echo "$lista" | tr : "$delim"
```

Ufa, como cresceu! Você pode usar este programa como modelo para seus próprios programas que terão opções de linha de comando, toda essa estrutura é reaproveitável. Para fechar de vez com chave de ouro, agora faremos o teste final de funcionamento, usando todas as opções, inclusive misturando-as. Espero que não apareça nenhum bug :)

```
$ ./usuarios-7.sh --help

Uso: usuarios-7.sh [OPÇÕES]
```

```
OPÇÕES:
  -d, --delimiter C  Usa o caractere C como delimitador
  -r, --reverse      Inverte a listagem
  -s, --sort         Ordena a listagem alfabeticamente
  -u, --uppercase    Mostra a listagem em MAIÚSCULAS
  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
```

```
$ ./usuarios-7.sh -V
usuarios-7.sh Versão 7
$ ./usuarios-7.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$ ./usuarios-7.sh --sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
```

```
$ ./usuarios-7.sh --sort --reverse
www     World Wide Web Server
uucp    Unix to Unix Copy Protocol
unknown Unknown User
tokend  Token Daemon
sshd    sshd Privilege separation
root    System Administrator
qtss    QuickTime Streaming Server
postfix Postfix User
mysql   MySQL Server
mailman Mailman user
lp      Printing Services
jabber  Jabber User
eppc    Apple Events User
daemon  System Services
amavisd Amavisd User
$ ./usuarios-7.sh --sort --reverse --uppercase
WWW     WORLD WIDE WEB SERVER
UUCP    UNIX TO UNIX COPY PROTOCOL
UNKNOWN UNKNOWN USER
TOKEND  TOKEN DAEMON
SSHD    SSHD PRIVILEGE SEPARATION
ROOT    SYSTEM ADMINISTRATOR
QTSS    QUICKTIME STREAMING SERVER
POSTFIX POSTFIX USER
MYSQL   MYSQL SERVER
MAILMAN MAILMAN USER
LP      PRINTING SERVICES
JABBER  JABBER USER
EPPC    APPLE EVENTS USER
DAEMON  SYSTEM SERVICES
AMAVISD AMAVISD USER
$ ./usuarios-7.sh -s -d ,
amavisd,Amavisd User
daemon,System Services
eppc,Apple Events User
jabber,Jabber User
lp,Printing Services
mailman,Mailman user
mysql,MySQL Server
postfix,Postfix User
```

```
qtss,QuickTime Streaming Server
root,System Administrator
sshd,sshd Privilege separation
tokend,Token Daemon
unknown,Unknown User
uucp,Unix to Unix Copy Protocol
www,World Wide Web Server
$
```

Como (e quando) usar o getopts

Nas páginas anteriores vimos em detalhes como processar opções e argumentos informados pelo usuário na linha de comando. Vimos também que não existe um padrão, mas o uso da maioria indica um padrão estabelecido na prática. Tentando preencher estas duas lacunas (padronização e processamento), o Bash criou um comando interno (builtin) chamado de `getopts`.

O comando `getopts` serve para processar opções de linha de comando. Ele toma conta do procedimento de identificar e extrair cada opção informada, além de fazer as duas verificações básicas de erro: quando o usuário digita uma opção inválida e quando uma opção vem sem um argumento obrigatório.

Por ser rápido e lidar com a parte mais chata da tarefa de lidar com opções, o getopts pode ser uma boa escolha para programas simples que desejam usar apenas opções curtas. Porém, a falta de suporte às --opções-longas é sua maior limitação, impossibilitando seu uso em programas que precisam delas.

Prós e contras do getopts

getopts	"na mão"
Funciona somente no Bash	Funciona em qualquer shell
Somente opções curtas (`-h`)	Opções curtas e longas (`-h`, `--help`)
Opções curtas juntas ou separadas (`-abc` e `-a -b -c`)	Opções curtas somente separadas (`-a -b -c`)
Verificação automática de argumento nulo	Verificação de argumentos para opções é feita manualmente
As opções válidas devem ser registradas em dois lugares	As opções válidas são registradas somente em um lugar
Loop nas opções é feito automaticamente	Loop nas opções é feito manualmente com o `shift`

Se você já tem um programa e quer incluir rapidamente o suporte a algumas opções de linha de comando, comece com o getopts e opções curtas que é a solução mais rápida. Depois se você julgar importante incluir também as opções longas, mude para a solução caseira.

O `getopts` é um comando diferente, que leva um certo tempo para acostumar-se com os seus detalhes. Para começar, veremos um exemplo com muitos comentários, para que você tenha uma visão geral de seu uso. Leia com atenção:

getopts-teste.sh

```
#!/bin/bash
# getopts-teste.sh
#
# Aurélio, Novembro de 2007

# Loop que processa todas as opções da linha de comando.
# Atenção ao formato das opções válidas ":sa:"
# - Os dois-pontos do início ligam o modo silencioso
# - As opções válidas são 'sa:', que são -s e -a
# - Os dois-pontos de 'a:' representam um argumento: -a FOO
#
while getopts ":sa:" opcao
do
    # $opcao guarda a opção da vez (ou ? e : em caso de erro)
    # $OPTARG guarda o argumento da opção (se houver)
    #
    case $opcao in
         s) echo "OK Opção simples (-s)";;
         a) echo "OK Opção com argumento (-a), recebeu: $OPTARG";;
        \?) echo "ERRO Opção inválida: $OPTARG";;
         :) echo "ERRO Faltou argumento para: $OPTARG";;
    esac
done

# O loop termina quando nenhuma opção for encontrada.
# Mas ainda podem existir argumentos, como um nome de arquivo.
# A variável $OPTIND guarda o índice do resto da linha de
# comando, útil para arrancar as opções já processadas.
#
echo
shift $((OPTIND - 1))
echo "INDICE: $OPTIND"
echo "RESTO: $*"
echo
```

Antes de entrarmos nos detalhes do código, veja alguns exemplos de sua execução, que ajudarão a compreender o que o `getopts` faz. No primeiro exemplo foram passadas duas opções (uma simples e outra com argumento) e um argumento solto, que é um nome de arquivo qualquer. No segundo exemplo temos apenas o nome de arquivo, sem opções. Por fim, no terceiro fizemos tudo errado :)

```
$ ./getopts-teste.sh -a FOO -s /tmp/arquivo.txt
OK Opção com argumento (-a), recebeu: FOO
OK Opção simples (-s)

INDICE: 4
RESTO: /tmp/arquivo.txt

$ ./getopts-teste.sh /tmp/arquivo.txt

INDICE: 1
RESTO: /tmp/arquivo.txt

$ ./getopts-teste.sh -z -a
ERRO Opção inválida: z
ERRO Faltou argumento para: a

INDICE: 3
RESTO:

$
```

Entendeu? Funcionar ele funciona, mas tem uma série de detalhes que são importantes e devem ser respeitados. Estes são os passos necessário para usar o `getopts` corretamente:

- Colocar o `getopts` na chamada de um `while`, para que ele funcione em um loop.
- O primeiro argumento para o `getopts` é uma string que lista todas as opções válidas para o programa:
  - Se o primeiro caractere desta string for dois-pontos ":", o `getopts` funcionará em modo silencioso. Neste modo, as mensagens de erro não são mostradas na tela. Isso é muito útil para nós, pois elas são em inglês. Outra vantagem deste modo é no tratamento de erros, possibilitando diferenciar quando é uma opção inválida e quando falta um argumento. **Use sempre este modo silencioso**.

    - Liste as letras todas grudadas, sem espaços em branco. Lembre-se que o getopts só entende opções curtas, de uma letra. Por exemplo, "`hVs`" para permitir as opções `-h`, `-V` e `-s`.

    - Se uma opção requer argumento (como a `-d` do nosso `usuarios.sh`), coloque dois-pontos logo após. Por exempo, em "`hVd:sru`" somente a opção `-d` precisa receber argumento.

- O segundo argumento do `getopts` é o nome da variável na qual ele irá guardar o nome da opção atual, útil para usar no `case`. Aqui você pode usar o nome que preferir.

- Dentro do `case`, liste todas as opções válidas. Note que não é necessário o hífen! Se a opção requer um argumento, ele estará guardado na variável `$OPTARG`. Trate também as duas alternativas especiais que o `getopts` usa em caso de erro:

    - Uma interrogação é utilizada quando o usuário digita uma opção inválida. A opção digitada estará dentro de `$OPTARG`.

    - Os dois-pontos são utilizados quando faltou informar um argumento obrigatório para uma opção. A opção em questão estará dentro de `$OPTARG`.

- Nenhum `shift` é necessário, pois o `getopts` cuida de todo o processo de avaliação de cada opção da linha de comando.

- Por fim, se processadas todas as opções e ainda sobrar algum argumento, ele também pode ser obtido. Isso geralmente acontece quando o aplicativo aceita receber um nome de arquivo como último argumento, assim como o `grep`. A variável `$OPTIND` guarda o índice com a posição deste argumento. Como podem ser vários argumentos restantes, o mais fácil é usar um `shift N` para apagar todas as opções, sobrando apenas os argumentos. Onde `N = $OPTIND - 1`.

Como você percebeu, são vários detalhes, o `getopts` não é um comando trivial. Mas você se acostuma. E usado uma vez, você pode aproveitar a mesma estrutura para outro programa. Analise o exemplo que acabamos de ver, digite-o, faça seus testes. O `getopts` facilita sua vida, mas primeiro você deve ficar mais íntimo dele. Segue uma tabelinha bacana que tenta resumir todos estes detalhes:

Resumo das regras do getopts

while getopts OPÇÕES VAR; do ... done	
OPÇÕES (string definida pelo programador)	
a	Opção simples: -a
a:	Opção com argumento: -a FOO
:	Liga modo silencioso (se for o primeiro caractere)
$VAR (gravada automaticamente em cada laço do loop)	
a-z	Opção que está sendo processada agora
?	Opção inválida ou falta argumento (modo normal) Opção inválida (modo silencioso)
:	Falta argumento (modo silencioso)
Variáveis de ambiente	
$OPTERR	Liga/desliga as mensagens de erro (o padrão é OPTERR=1)
$OPTARG	Guarda o argumento da opção atual (se houver)
$OPTIND	Guarda o índice do resto da linha de comando (não-opções)

O `usuarios.sh`, que estudamos bastante, pode ser modificado para funcionar usando o `getopts`. A desvantagem é que somente as opções curtas podem ser utilizadas. O interessante aqui é ver o `diff` das duas versões, para ficar bem fácil enxergar as diferenças das duas técnicas:

usuarios.sh (diff da versão 7 para usar getopts)

```
--- usuarios-7.sh  2007-11-20 18:19:22.000000000 -0200
+++ usuarios-7-getopts.sh 2007-11-21 00:23:18.000000000 -0200
@@ -1,92 +1,88 @@
 #!/bin/bash
 # usuarios.sh
 #
 # Mostra os logins e nomes de usuários do sistema
 # Obs.: Lê dados do arquivo /etc/passwd
 #
 # Versão 1: Mostra usuários e nomes separados por TAB
 # Versão 2: Adicionado suporte à opção -h
 # Versão 3: Adicionado suporte à opção -V e opções inválidas
 # Versão 4: Arrumado bug quando não tem opções, basename no
 #           nome do programa, -V extraindo direto dos cabeçalhos,
 #           adicionadas opções --help e --version
 # Versão 5: Adicionadas opções -s e --sort
 # Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
```

```
 #           leitura de múltiplas opções (loop)
 # Versão 7: Melhorias no código para que fique mais legível,
 #           adicionadas opções -d e --delimiter
+# Versão 7g: Modificada para usar o getopts
 #
 # Aurélio, Novembro de 2007
 #

 ordenar=0           # A saída deverá ser ordenada?
 inverter=0          # A saída deverá ser invertida?
 maiusculas=0        # A saída deverá ser em maiúsculas?
 delim='\t'          # Caractere usado como delimitador de saída

 MENSAGEM_USO="
 Uso: $(basename "$0") [OPÇÕES]

 OPÇÕES:
-  -d, --delimiter C  Usa o caractere C como delimitador
-  -r, --reverse      Inverte a listagem
-  -s, --sort         Ordena a listagem alfabeticamente
-  -u, --uppercase    Mostra a listagem em MAIÚSCULAS
+  -d C     Usa o caractere C como delimitador
+  -r       Inverte a listagem
+  -s       Ordena a listagem alfabeticamente
+  -u       Mostra a listagem em MAIÚSCULAS

-  -h, --help         Mostra esta tela de ajuda e sai
-  -V, --version      Mostra a versão do programa e sai
+  -h       Mostra esta tela de ajuda e sai
+  -V       Mostra a versão do programa e sai
 "

 # Tratamento das opções de linha de comando
-while test -n "$1"
+while getopts ":hVd:rsu" opcao
 do
-  case "$1" in
+  case "$opcao" in

       # Opções que ligam/desligam chaves
-      -s | --sort     ) ordenar=1    ;;
```

```
-       -r | --reverse  ) inverter=1   ;;
-       -u | --uppercase) maiusculas=1 ;;
-
-       -d | --delimiter)
-           shift
-           delim="$1"
-
-           if test -z "$delim"
-           then
-               echo "Faltou o argumento para a -d"
-               exit 1
-           fi
+       s) ordenar=1    ;;
+       r) inverter=1   ;;
+       u) maiusculas=1 ;;
+
+       d)
+           delim="$OPTARG"
        ;;

-       -h | --help)
+       h)
            echo "$MENSAGEM_USO"
            exit 0
        ;;

-       -V | --version)
+       V)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d \#
            exit 0
        ;;

-       *)
-           echo Opção inválida: $1
+       \?)
+           echo Opção inválida: $OPTARG
+           exit 1
+       ;;
+
```

```
+      :)
+          echo Faltou argumento para: $OPTARG
           exit 1
       ;;
    esac
-
-   # Opção $1 já processada, a fila deve andar
-   shift
 done

 # Extrai a listagem
 lista=$(cut -d : -f 1,5 /etc/passwd)

 # Ordena, inverte ou converte para maiúsculas (se necessário)
 test "$ordenar"    = 1 && lista=$(echo "$lista" | sort)
 test "$inverter"   = 1 && lista=$(echo "$lista" | tac)
 test "$maiusculas" = 1 && lista=$(echo "$lista" | tr a-z A-Z)

 # Mostra o resultado para o usuário
 echo "$lista" | tr : "$delim"
```

A mudança mais marcante é a eliminação do código de verificação do argumento para a opção `-d`. Ele já está pronto para ser usado em `$OPTARG`. Caso o argumento não tenha sido informado, o `case` cairá direto na alternativa ":", que mostra uma mensagem de erro genérica. O uso do `shift` também foi abolido. Fora isso, são apenas mudanças bem pequenas.

Para finalizarmos este capítulo, um exemplo da execução desta versão modificada, com a limitação de não poder usar opções longas, porém com a vantagem de poder juntar várias opções curtas em um mesmo argumento `-sru`.

```
$ ./usuarios-7-getopts.sh -h

Uso: usuarios-7-getopts.sh [OPÇÕES]

OPÇÕES:
  -d C     Usa o caractere C como delimitador
  -r       Inverte a listagem
  -s       Ordena a listagem alfabeticamente
  -u       Mostra a listagem em MAIÚSCULAS

  -h       Mostra esta tela de ajuda e sai
  -V       Mostra a versão do programa e sai
```

```
$ ./usuarios-7-getopts.sh -V
usuarios-7-getopts.sh Versão 7g
$ ./usuarios-7-getopts.sh -sru -d ,
WWW,WORLD WIDE WEB SERVER
UUCP,UNIX TO UNIX COPY PROTOCOL
UNKNOWN,UNKNOWN USER
TOKEND,TOKEN DAEMON
SSHD,SSHD PRIVILEGE SEPARATION
ROOT,SYSTEM ADMINISTRATOR
QTSS,QUICKTIME STREAMING SERVER
POSTFIX,POSTFIX USER
MYSQL,MYSQL SERVER
MAILMAN,MAILMAN USER
LP,PRINTING SERVICES
JABBER,JABBER USER
EPPC,APPLE EVENTS USER
DAEMON,SYSTEM SERVICES
AMAVISD,AMAVISD USER
$
```