Curso Básico
de Programação em Bash

Blau Araujo

Índice

Playlist....7
Vídeos das aulas complementares....7
Estudos de casos e dicas....7
Como apoiar o nosso trabalho....8
Aula 1 – Conceitos Básicos....9
1.1 – O que é o 'shell'....9
1.2 – Terminais e consoles....9
1.3 – O 'prompt' de comandos....10
1.4 – A aparência do 'prompt'....11
Uma pequena convenção para os nossos estudos....12
1.5 – Shell interativo e não-interativo....12
1.6 – Tipos de shell....13
O modo POSIX do Bash....15
1.7 - Os comandos builtin do Bash....15
1.8 - Como saber que tipo de shell você está utilizando....17
Método 1: comando 'echo $0'....17
Método 2: comando 'echo $SHELL'....18
Aula 2 – Antes do Primeiro Script....20
2.1 – Entendendo a utilidade dos scripts....20
2.2 - Sobre a execução de scripts....21
2.3 - Cuidados e boas práticas....22
2.3.1 - Nunca execute linhas de comando e scripts sem saber o que eles fazem!....22
2.3.2 - Preste atenção no prompt de comandos!....23
2.3.3 - Seja organizado!....24
2.3.4 - Nomes importam!....24
2.3.5 - Nunca inclua sua pasta de testes no $PATH!....25
Aula 3 – Nosso Primeiro Script....26
3.1 - As etapas de criação de um script em bash....26
3.2 - Etapa 1: Criar o arquivo do nosso script....26

3.2.1 - Sobre o nome do arquivo....26
3.2.2 - Sobre o diretório....27
3.2.3 - Como criar um novo arquivo pelo terminal....29
3.3 - Etapa 2: Escrever o conteúdo do script no arquivo....30
3.3.1 - Definindo o interpretador de comandos....31
3.3.2 - E se eu não definir um interpretador?....31
3.4 - Etapa 3: Tornar o arquivo executável....32
3.4.1 - Como saber se o arquivo é executável?....32
3.4.2 - O comando builtin 'test'....33
3.5 – Nosso primeiro script!....34
3.6 - Um script para criar scripts....36
3.6.1 - Decidindo onde criar o novo script....37
3.6.2 - Criando o arquivo do nosso script....38
3.6.3 - Tornando nosso script executável....39
3.6.4 - Editando o nosso script....40
3.6.5 - Evitando problemas....41
3.6.6 - Agora temos dois scripts!....43
Aula 4 – Variáveis....44
4.1 - Conceito....44
4.2 - Nomeando variáveis....44
4.3 - Tipos de variáveis....45
4.4 - Variáveis vetoriais....45
4.5 - Variáveis inalteráveis (read-only)....47
4.6 - Destruindo variáveis....47
4.7 - Atribuindo saídas de comandos a variáveis....48
4.8 - Acessando os valores das variáveis....48
Aula 5 – Variáveis Especiais....50
5.1 - Caracteres com significado especial....50
5.2 - Obtendo o status de saída do último comando....51
5.3 - Passando argumentos para o scripts....54
5.4 - Contando o número de argumentos....56
5.5 - Um pequeno resumo....57

Aula 6 – Vetores....58
6.1 - Um nome, muitos valores....58
6.2 – Criando vetores indexados....58
6.3 – Acessando os valores da array....60
6.4 – Lendo todos os valores de uma array....61
6.5 – Listando os valores da array por faixas de índices....62
6.6 – Descobrindo o número de elementos de uma array....63
Aula 7 – Concatenação de Strings....64
7.1 - Expansão de parâmetros....64
7.2 - Inserindo strings em strings....66
7.3 – O operador de concatenação....67
Aula 8 – Operações Aritméticas....69
8.1 - As operações básicas....69
8.2 - Operadores aritméticos....69
8.3 - Operadores de atribuição....70
8.4 - Precedência....71
8.5 – O problema do 'declare -i'....71
8.6 - O comando interno 'let'....74
Efetuando múltiplas expressões....75
8.7 – O comando composto '((expressão))'....76
8.8 - A expansão aritmética '$((expressão))'....76
8.9 - E os números não inteiros?....77
Aula 9 – Expansões do Shell....79
9.1 - O que são expansões....79
9.2 - Expansão de caminhos....80
9.3 - Expansão de nomes de arquivos....81
9.4 - Expansão de chaves....82
9.5 - Quebra (split) de palavras....83
9.6 - Substituição de comandos....85
Novas e antigas sintaxes das substituições de comandos....86
Substituições de comandos podem ser aninhadas....86
Variáveis compartilhadas por sessões diferentes....86

9.7 - Remoção de aspas.....86
Aula 10 – Expansões de Parâmetros.....88
10.1 - Trocando o nome pela pessoa.....88
10.2 - Indireções.....89
10.3 - Substrings.....90
10.4 - Comprimento de strings e número de elementos de arrays.....93
10.5 - Testando variáveis.....93
10.6 – Maiúsculas e minúsculas.....95
10.7 – Aparando strings.....96
10.8 - Busca e substituição de padrões.....97
Aula 11 – O loop 'for'.....99
11.1 - Comandos compostos.....99
11.2 - Sintaxe.....99
11.3 - Percorrendo as palavras em uma string.....100
11.4 - Percorrendo elementos de uma array.....103
11.5 - Percorrendo nomes de arquivos.....104
11.6 - Percorrendo faixas numéricas e alfabéticas.....104
11.7 - Controlando a execução do loop 'for'.....105
Aula 12 – Loops 'while' e 'until'.....107
12.1 - Estruturas de repetição condicional.....107
12.2 - Sintaxe.....107
12.3 - O loop 'while'.....108
12.4 - O loop 'until'.....109
12.5 - Loops infinitos.....109
12.6 - Interrompendo loops infinitos.....110
Aula 13 – O menu 'select'.....112
13.1 - Um menu simplificado.....112
13.2 - Sintaxe.....112
13.3 - O prompt 'PS3'.....113
13.4 - Menus com 'while' e 'until'.....114
Aula 14 – Estruturas de decisão 'if' e 'case'.....116
14.1 - A estrutura 'if', 'elif', 'then', 'else'.....116

14.2 - Um engano muito comum....117
14.3 - Operadores de encadeamento condicional....118
14.4 – A estrutura 'case'....119
Aula 15 – Funções....122
15.1 - Conceito....122
15.2 - Sintaxe geral....122
15.3 - Nossa primeira função....123
15.4 - Passando “argumentos”....124
15.5 - Retornando valores....125
15.6 - Escopo de variáveis....127
15.7 - A variável 'FUNCNAME'....128
15.8 - Diferenciando funções de comandos com o mesmo nome....129

Playlist

Playlist completa no Youtube
Duração total: 10 horas (média de 40 minutos por aula)

Vídeos das aulas complementares

- Um script para gerar scripts
- Colocando seus scripts no PATH
- Checando o PID do script
- Um pouco mais sobre o comando 'test'

Duração total: 100 minutos

Estudos de casos e dicas

- Alternando o compositor do Xfce via script
- Testando dependências no Bash
- Como descobrir o nome da função em execução
- Usar 'test' em vez de 'if', 'elif' e 'else'?
- Dica: o comando 'command'
- Função 'mp': como eu faço buscas no manual pelo terminal

Com o tempo, mais vídeos serão acrescentados a esta lista.

Como apoiar o nosso trabalho

Seu apoio é muito importante para a criação e a manutenção dos cursos gratuitos do canal debxp, qualquer valor ajuda demais!

- Apoio regular mensal (Apoia.se)
- PicPay
- PayPal
- PagSeguro

Aula 1 – Conceitos Básicos

1.1 – O que é o 'shell'

Mesmo que você nunca tenha criado nenhum script, mesmo que não saiba nada de programação, se você usa o terminal ou o console de qualquer distribuição GNU/Linux, ou o terminal do seu Mac, ou até o console do Windows, você já está utilizando algum tipo de *shell*.

De forma geral, você pode entender o *shell* como uma camada que envolve o sistema operacional como uma "casca" (daí no nome *shell*). Essa camada é responsável por fazer uma interface entre o usuário e o núcleo do sistema, que é chamado de *kernel*. A principal função do *kernel* é fazer a parte física do seu computador (ou *hardware*) funcionar. Mas, para isso, ele também precisa receber instruções dos vários programas que você executará e, principalmente, instruções vindas de você, o usuário. É aí que entra o *shell*.

1.2 – Terminais e consoles

Nos tempos mais remotos da computação, a comunicação física entre o usuário e o *kernel* era feita através da entrada de comandos por fitas ou cartões perfurados, com ou sem a ajuda de um teclado, e do recebimento de respostas através de uma impressora.

Naquela época, o conjunto composto por um equipamento de exibição de mensagens e um dispositivo de entrada de comandos era chamado de **console**. Mais tarde, quando os grandes computadores já permitiam o compartilhamento de seus recursos entre vários consoles distribuídos em locais diferentes, cada uma dessas estações de trabalho passou a ser chamada de **terminal**.

Com os avanços tecnológicos, à medida que os equipamentos foram reduzindo de tamanho e os monitores de vídeo começaram a ser utilizados para exibir mensagens, os consoles e terminais já não eram mais equipamentos exatamente. Assim como hoje nós utilizamos sistemas operacionais inteiros virtualizados, os consoles e terminais também foram virtualizados através de programas, dando origem à

terminologia **console virtual** (VC) e **terminal virtual** (VT).

Hoje portanto, quando dizemos "console" ou "terminal", estamos nos referindo justamente à virtualização em *software* (feita pelo *kernel*) dos antigos terminais e consoles eletrônicos dos primórdios da computação, e o programa responsável por tornar essa virtualização capaz de receber comandos de um teclado e exibir mensagens em um monitor, é precisamente o *shell*!

1.3 – O 'prompt' de comandos

Assim que você faz o login no console ou abre um terminal no seu ambiente gráfico, o *shell* é iniciado e nos mostra um ou mais caracteres e um cursor no local onde serão exibidos os comandos digitados por nós. Este ponto onde aparece o cursor é o **prompt de comandos**. Sua função é indicar que o *shell* está "pronto" (daí *prompt*) para receber comandos. Na verdade, podemos dizer que **acessar o prompt de comandos** é o mesmo que **acessar o shell**.

Nós temos basicamente três formas de acessar o *prompt* de comandos em sistemas GNU/Linux:

- Nos ambientes gráficos, abrindo um emulador de terminal.
- A partir de um ambiente gráfico, podemos abrir um console virtual teclando o atalho `Ctrl+F[n]`, onde `F[n]` é uma tecla de função entre `F1` e `F6`, teclando `Ctrl+F7` (ou `Ctrl+F8` em algumas distribuições) para retornar ao ambiente gráfico.
- Simplesmente não instalando/iniciando um ambiente gráfico, caso em que temos apenas o console virtual para trabalhar.

Ainda em termos gerais, as principais diferenças entre trabalhar com um emulador de terminal ou trabalhar com um console são:

Emulador de Terminal (ambiente gráfico)	**Console Virtual (modo "texto")**
Você já está "logado" no ambiente gráfico e, portanto, não será pedida nenhuma informação de *login* e senha.	Sempre que for iniciado, você terá que informar o seu *login* e senha para ter acesso ao *prompt*.
Você pode usar o mouse para	Embora existam programas para

selecionar textos, rolar o histórico da sessão, copiar, colar, etc...	implementar o uso do mouse, o normal é que todas as ações sejam realizada por atalhos de teclado.
A quantidade de caracteres por linhas e colunas de texto dependem da largura e da altura da janela do terminal e do tamanho da fonte que você escolheu.	A quantidade de linhas e colunas de texto dependem apenas da resolução que o *kernel* identificou para o seu monitor.
Você pode escolher diversos tipos de fontes e exibir corretamente caracteres gráficos dos mais diversos tipos.	Você está limitado ao conjunto de fontes específicas para o console e ao conjunto de caracteres que elas são capazes de exibir corretamente.

Fora isso, as únicas diferenças seriam aquelas relativas ao contexto dos seus comandos – não faria muito sentido executar comandos para abrir programas gráficos ou executar tarefas com eles estando no console, por exemplo.

1.4 – A aparência do 'prompt'

Dependendo do sistema operacional e das customizações feitas, o *prompt* pode apresentar várias informações, mas o que importa para nós por enquanto são os símbolos mostrados imediatamente antes do cursor. Quando vemos `$` ou `%` antes do cursor, o *shell* está indicando que você fez login como um **usuário normal**. Já o sinal `#` informa que você fez login como usuário administrativo, o usuário **root**.

O usuário **root** é uma conta especial que possui permissões para realizar qualquer tipo de operação no sistema, **inclusive destruí-lo completamente!** Portanto, a cerquilha (`#`) é um aviso de que estamos numa zona altamente perigosa para quem não sabe exatamente o que está fazendo e como fazer as coisas.

> ***Importante!*** *Fique atento ao seu prompt e, principalmente, a menos que seja orientado para isso,* ***jamais execute os nossos exemplos e experimentos como usuário root****!*

Uma pequena convenção para os nossos estudos

Além de informar se estamos "logados" como usuários normais ou administrativos, o shell também é capaz de informar o nome do diretório em que estamos trabalhando. Como veremos bem mais adiante, em *shells* como o Bash, por exemplo, o símbolo `~` (til) representa o nome da sua pasta pessoal de usuário.

Por exemplo, se o seu nome de usuário for `blau`, o til representará o diretório `/home/blau`. Desta forma, o shell tem um jeito de informar em que pasta você está trabalhando sem ocupar muito espaço no prompt.

Outras informações que costumam vir configuradas para serem exibidas em sistemas GNU/Linux são o nome do usuário e o nome da máquina na rede (*hostname*). Então, é muito comum encontrar algo assim no prompt após o *login* ou quando abrimos um emulador de terminal:

```
blau@enterprise:~$
```

Onde `blau` é o meu nome de usuário, `@` é um símbolo que, em inglês é lido como ***at*** ("em", em português), `enterprise` foi o nome que eu dei para a minha máquina na rede durante a instalação, `:` é só um separador, `~` é a minha pasta *home* e, finalmente, `$` me diz que estou "logado" como um usuário comum.

Aqui nos nossos exemplos e execícios, porém, nós não precisamos de todas essas informações. Por isso, vamos adotar a seguinte convenção:

```
:~$ --> Pasta corrente mais indicação de usuário normal;
```

Ou...

```
:~# --> Pasta corrente mais indicação de usuário 'root';
```

1.5 – Shell interativo e não-interativo

Quando abrimos um terminal e começamos a digitar comandos, nós estamos utilizando o *shell* de forma **interativa**. No modo interativo, nós entramos com um comando, o *shell* processa esse comando, nos dá uma resposta, nós vemos a

resposta, pensamos e decidimos o que fazer em seguida. Ou seja, nós interagimos diretamente com o shell através dos nossos comandos e observando os resultados obtidos.

Mas existe uma outra forma de trabalhar com o *shell* que pode ser muito útil e prática, principalmente quando temos que executar vários comandos em sequência e avaliar os possíveis retornos para decidir quais comandos dar em seguida. Trata-se do modo **não-interativo**, que nada mais é do que escrever todos os comandos em um arquivo de texto, tornar esse arquivo executável e simplesmente mandar o *shell* executá-lo de uma só vez. A este arquivo, que contém todos os comandos e instruções que queremos que o *shell* execute, nós damos o nome de **script**.

Como veremos, além dos comandos e programas para executar tarefas específicas, o *shell* também oferece diversos tipos de recursos capazes de transformar uma simples lista de comandos a serem executados em lote em verdadeiros programas!

> ***Observação:*** *como diz Aurélio Jargas, autor de vários materiais incríveis sobre programação em shell e expressões regulares,* ***"um programa é apenas um script feito do jeito certo"****.*

1.6 – Tipos de shell

Existem vários tipos de *shell*, figurando entre eles:

Nome	Executável	Descrição
Bourne Shell	`sh`	Desenvolvido por Stephen Bourne, da AT&T, é o Shell padrão do UNIX 7 em substituição do Thompson Shell, cujo executável possuía o mesmo nome, **sh**.
Bourne-Again Shell	`bash`	GNU/Bash ou, como é mais conhecido, Bash, é um shell de comandos Unix e uma linguagem interpretada escrita inicialmente por Brian Fox para o Projeto GNU em substituição ao Bourne Shell. Quando Fox foi afastado da FSF, em 1994, o desenvolvimento do Bash passou para Chet Ramey.
Almquist Shell	`ash`, `sh`	É um shell Unix escrito originalmente por Kenneth Almquist no fim dos anos '80 como um clone da

		variente System V.4 do Bourne Shell e substituiu o Bourne Shell original nas versões do Unix BSD lançadas no começo dos anos '90, razão pela qual algumas de suas implementações ainda utilizam o nome de executável `sh`.
Debian Almquist Shell	`dash`, `sh`	Em 1997, o Almquist Shell foi portado do NetBSD para o Debian, que em 2002 lançou uma versão renomeada para Debian Almquist Shell, ou `dash`, priorizando a compatibilidade com os padrões POSIX e uma implementação bem mais enxuta em relação à original.
Korn Shell	`ksh`	Desenvolvido sobre o código do Bourne Shell por David Korn no começo dos anos '80, o KornShell era inicialmente um projeto proprietário e mais tarde adotou uma licença compatível com as diretrizes Open Source.
Z Shell	`zsh`	Criado com a proposta de ampliar as funcionalidades do Bourne Shell, o Zsh traz diversos recursos presentes no Bash e no KornShell. Em 2019, foi adotado como shell padrão do macOS Catalina, papel que era ocupado até então pelo Bash.

E a lista poderia seguir por várias páginas! Mas, para nós o que realmente interessa é o Bourne-Again Shell, que é o shell padrão do Projeto GNU e, portanto, o shell mais presente em sistemas GNU/Linux.

Outra coisa importante de destacar, é que cada *shell* tem a sua própria forma de reconhecer e interpretar os comandos dos usuários e trabalha com sintaxes que podem ser muito diferentes. Então, se estiver escrevendo um script em Bash que precisará ser compatível com plataformas que utilizem outros *shells*, provavelmente será necessário observar as definições da **norma POSIX** (*Portable Operating System Interface*, “Interface Portável Entre Sistemas Operacionais”, em português). Como o nome diz, o objetivo é garantir a portabilidade do código de um programa ou de um script a partir de um conjunto de normas.

Ao longo deste treinamento, nós não teremos esse tipo de preocupação. Nosso objetivo aqui é conhecer e explorar o máximo possível as funcionalidades e os

recursos do Bash. Contudo, o modo POSIX é uma dessas funcionalidades, então vamos tirar esse elefante da sala de uma vez.

O modo POSIX do Bash

Para começar, todo *shell* também é um programa executável, e o executável do Bash chama-se `bash`. Ele é invocado automaticamente após o *login*, mas também pode ser executado no terminal com o objetivo de iniciar outra sessão do *shell*. Este comportamento é equivalente ao que acontece quando executamos um script, ou seja, exceto em situações bem específicas que não vêm ao caso agora, cada script irá iniciar uma nova sessão do *shell* quando for executado.

Com isso em mente, fica mais fácil entender que o Bash possui algumas opções de execução, entre elas, a opção de ser executado no modo de compatibilidade com as normas POSIX, que é o **modo POSIX**. Quando iniciado com a opção `--posix`, ou executando o comando `set -o posix` em uma sessão já iniciada, o Bash terá seu comportamento padrão alterado para atender as normas POSIX.

Entrando em modo POSIX, o Bash altera uma lista de 59 aspectos do seu comportamento normal, o que não caberia detalhar neste tópico introdutório. Mas, se estiver curioso, a lista completa de mudanças pode ser lida diretamente no manual do Bash.

1.7 - Os comandos builtin do Bash

O bash possui um farto conjunto comandos internos chamados de ***builtin***, e nós utilizamos alguns deles bem frequentemente no terminal, como é o caso do comando `cd`, usado para mudar de diretório.

Para ver a lista completa dos *builtins*, nós podemos executar o seguinte comando *builtin*:

```
:~$ help
```

Também podemos ver as informações sobre um comando interno do Bash com a sintaxe:

```
help nome_do_comando
```

Isso fará com que um manual resumido do comando seja exibido no terminal, ou uma mensagem de erro, caso o comando não seja um *builtin*.

> ***Nota:*** *nem tudo que você executa no terminal é um comando interno do bash!*

Então, se estiver na dúvida se um comando é ou não é *builtin*, basta executar `help nome_do_comando`. Se a resposta for um erro, não é um *builtin*.

Outra forma de descobrir, é com o comando interno `type`:

```
type nome_do_comando
```

Se for o caso, ele retornará a mensagem:

```
nome_do_comando é um comando interno do shell
```

Para entender melhor, é uma boa ideia você abrir agora mesmo um terminal e executar os dois comandos abaixo:

```
:~$ help type
:~$ type help
```

Aliás, aproveitando que está com o terminal aberto, experimente esses comandos:

```
:~$ help cd
:~$ help ls
:~$ help command (observe o que diz na opção '-v')
:~$ type cd
:~$ type ls
:~$ type command
```

Observe não só o que o `help` diz sobre esses comandos, mas principalmente as mensagens que o Bash retorna, se são *builtins* ou não, e anote as suas descobertas.

Aproveite para ver o que acontece com os comandos abaixo:

```
:~$ command -v type
:~$ command -v help
:~$ command -v cd
:~$ command -v ls
```

1.8 - Como saber que tipo de *shell* você está utilizando

Para encerrar este tópico, nós vamos ver duas formas de descobrir qual *shell* está sendo utilizado no seu sistema. As duas formas podem apresentar resultados equivocados, mas são um bom ponto de partida.

Método 1: comando 'echo $0'

No Bash, `$0` é o que nós chamamos de **parâmetro posicional** (não se assuste com o nome por enquanto, nós falaremos sobre isso em detalhes em outros tópicos), que é uma variável especial que armazena o nome do programa em execução. Se você está no prompt de comando, o programa em execução o *shell* é o próprio *shell*!

Mas, o que faz esse tal de `echo`?

Não me pergunte, veja você mesmo!

```
:~$ help echo
```

Se você foi pesquisar, deve ter encontrado isso:

```
:~$ help echo
echo: echo [-neE] [ARG ...]
    Write arguments to the standard output.

    Display the ARGs, separated by a single space character
    and followed by a newline, on the standard output.
```

Que poderia ser traduzido como algo assim:

```
:~$ help echo
echo: echo [-neE] [ARGUMENTOS ...]
    Escreve argumentos na saída padrão.
```

```
    Exibe os ARGUMENTOS separados por um único espaço
    e seguidos de uma nova linha na saída padrão.
```

No nosso comando (`echo $0`), o argumento é o conteúdo armazenado em `$0`, e a tal da **saída padrão**, de forma bem simplificada, nada mais é do que a tela do seu terminal. Então, podemos dizer que o comando `echo` imprime na tela aquilo que nós passarmos para ele como argumento. Por exemplo, experimente isso:

```
:~$ echo Olá, mundo!
```

Aqui, a frase `Olá, mundo!` é o argumento que nós queremos que o comando `echo` exiba. Se você testou, deve ter visto algo como isso:

```
:~$ echo Olá, mundo!
Olá, mundo!
```

Agora que você já sabe como funciona o comando `echo` e que a variável `$0` pode conter o nome do shell que você está usando, vamos ver o que acontece. Aqui no meu terminal, a saída foi essa:

```
:~$ echo $0
bash
```

Mas, perceba uma coisa: a variável `$0` nem sempre irá conter o nome do *shell*! Esta foi uma situação especial em que você estava trabalhando diretamente no *prompt*. Lembre-se de que eu disse: *`$0` é uma variável especial que armazena o nome do programa em execução*. Se você chamar esse comando dentro de um script, por exemplo, ela vai conter o nome do script, pois é ele, e não o *prompt do shell* que está sendo executado.

Ou seja, este método só funciona no modo interativo e não serve para verificar qual é o shell que está sendo usado dentro de um script!

Método 2: comando 'echo $SHELL'

Aqui, nós vamos dar uma olhada no conteúdo de outra variável do Bash, a variável

`$SHELL`. O interessante dessa variável é que ela está disponível para ser consultada por qualquer comando ou programa, por isso é chamada de **variável de ambiente**. Sua função é armazenar o caminho do executável do *shell* configurado para um determinado usuário assim que ele fizer um *login*. Executando o comando, foi isso que eu obtive na saída:

```
:~$ echo $SHELL
/bin/bash
```

Nota: *Estas não são as únicas formas de encontrar o nome do shell em uso, mas são as mais simples para o propósito dessa introdução.*

Aula 2 – Antes do Primeiro Script

2.1 – Entendendo a utilidade dos scripts

Vamos relembrar alguns conceitos da primeira aula:

- O shell é um interpretador que executa comandos que podem vir da entrada padrão (terminal), ou de um arquivo (script).
- Um script é um arquivo de texto contendo os mesmos comandos e instruções que você digitaria no terminal.
- Quando trabalhamos com o terminal, estamos usando o shell de forma "interativa".
- Quando usamos scripts, estamos usando o shell de forma "não interativa".

Sendo assim, a utilidade mais básica de um script, é a possibilidade de executar comandos "em lote", em vez de digitar cada um deles no terminal.

Considere a situação hipotética de um usuário que precise, por algum motivo, executar os quatro comandos abaixo em sequência e com uma certa frequência...

```
:~$ whoami
:~$ hostname
:~$ uptime -p
:~$ uname -rms
```

Esses comandos (que não são comandos internos do Bash) retornariam as seguintes informações:

Comando	**Descrição**
`whoami`	retorna o nome do usuário
`hostname`	retorna o nome do host do sistema (nome da máquina)
`uptime`	retorna há quanto tempo a máquina está ligada
`uname`	retorna informações sobre o kernel do sistema

E a saída no terminal poderia ser algo assim...

```
:~$ whoami
arthur
:~$ hostname
excalibur
:~$ uptime -p
up 2 weeks, 3 days, 19 hours, 42 minutes
:~$ uname -rms
Linux 4.19.0-6-amd64 x86_64
```

Sabendo que o shell permite a execução de comandos a partir de um arquivo, faz muito mais sentido criar um script contendo todos esses comandos e executá-los todos chamando apenas esse script. O conteúdo do arquivo teria a seguinte aparência...

```
#!/usr/bin/env bash

whoami
hostname
uptime -p
uname -rms
```

Observe que, a não ser pela primeira linha do arquivo (da qual falaremos no próximo tópico), que é onde nós indicamos para o shell qual será o interpretador de comandos utilizado, o restante do arquivo contém apenas a mesma lista de comandos que o nosso usuário teria que digitar e executar manualmente no terminal! Ou seja, o nosso usuário hipotético (o `arthur`) poderia salvar esse arquivo com o nome `infos`, ou outro nome qualquer, e executá-lo de uma única vez no terminal.

```
:~$ ./infos
arthur
excalibur
up 2 weeks, 3 days, 19 hours, 42 minutes
Linux 4.19.0-6-amd64 x86_64
```

2.2 - Sobre a execução de scripts

Observe que o usuário do exemplo acima chamou seu script utilizando `./` antes do

nome do arquivo.

```
:~$ ./infos
```

Isso significa que o arquivo executável do script está na pasta em que ele está trabalhando – no caso, `/home/arthur`. Se ele estivesse trabalhando em qualquer outro local – na pasta `~/docs/clientes`, por exemplo – a forma de executar o script seria a seguinte:

```
:~/docs/clientes$ ~/infos
```

Ou assim:

```
:~/docs/clientes$ /home/arthur/infos
```

Mas, também existe a possibilidade de salvar o arquivo do script em uma pasta específica e chamá-lo a partir de qualquer outra em que você esteja trabalhando sem indicar o caminho completo até ele. Nós fazemos isso configurando uma variável de ambiente chamada `PATH`.

Nós entraremos nos detalhes sobre o $PATH e como configurá-lo em outro tópico. Por enquanto, basta saber que, se o arquivo do nosso usuário hipotético estivesse numa pasta definida na variável de ambiente `PATH`, onde quer que ele estivesse, bastaria executar o script da seguinte forma:

```
:~$ infos
```

O que é bem mais interessante, mas exige alguns cuidados.

2.3 - Cuidados e boas práticas

Já que mencionamos “cuidados”, vamos aproveitar para falar de alguns deles.

2.3.1 - Nunca execute linhas de comando e scripts sem saber o que eles fazem!

Quando estamos aprendendo, é muito comum sairmos digitando (ou até colando) indiscriminadamente qualquer linha de comando ou script que encontramos na

internet.

Nunca faça isso, e aqui estão alguns motivos.

- Deveria ser óbvio, mas comandos nem sempre são inofensivos. Se você não sabe o que um comando faz, a possibilidade de você acabar executando algo capaz de destruir o seu sistema é real!
- Outra realidade são os scripts maliciosos, feitos com o propósito único de prejudicar os outros.
- Mesmo que não sejam maliciosos, os scripts e comandos que você executa sem saber para que servem e como funcionam podem não fazer o que você acha que eles fazem.
- O desenvolvedor do script, ou a pessoa que compartilha um comando imaginando que está ajudando de alguma forma, também pode ser inexperiente, e a solução pode acaber sendo incorreta e até prejudicial.
- Mesmo que não seja incorreta e o desenvolvedor, ou a pessoa que compartilhou um comando, seja experiente, a solução pode ser simplesmente incompatível com o seu sistema, causando mais problemas do que você já tem.

Portanto, antes de executar qualquer comando ou script no seu sistema, você deve sempre analisar e entender o que ele faz!

> ***Importante!*** *Se você* ***ainda*** *não é capaz de analisar comandos ou scripts, busque a orientação de alguém mais experiente que seja da sua confiança, ou simplesmente não execute!*

2.3.2 - Preste atenção no prompt de comandos!

Como dissemos no primeiro tópico, o símbolo no final do prompt indica se estamos trabalhando como usuário normal (`$`) ou como usuário administrativo (`#`). Como usuário administrativo (ou `root`), você tem privilégios para executar qualquer coisa, e basta um comando errado para danificar de forma irrecuperável o seu sistema!

Observe, porém, que o seu prompt pode estar indicando que você está trabalhando como usuário normal (`$`) mas, ainda assim, você pode estar executando um comando como usuário administrativo! É o que acontece quando precedemos um comando com o comando `sudo`.

> ***Nota:*** *O `sudo` (substitute user and do – "troque de usuário e faça") é um comando (utilitário) que nos dá privilégios temporários de **super usuário**, mas não altera o prompt, o que pode causar alguma confusão. Apesar de pedir uma senha antes de executar o comando que vem depois dele, o que até poderia servir de alerta, o sudo possui um tempo de validade. Ou seja, numa mesma sessão, apenas o primeiro comando executado com `sudo` pedirá a senha do usuário.*

2.3.3 - Seja organizado!

Quando estamos estudando, testando um código, ou quando estamos com pressa, não é raro sairmos criando scripts em qualquer pasta. Esta, definitivamente, não é uma boa prática!

O melhor é sempre ter uma pasta reservada para seus testes e experimentos. Você pode, por exemplo, criar uma pasta chamada `testes` na sua pasta pessoal...

```
:~$ mkdir ~/testes
```

Dentro dela, você pode criar novas pastas para cada assunto que estiver estudando. Assim você não bagunça o seu sistema, não corre o risco de executar algo que não deve no lugar errado, e ainda mantém um controle das coisas que andou experimentando.

2.3.4 - Nomes importam!

Ao criar os arquivos dos seus scripts, verifique se ele não tem o mesmo nome de algum comando ou utilitário existente sistema. Na hora de executar qualquer coisa, o shell dará prioridade para os comandos que aparecerem primeiro na lista de caminhos indicados na variável de ambiente `PATH`. Então, é provável que o seu script nem venha a ser executado se ele possuir um nome que conflite com o de algum aplicativo, a menos que você tenha incluído as suas pastas de execução locais antes das pastas globais no `PATH`, o que é muito errado!

Outro problema da falta de atenção com os nomes dos arquivos, aconteceria no caso de você tentar disponibilizar um script para todos os usuários do sistema. Neste caso, o script deveria estar na pasta `/usr/local/bin` ou `/usr/bin`, que é onde ficam todos os executáveis a que todos os usuários têm acesso. Um erro aqui seria ainda mais

grave, já que só é possível copiar arquivos para essas pastas como usuário administrativo, e **isso permitiria sobrescrever sem qualquer aviso um aplicativo ou utilitário do sistema com o mesmo nome do seu script!**

2.3.5 - Nunca inclua sua pasta de testes no $PATH!

Ainda sobre os caminhos de execução, nunca inclua no `PATH` a sua pasta de testes e experimentos!

Os motivos são até meio óbvios a essa altura, mas nunca é demais alertar que os scripts na pasta de testes **não são projetos finalizados**, eles podem conter erros e causar problemas. Quando você tiver certeza de que eles podem ser usados em produção, basta movê-los para um local definitivo que esteja no `PATH`.

> ***Nota:*** *Essas boas práticas e cuidados referem-se a detalhes a que devemos estar sempre atentos, principalmente no processo de aprendizado. Existem outros cuidados e regras de boas práticas, especialmente os relacionados ao processo de produção de código, que serão vistos ao longo dos próximos tópicos.*

Aula 3 – Nosso Primeiro Script

3.1 - As etapas de criação de um script em bash

A criação de um script sempre envolverá pelo menos três etapas:

- Criar o arquivo do script
- Editar o arquivo do script
- Tornar o arquivo executável

Neste tópico, nós abordaremos cada uma dessas etapas e, ao final, criaremos juntos um script que nos ajudará a criar scripts.

3.2 - Etapa 1: Criar o arquivo do nosso script

Os scripts são basicamente arquivos de texto puro sem formatação (*raw text*), e você pode criá-los da forma que preferir. Obviamente, como todas as etapas seguintes exigirão a digitação de comandos no terminal, faz mais sentido fazer todos os procedimentos por lá, e esta será a abordagem que nós adotaremos.

Para criar um novo arquivo no terminal, nós precisamos do nome do arquivo que queremos criar e do local (diretório) onde ele será criado.

3.2.1 - Sobre o nome do arquivo

Como vimos no tópico anterior, escolher bem o nome do arquivo é muito importante. Além disso, nós temos que seguir algumas regrinhas básicas:

- O nome não pode conter espaços, acentos e outros símbolos tipográficos além dos caracteres alfabéticos, números, pontos (`.`), o sinal de menos (`-`) e o sublinhado (`_`).
- Quanto às extensões, principalmente a extensão `.sh`, muito utilizada, elas são irrelevantes para o shell e só servem para nos informar de que se trata do arquivo de um script ou para diferenciar scripts de comandos no momento da execução.

> ***Nota:*** *No Linux, e em outros sistemas unix like, o ponto (.) no nome de um arquivo é um caractere como outro qualquer e não possui nenhum significado especial.*

3.2.2 - Sobre o diretório

Em produção, é bem provável que você queira que o seu script seja executado de qualquer diretório. Neste caso, é interessante que ele seja criado (ou movido após finalizado) numa pasta que esteja no `PATH` do sistema ou em um caminho da sua pasta pessoal que você tenha incluído no `PATH`.

Se você não sabe quais são essas pastas, basta executar no terminal...

```
:~$ echo $PATH
```

Aqui, o comando `echo` irá exibir todas as pastas que estão listadas na variável de ambiente `PATH`, cada uma separada por dois pontos (`:`) das outras. Por exemplo, aqui no meu sistema:

```
:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/home/blau/bin
```

Esta lista mostra que os aplicativos e scripts que eu executar no prompt de comando serão procurados nestes caminhos na seguinte ordem:

```
/usr/local/bin
/usr/bin
/bin
/usr/local/games
/usr/games
/home/blau/bin
```

Repare que a última pasta em que o shell irá procurar os executáveis será em `/home/blau/bin`, que é um diretório (`bin`) na minha pasta de usuário (`/home/blau`). Este caminho não existia originalmente na variável de ambiente `PATH` do meu sistema, eu tive que incluí-lo com o seguinte comando:

```
:~$ PATH=$PATH:/home/blau/bin
```

Este tipo de procedimento é chamado de **atribuição**. Para isso, nós utilizamos o **operador de atribuição** `=`. À esquerda, nós indicamos o nome da variável a que queremos atribuir um valor, que é o que vem à direita do operador:

```
# Atribuição de um valor à uma variável...
NOME_DA_VARIÁVEL=VALOR
```

No caso, a variável `PATH` receberá o valor `$PATH:/home/blau/bin`, onde:

- `$` é como nós dizemos ao shell que precisamos acessar o valor armazenado em uma variável (no caso, `PATH`);
- `:` é o separador de cada caminho da lista;
- e `/home/blau/bin` é o caminho que queremos incluir ao final da lista.

Todo esse valor é o que chamamos de *string*, que nada mais é do que uma sequência de caracteres. Portanto, neste procedimento de atribuição, nós estamos **concatenando** o valor que já está em `PATH` com o caractere do separador (`:`) e o caminho que queremos incluir na lista.

> ***Nota:*** *Nós teremos tópicos sobre atribuições de valores a variáveis e concatenação de string mais para frente.*

Mas, se o seu arquivo não estiver em uma das pastas listadas, ele ainda poderá ser executado indicando a sua localização:

```
:~$ /caminho/da/pasta/do/seu_scritp
```

Se você estiver na mesma pasta do arquivo do script, basta executá-lo assim:

```
:~$ ./seu_scritp
```

Onde o `./` informa para o shell que o arquivo `seu_script` está na pasta corrente (a pasta em que você está trabalhando no momento).

Para efeito de treinamento e estudo, nós não vamos criar nem executar os nossos scripts nas pastas listadas no `PATH`. Afinal, nós já vimos esta não seria uma boa prática.

3.2.3 - Como criar um novo arquivo pelo terminal

Para criar um arquivo novo pelo terminal, sem abri-lo através de um editor de textos nem criando um arquivo vazio pelo gerenciador de arquivos, nós utilizamos uma coisa chamada **redirecionamento de saída para arquivos**, cuja sintaxe envolve o uso de um ou dois símbolos de maior (`>` ou `>>`). A diferença entre usarmos um ou dois símbolos de maior é:

- `>` - Cria um novo arquivo com determinado conteúdo mesmo que ele já exista. Se o arquivo existir, ele será recriado com o novo conteúdo.
- `>>` - Inclui um determinado conteúdo ao final de um arquivo existente. Se o arquivo não existir, ele será criado.

O conceito básico desse processo de criação de arquivos está no fato de que o shell é capaz de manipular entradas e saídas de comandos, permitindo que elas sejam direcionadas para outros comandos e arquivos.

Aqui, nós utilizaremos o redirecionamento da saída de um comando para um arquivo (ainda inexistente), o que obedece a sintaxe:

```
Novo arquivo: comando > arquivo
```

Ou...

```
Append: comando >> arquivo
```

Porém, se não existir um comando sendo executado à esquerda, nenhuma saída será gerada, o que resultará em um arquivo vazio, e é exatamente dessa característica que iremos nos valer para criar novos arquivos para os nossos scripts:

```
:~$ >> NOVO_ARQUIVO_DE_SCRIPT
```

Nota: *As duas abordagens (`>` ou `>>`) funcionam. Contudo, só devemos utilizar um sinal de maior quando tivermos certeza de que não existe outro arquivo com o mesmo nome, ou ele seria sobrescrito. Com dois sinais de maior (`>>`), mesmo que exista um arquivo com o mesmo nome, ele só tentará acrescentar uma string vazia ao final desse arquivo, o que não produzirá nenhuma alteração.*

Também é interessante observar que existe um outro comando (um utilitário, na verdade) capaz de criar arquivos pelo terminal: o comando `touch`. Apesar de o resultado ser o mesmo, quer dizer, você acaba criando o arquivo se ele não existir, esta não é a finalidade original do `touch`, que é um comando para alterar o marcador da data e da hora do último acesso a um arquivo. Além disso, o `touch` é um utilitário do sistema, não um comando interno do shell, o que pode implicar em códigos com menor desempenho.

> ***Nota:*** *Na linha de comando, não é possível criar vários arquivos* ***apenas*** *com um único redirecionamento de saída, mas é possível fazer isso com o `touch`. Existem formas de criar múltiplos arquivos com os redirecionamentos, mas todas elas envolvem recursos do Bash que nós veremos em outros tópicos.*

Por fim, você ainda pode criar novos arquivos pelo terminal utilizando um editor de textos, como o `nano` ou o `vim`, por exemplo.

3.3 - Etapa 2: Escrever o conteúdo do script no arquivo

Você pode utilizar o editor de textos da sua preferência, mas nós estamos executando comandos no terminal e, portanto, é muito mais prático e interessante permanecer nele.

Existem várias opções de editores para a linha de comando, mas os dois mais populares são o **GNU Nano** e o **Vim**. Para uso imediato, e não tirar o foco do aprendizado do Bash, eu recomendo utilizar o `nano`. Ele pode não ser tão rico em recursos e nem tão poderoso quanto o Vim, mas oferece tudo de que precisamos nesta fase de descobertas.

> ***Nota:*** *Vale muito a pena aprender a utilizar o Vim, eu só não recomendo misturar as estações neste momento, quando estamos diante de tantas informações novas para assimilar.*

Para editar o arquivo recém-criado (ou criar um arquivo que ainda não existe), basta executar:

```
:~$ nano nome_do_novo_arquivo
```

Depois disso, é só começar a escrever o código, salvar com `Ctrl+S` e sair com `Ctrl+X`.

3.3.1 - Definindo o interpretador de comandos

A primeira linha do nosso script, antes de todos os comandos e instruções, deve conter a definição do interpretador de comandos. Esta linha é chamada de várias formas, sendo as mais comuns: ***hashbang*** e ***shebang***, como uma referência aos dois caracteres iniciais da linha (`#!`).

No caso de querermos utilizar o Bash, nós podemos escrever a linha do interpretador de duas formas:

```
#!/bin/bash
```

Ou...

```
#!/usr/bin/env bash
```

A diferença entre elas, é que a primeira (`#!/bin/bash`) pressupõe a existência do executável do Bash na pasta `/bin`, enquanto a segunda (`#!/usr/bin/env bash`) consultará os recursos do ambiente para descobrir onde está o executável.

As duas formas estão corretas e têm grandes chances de funcionar na maioria das distribuições Linux, mas a segunda forma é mais portável, ou seja, pode ser compatível com mais plataformas.

3.3.2 - E se eu não definir um interpretador?

Neste caso, o script será interpretado pelo shell padrão do sistema, seja ele qual for, e os seus comandos e instruções podem não ser reconhecidos, o que resultaria em erros de sintaxe.

> ***Importante!*** *Lembre-se de que o shell pode ser utilizado de forma interativa (linha de comando) ou não-interativa (script) e, geralmente, os sistemas GNU/Linux utilizam algum shell compatível com as normas POSIX no modo não-interativo, como é o caso*

> *do Debian, que utiliza o `dash` como shell padrão para o modo não-interativo.*

Como regra geral, nós sempre definimos o interpretador, mas o script ainda poderá funcionar corretamente caso ele contenha apenas uma lista de programas utilitários a serem executados em sequência.

3.4 - Etapa 3: Tornar o arquivo executável

Todo arquivo possui atributos de permissão, como leitura, escrita e execução. Além disso, essas permissões podem ser dadas a grupos de usuários diferentes, inclusive para todos os usuários.

Para alterar as permissões de um arquivo, existe o utilitário `chmod`, que é usado segundo a sintaxe geral:

```
chmod opções arquivo
```

Para tornar o nosso arquivo executável, basta utilizá-lo desta forma:

```
:~$ chmod +x arquivo
```

Onde a opção `+x` ativa o indicador (*flag*) de executável do arquivo.

3.4.1 - Como saber se o arquivo é executável?

A forma mais simples e eficiente de descobrir se um arquivo é executável ou não é com o comando:

```
:~$ ls -l nome_do_arquivo
```

Se ele for executável, você verá diversos `x` na primeira coluna das informações exibidas sobre o arquivo. Por exemplo:

```
:~$ ls -l teste.sh
-rwxr-xr-x 1 blau blau 6084 out 1 17:15 teste.sh
   ^  ^  ^
```

```
   |  |  |
   +--+--+--- indicadores de arquivo executável
```

3.4.2 - O comando builtin 'test'

Mas existe uma forma bem mais interessante de fazer isso utilizando os comandos builtin do Bash:

```
test -x nome_do_arquivo; echo $?
```

Executando isso na linha de comando ou em um script, você verá um `0`, se for executável, ou um número diferente de zero se não for executável. Por exemplo:

```
:~$ test -x teste.sh; echo $?
0

:~$ test -x arquivo.txt; echo $?
1
```

Aqui, `$?` é uma variável especial do shell que armazena o estado de saída do último comando executado. Se o programa anterior foi executado com sucesso, ela armazenará o valor `0`. Caso contrário, ela armazenará um valor diferente de `0` (`1`, no exemplo acima).

Uma forma ainda mais bacana de executar o comando `test` no Bash seria:

```
[[ -x nome_do_arquivo ]]; echo $?
```

Vejamos nos exemplos:

```
:~$ [[ -x teste.sh ]]; echo $?
0

:~$ [[ -x arquivo.txt ]]; echo $?
1
```

> ***Nota:*** *O comando `test` também pode ser representado com apenas um par de colchetes (`[ expressão ]`), mas o Bash oferece recursos adicionais quando utilizamos pares duplos de colchetes.*

3.5 – Nosso primeiro script!

Nosso primeiro script será um pequeno aprimoramento do exemplo do tópico anterior. Antes, porém, precisamos criar o arquivo e torná-lo executável.

Se você ainda não criou uma pasta de testes para o nosso curso, essa é a sua chance!

```
:~$ mkdir ~/testes
```

Com a pasta criada, você pode mudar o diretório corrente para ela:

```
:~$ cd ~/testes
:~/testes$
```

Agora crie o arquivo do nosso primeiro script:

```
:~/testes$ >> infos
```

E torne-o executável com o comando:

```
:~/testes$ chmod +x infos
```

Pronto, nosso arquivo já existe e é executável! Mas ele ainda está vazio, e é isso que nós resolveremos agora:

```
:~/testes$ nano infos
```

Com o editor aberto, vamos digitar as linhas do código da forma que está abaixo:

```
#!/usr/bin/env bash

whoami
hostname
```

```
uptime -p
uname -rms
```

Quando terminarmos, vamos salvar o arquivo (`Ctrl+S`) e sair do `nano` (`Ctrl+X`).

Se você digitou tudo direitinho e seguiu todos os passos anteriores, nós já podemos executar o nosso script!

```
:~/testes$ ./infos
```

O que deve apresentar na saída algo parecido com isso:

```
:~/testes$ ./infos
blau
enterprise
up 2 weeks, 3 days, 19 hours, 42 minutes
Linux 4.19.0-6-amd64 x86_64
```

Mas as informações jogadas assim no terminal não fazem muito sentido. Vamos melhorar isso!

Abra novamente o arquivo com o editor `nano` e altere o conteúdo para que fique exatamente da forma que eu mostro aqui:

```
#!/usr/bin/env bash

clear

echo ""
echo "Informações super importantes!"
echo "--------------------------------------"

echo -n "Usuário : "
whoami

echo -n "Hostname: "
hostname

echo -n "Uptime  : "
uptime -p
```

```
echo -n "Kernel  : "
uname -rms

echo "-------------------------------------"
echo ""
```

Salve, saia do editor e execute seu script melhorado:

```
:~/testes$ ./infos
```

Se você digitou tudo direitinho, o resultado deve ficar mais ou menos assim:

```
Informações super importantes!
-------------------------------------
Usuário : blau
Hostname: enterprise
Uptime  : up 3 hours, 18 minutes
Kernel  : Linux 4.19.0-6-amd64 x86_64
-------------------------------------
```

De tudo que aparece nesse primeiro script, as novidades são:

- `clear` - comando para "limpar" o terminal.
- `echo -n` - onde `-n` é uma opção do comando `echo` que impede que ele crie uma nova linha ao final da string exibida.

> ***Nota:*** *Nos próximos tópicos nós veremos outras formas de trabalhar com o `echo` sem precisar da opção `-n` para concatenar strings com saídas de comandos.*

3.6 - Um script para criar scripts

Vamos pensar um pouco em tudo que vimos neste tópico e em todos os comandos que teremos que executar para criar um novo script:

Etapa	**Comandos**
Decidir onde criar o arquivo	*Nenhum comando diretamente associado*

Etapa	**Comandos**
Criar um novo arquivo	`>> nome_do_arquivo`
Tornar o arquivo executável	`chmod -x nome_do_arquivo`
Editar o arquivo	`nano nome_do_arquivo`
Escrever a linha do interpretador	`#!/usr/bin/env bash`

Para mim, isso se parece exatamente com o tipo de processo repetitivo que merece ser transformado em script! Então, vamos criá-lo!

3.6.1 - Decidindo onde criar o novo script

Para começar, um script para criar scripts só faz sentido se pudermos chamá-lo da pasta em que estivermos querendo criar um novo script. Sendo assim, esta é uma boa oportunidade de criar uma pasta para os nossos scripts que podem ser acessados de qualquer local. Para isso, crie uma pasta chamada `bin` e entre nela:

```
:~$ mkdir ~/bin
:~$ cd ~/bin
:~/bin$
```

Alternativamente, se você não tem certeza de que a pasta `~/bin` não existe, você pode tentar criá-la com a opção `-p` do comando `mkdir`. O resultado será o mesmo, mas isso evitará erros caso a pasta já exista:

```
:~$ mkdir -p ~/bin
:~$ cd ~/bin
:~/bin$
```

Agora, vamos incluir esta pasta no seu `PATH`. Note que algumas distribuições GNU/Linux já configuram o Bash para reconhecer a pasta `~/bin` como um dos caminhos de executáveis do usuário. Para conferir, execute:

```
:~/bin$ echo $PATH
```

Se a pasta `/home/nome_do_usuario/bin` não aparecer na lista, você pode incluí-la no final do seu arquivo `~/.bashrc`.

> ***Nota:*** *O arquivo `~/.bashrc` é um arquivo de configurações pessoais do Bash. Você deve ter muito cuidado ao manipulá-lo, mas geralmente é seguro se você se limitar a incluir coisas no final dele enquanto está aprendendo.*

Abra o arquivo `~/.bashrc` com o editor `nano`, vá até o final e inclua esta linha:

```
export PATH="$PATH:/$HOME/bin"
```

> ***Nota:*** *O comando `export` serve para definir que a variável será uma variável de ambiente do shell, enquanto `$HOME` é uma variável de ambiente que contém o caminho da sua pasta de usuário sem a barra (`/`) no final.*

Salve (`Ctrl+S`) e saia do editor (`Ctrl+X`). Para aplicar as mudanças feitas em `~/.bashrc`, execute o comando:

```
:~/bin$ source ~/.bashrc
```

O comando `source` para executar, no shell atual, os comandos existentes em um arquivo. Na prática, o que estamos fazendo é recarregar as configurações em `~/.bashrc` sem precisarmos fechar e abrir novamente o terminal.

Execute novamente o comando abaixo e veja se a usa pasta `/home/nome_do_usuario/bin` aparece na lista.

```
:~$ echo $PATH
```

Se tudo estiver correto, vamos ao próximo passo.

3.6.2 - Criando o arquivo do nosso script

Ainda na pasta `~/bin`, vamos criar o arquivo do nosso script criador de scripts, a que vamos chamar de `novo-script.sh`. Em princípio, nós podemos criá-lo com o comando:

```
:~/bin$ >> novo-script.sh
```

Porém, em vez de criarmos um arquivo vazio, existe um "macete" que nos permite criar o arquivo já com a linha do interpretador, o que nos poupa de mais uma etapa do processo. A ideia básica é aproveitar o redirecionamento de saída para arquivos da seguinte forma:

```
:~/bin$ echo '#!/usr/bin/env bash' >> novo-script.sh
```

Observe que a saída do comando `echo` será a string `#!/usr/bin/env bash` seguida de uma quebra de linha, e é exatamente isso que será incluído no arquivo `novo-script.sh` através do redirecionamento (`>>`).

> ***Importante!*** *É* ***obrigatório*** *que* `#!/usr/bin/env bash` *esteja entre* ***aspas simples****! Curiosamente, a exclamação (* `!`*) é um comando do shell que nos permite buscar eventos no histórico de comandos (os comandos que você já executou) e, com aspas duplas, ele seria executado!*

Antes de continuarmos, vamos atualizar os nossos procedimentos?

Etapa	**Comandos**
Decidir onde criar o arquivo	*Nenhum comando diretamente associado*
Criar um novo arquivo	`echo '#!/usr/bin/env bash' >> nome_do_arquivo`
Tornar o arquivo executável	`chmod -x nome_do_arquivo`
Editar o arquivo	`nano nome_do_arquivo`
Escrever a linha do interpretador	`#!/usr/bin/env bash`

3.6.3 - Tornando nosso script executável

Em seguida, vamos tornar o script `novo-script.sh` executável:

```
:~/bin$ chmod +x novo-script.sh
```

3.6.4 - Editando o nosso script

Chegou a hora de abrirmos o nosso arquivo recém-criado para escrevermos algum código nele. Se você seguiu todos os passos até aqui, ao abrir o arquivo com o comando...

```
:~/bin$ nano novo-script.sh
```

Você deve encontrar apenas esta linha:

```
#!/usr/bin/env bash
```

Agora, nós só temos que escrever as instruções relativas a cada um dos passos necessários para a criação de um novo script:

```
#!/usr/bin/env bash

echo '#!/usr/bin/env bash' >> nome_do_arquivo
chmod + x nome_do_arquivo
nano nome_do_arquivo
```

Mas... Espera um pouco!

Como o script vai saber o nome do arquivo que eu pretendo criar?

Uma das formas de resolver esse problema é recorrendo a outra variável especial do shell que está no mesmo grupo da variável `$0`, que nós vimos no fim do primeiro tópico. Este grupo de variáveis especiais é chamado de **parâmetros posicionais** (nós teremos um tópico completo dedicado a eles), e correspondem ao nome do programa ou script em execução (`$0`) e tudo que for passado para ele como argumento após o nome (`$1`, `$2`, `$3`, etc...). Ou seja, nós podemos executar o nosso script da seguinte forma:

```
:~$ novo-script nome_do_arquivo
```

Onde `nome_do_arquivo`, seja ele qual for, ficará armazenado no parâmetro posicional `$1`, e nós poderemos acessá-lo de dentro do nosso script assim:

```
#!/usr/bin/env bash

echo '#!/usr/bin/env bash' >> $1
chmod + x $1
nano $1
```

> ***Nota:*** *Cada argumento passado para um script será armazenado nos parâmetros posicionais segundo a ordem em que forem escritos considerando-se os espaços entre eles como separadores.*

3.6.5 - Evitando problemas

Nós já garantimos que um arquivo de mesmo nome não será totalmente sobrescrito utilizando o redirecionamento `>>`. Mas, caso aconteça a coincidência dos nomes, a linha do interpretador seria escrita no final desse arquivo. Portanto, nós precisamos garantir que isso jamais aconteça.

Voltando ao comando `test` e à sua sintaxe `[[ expressão ]]`, um dos testes que podemos fazer é verificar se um determinado arquivo já existe, o que é feito através do operador `-f`. Então, vamos incluir um teste no começo do nosso script:

```
#!/usr/bin/env bash

[[ -f $1 ]] && echo "Arquivo já existe! Saindo..." && exit 1

echo '#!/usr/bin/env bash' >> $1
chmod + x $1
nano $1
```

Nesta nova linha, nós testamos se o nome de arquivo armazenado em `$1` já existe na pasta atual. Se existir, o comando `test` irá retornar um status de saída `0` (sucesso). Depois do teste em si, nós temos o conector lógico `&&`, cuja função é permitir a execução do comando que vier depois dele **apenas se o comando anterior foi executado com sucesso** (retornou um status de saída `0`). Consequentemente, se o teste retornar status `0`, o comando `echo "Arquivo já existe! Saindo..."` será executado e, em seguida, o segundo conector lógico `&&` permitirá a execução do comando `exit 1`, que encerrará a execução do script e retornará para o shell o status

`1`, indicado por nós.

No fim das contas, toda esta linha do código serve para impedir que o script continue sendo executado caso exista outro arquivo na mesma pasta com um nome igual ao do que estamos tentando criar.

Mas, ainda falta prevenir um outro tipo de problema: se o nome do novo arquivo contiver espaços, apenas a primeira parte desse nome será armazenada em `$1`. Além disso, nós podemos nos esquecer de informar um nome para o arquivo do novo script, o que também precisa ser tratado de alguma forma.

Felizmente, o comando `test` pode nos ajudar novamente! Desta vez, para verificar se o número de argumentos passados para o script no prompt de comandos está correta. Mais de um argumento, ou nenhum argumento, não são opções válidas. Então, só nos serve continuar executando o script se houver um e apenas um argumento.

Sendo assim, vamos incluir mais uma linha antes das outras no nosso script:

```
#!/usr/bin/env bash

[[ $# -ne 1 ]] && echo "Digite o nome de apenas um arquivo! Saindo..." &&
exit 1

[[ -f $1 ]] && echo "Arquivo já existe! Saindo..." && exit 1

echo '#!/usr/bin/env bash' >> $1
chmod + x $1
nano $1
```

O que acontece depois do teste é idêntico ao que vimos no teste de arquivo existente. A diferença aqui está no teste em si. Nesta nova linha, `[[ $# -ne 1 ]]`, a variável especial do shell, `$#`, armazena a quantidade de argumentos passados para um script. Em seguida, nós utilizamos o operador de comparação numérica, `-ne` (*not equal*, "diferente"), para verificar se a quantidade de argumentos é diferente de `1`. Se houver zero argumentos ou mais de um argumento, o teste retornará status `0` (sim, o valor em `$#` é diferente de `1`), fazendo com que a mensagem seja exibida e o script seja encerrado imediatamente, retornando status de saída `1` para o shell.

3.6.6 - Agora temos dois scripts!

O nosso script criador de scripts está pronto! Basta salvar e utilizar à vontade. A essa altura, você já deve estar entendendo bem melhor como pode ser útil criar scripts. Além disso, nós demos vários saltos no conteúdo dos próximos tópicos:

- Variáveis de ambiente `HOME`, `PATH` e o comando `export`
- Parâmetros posicionais
- Variáveis especiais `$?` e `$#`
- Comando `test` e alguns dos seus operadores
- O conector lógico `&&`
- Redirecionamentos para arquivos `>` e `>>`

Mas, não se preocupe. Todos eles serão bastante explorados no momento certo.

Aula 4 – Variáveis

4.1 - Conceito

Uma variável é basicamente um **nome**, representado por uma **string** (basicamente, uma sequência de caracteres), que aponta para uma determinada informação, a que chamamos de **valor**. Com o shell, nós podemos criar, atribuir e eliminar variáveis.

Para criar uma variável, portanto, nós precisamos definir um nome e, em seguida, atribuir a ele um valor, o que é feito no shell através do operador de atribuição `=`.

Exemplos:

```
:~$ fruta="banana"
:~$ nota=10
:~$ nomes=("João" "Maria" "José")
:~$ retorno=$(whoami)
```

> *Observe que não existem espaços antes ou depois do operador de atribuição!*

Além dessa forma explícita de criar uma variável, nós também podemos atribuir valores a nomes através de comandos. É o caso, por exemplo, do comando interno `read`, que lê a informação que o usuário digita no terminal e a atribui a um nome...

```
:~$ read -p "Digite seu nome: " nome
Digite seu nome: Renata
:~$ echo $nome
Renata
:~$
```

4.2 - Nomeando variáveis

Por uma questão de boas práticas, os nomes das variáveis podem conter apenas **caracteres maiúsculos e minúsculos de A a Z** (sem acentos ou cedilha), **números**

(nunca no começo!) e o **sublinhado** (`_`). Abaixo, podemos ver alguns exemplos de nomes válidos:

```
fruta  FRUTA    FRUTA1   Fruta1
_fruta _FRUTA   FRUTA_1  FRU_TA_1
```

Já esses nomes retornariam um erro no shell...

```
CAPÍTULO  2nome  nome!       RET*
1VAR      VAR 1  nome-aluno  ação
```

4.3 - Tipos de variáveis

Diferente de algumas outras linguagens, as variáveis não são declaradas especificando o tipo de valor que carregam. Ou seja, para criar uma variável do tipo **string**, basta criar um nome e atribuir a ele uma sequência de caracteres entre aspas duplas ou simples:

```
:~$ var1="Isso é uma string..."
:~$ var2='Isso também!'
```

Se você quiser criar uma variável numérica, basta atribuir um número ao nome...

```
:~$ valor1=10
:~$ valor2=23154
```

> *Mesmo não sendo algo diretamente relacionado com variáveis, é bom saber desde já que **o bash não tem suporte nativo a operações matemáticas com ponto flutuante** (números com casas decimais), apenas com números inteiros. Para esse tipo de operação, nós temos que recorrer a recursos externos, como a linguagem de cálculo o **bc** ou o utilitário de calculadora **dc**.*

4.4 - Variáveis vetoriais

Além das variáveis **escalares** (quando um nome aponta para apenas um valor) que

nós vimos acima, nós também podemos criar variáveis **vetoriais** (quando um nome aponta para uma coleção de valores), as chamadas **arrays** (ou **matrizes**, em português).

Exemplos...

```
:~$ alunos=("João" "Maria" "Renata")
```

Ou assim...

```
:~$ alunos[0]="João"; alunos[1]="Maria"; alunos[2]="Renata")
```

Repare que o nome da variável é "**alunos**", e ela contém três valores identificados pelos índices de `0` a `2`.

> *Quando eu digo que não precisamos declarar variáveis no Bash, não estou dizendo que isso não seja possível! O ponto aqui é que, para o Bash, o conceito de **declaração** é um pouco diferente de outras linguagens. Para declarar (no sentido do Bash) o tipo da variável, nós utilizamos o comando interno **declare**.*

Geralmente, porém, quando falamos de tipos de variáveis, no contexto do shell, nós estamos referindo a conceitos como:

Conceito	Descrição
Variáveis locais	Presentes na sessão atual do shell e disponíveis apenas para ele mesmo.
Variáveis de ambiente	Disponíveis para todos os processos filhos da sessão atual do shell.
Variáveis do shell	As variáveis especiais que o próprio shell define e que podem ser tanto locais quanto de ambiente.

Dentro de um script, as variáveis ainda podem ser **globais** (quando estão disponíveis para todas as instruções do mesmo script) ou **locais** (quando só podem ser acessadas por algum bloco de código específico).

> *Por padrão, todas as variáveis criadas dentro de um script são, para o próprio script,*

> *globais. Para torná-las locais, é preciso utilizar o comando interno `local`. Nós voltaremos a isso quando estivermos falando de **funções**.*

4.5 - Variáveis inalteráveis (read-only)

Normalmente, nós podemos criar variáveis com um determinado valor e mudá-lo à vontade quando necessário...

```
:~$ fruta="banana"
:~$ echo $fruta
banana
:~$ fruta="laranja"
:~$ echo $fruta
laranja
```

Porém, existem casos onde precisamos garantir que um determinado valor não seja mudado. Com o comando interno `readonly`, o shell retornará um erro caso ocorra uma tentativa de alterar o valor da variável.

```
:~$ fruta="banana"
:~$ readonly fruta
:~$ fruta="laranja"
bash: fruta: a variável permite somente leitura
```

4.6 - Destruindo variáveis

Quando criamos uma variável, ela entra para uma lista e passa a ser rastreada pelo shell enquanto durar a sessão. Para remover a variável dessa lista, nós utilizamos o comando interno `unset`.

```
:~$ fruta="banana"
:~$ echo $fruta
banana
:~$ unset fruta
:~$ echo $fruta
```

```
:~$
```

Variáveis inalteráveis não podem ser destruídas! A única forma de deletar uma variável read-only é encerrando (ou reiniciando) a sessão do shell.

4.7 - Atribuindo saídas de comandos a variáveis

Também é possível atribuir a saída de um comando a uma variável utilizando uma expansão **substituições de comando**, que nós veremos em detalhes mais adiante no curso. Para isso, basta utilizar uma das sintaxes abaixo:

```
:~$ usuario=$(whoami)
:~$ echo $usuario
arthur
```

Ou então...

```
:~$ maquina=`hostname`
:~$ echo $maquina
excalibur
```

4.8 - Acessando os valores das variáveis

A essa altura, de tanto utilizarmos o comando `echo`, você já deve saber como acessar o valor de uma variável. Mas vamos "formalizar" esse conhecimento: para ter acesso ao valor armazenado numa variável, basta acrescentar o caractere `$` ao início de seu nome, como neste exemplo:

```
:~$ read -p "Digite seu nome: " nome
Digite seu nome: Renata
:~$ mensagem="Olá, $nome!"
:~$ echo $mensagem
Olá, Renata!
```

No caso das variáveis vetoriais (**arrays**), porém, o procedimento é um pouco diferente, dependendo do que queremos ler da variável. Observe o exemplo:

```
:~$ nomes=("João" "Renata" "José")
:~$ echo $nomes
João
```

Dos três nomes dessa **array**, o shell só retornou o primeiro. Isso acontece porque, sem uma sintaxe especial, o shell irá tratar a variável `nomes` como se ela fosse uma **variável escalar**, e não uma **array**. Para acessar o conteúdo de uma array, nós utilizamos o seguinte formato:

```
${nome_da_variável[índice]}
```

> *A rigor, a notação padrão para acesso (expansão) ao valor de uma variável é `$ {nome}`, mas as chaves são opcionais quando trabalhamos com variáveis escalares.*

Aplicando essa ideia ao exemplo, para obter o valor de índice `1`, nós faríamos...

```
:~$ echo ${nomes[1]}
Renata
```

Isso porque os índices de uma array são contados a partir de zero.

Aula 5 – Variáveis Especiais

5.1 - Caracteres com significado especial

Uma coisa interessante sobre o shell/bash, é que ele utiliza alguns caracteres simples para representar os nomes de algumas variáveis que ele mesmo define durante uma sessão.

Por exemplo, o caractere `$`, além de ser o prefixo indicador de acesso a uma variável, também é o nome da variável em que o shell armazena o **PID** da sua própria sessão corrente. Ou seja, se executarmos...

```
:~$ echo $$
```

O retorno será o PID da sessão corrente do shell.

Aliás, esta é uma ótima oportunidade para comprovarmos o fato de um script em execução ser uma sessão do shell diferente daquela em que ele é chamado!

Crie este script com o nome `teste_pid`:

```
#!/usr/bin/env bash
echo "- O meu PID é: $$"
exit 0
```

Agora, execute-o no terminal desta forma (não se esqueça do `chmod +x ...`):

```
:~$ ./teste_pid; echo "- O PID desta sessão do shell é: $$"
```

Aqui, o resultado será parecido com isso:

```
:~$ ./teste_pid; echo "- O PID desta sessão do shell é: $$"
- O meu PID é: 31793
- O PID desta sessão do shell é: 15647
:~$
```

5.2 - Obtendo o status de saída do último comando

Outra variável especial muito útil é a `$?`. Com ela, nós podemos verificar o status de saída do último comando executado no terminal ou a partir de um script. Se o comando tiver sido executado com sucesso, o retorno será `0` (zero). Caso contrário, a variável `$?` armazenará qualquer valor diferente de zero.

Por exemplo...

```
:~$ apt install banana
# mensagem de erro #
:~$ echo $?
100

:~$ echo "Olá mundo"
Olá mundo
:~$ echo $?
0
```

O primeiro comando encerrou com erro 100 (e não usei `sudo` e o pacote `banana` não existe). Já o segundo comando encerrou com sucesso, portanto o seu status de saída foi zero. Isso é muito útil, especialmente dentro de scripts, para determinar o sucesso da execução de algum comando e, a partir dessa informação, tomar uma decisão.

Por exemplo, na primeira aula nós vimos que o comando interno `help` retorna um erro caso o comando de que buscamos informações não seja um comando interno, e agora nós podemos criar um script justamente para esta finalidade!

Vamos chamar o nosso script de `checa_builtin` e, para começar, vamos digitar os seguintes comandos nele:

```
#!/usr/bin/env bash

# Aqui nós executamos o comando "help"...
help ls &> /dev/null

# E aqui nós testamos a saída com o comando "test"...
[[ $? -eq 0 ]] && echo "ls é interno!" || echo "ls não é interno!"

exit 0
```

Antes de começarmos a análise deste código, você deve saber que as linhas que iniciam com cerquilha (`#`) são chamadas de "comentários" e elas são ignoradas pelo shell. Elas existem justamente para que o programador faça comentários no código, o que serve tanto para que ele se lembre do que pretendia fazer com determinado comando, quanto para facilitar a compreensão do código quando ele for lido por outras pessoas.

Vamos, então, ao primeiro comando:

```
help ls &> /dev/null
```

Aqui nós pedimos ao `help` que nos mostre as informações do comando `ls`. Como não estamos interessados na descrição, e sim se o comando `help` vai sair com erro, nós vamos redirecionar a saída de tudo que apareceria no terminal (um erro ou a informação sobre o comando `ls`) para o arquivo `/dev/null`.

> *De forma bem simplificada, o arquivo `/dev/null` pode ser entendido como o "limbo" do shell, um local para onde podemos mandar qualquer coisa que nós queremos que desapareça.*

O **redirecionamento da saída** é feito com o operador `>` (que nós já vimos quando falamos da criação de arquivos vazios) acrescido do símbolo `&`, indicando que queremos redirecionar tanto as mensagens normais que seriam exibidas no terminal (saída padrão, ou STDOUT) quanto as mensagens de erro (saída padrão de erro, ou STDERR).

Não se preocupe, nós teremos uma aula inteira sobre redirecionamentos no módulo avançado, quando falarmos dos "descritores de arquivos". No momento, basta saber que...

```
&> /dev/null
```

Impedirá que qualquer mensagem retornada pelo comando "help" seja exibida no terminal. Porém, mesmo sem exibir mensagem nenhuma, o sucesso ou o erro de execução do comando `help` ficará armazenado na variável `$?`, o que nos leva à próxima linha do script, onde nós utilizamos o comando interno `test`.

```
[[ $? -eq 0 ]]
```

O comando `test` avalia expressões condicionais, ou seja, ele retorna sucesso (`0`) caso uma expressão seja verdadeira, ou erro (`1`) se a expressão for falsa.

A expressão, no nosso caso, é uma comparação. Estamos comparando o valor em `$?` com o valor numérico `0` (zero), por isso utilizamos o operador de comparação numérica `-eq` (de *equal*, ou igual).

> *O comando `test` é um dos builtins mais interessantes e poderosos do shell/bash! Com ele podemos testar uma grande infinidade de coisas, da existência de arquivos e pastas a condições lógicas. Vale muito a pena conferir nos manuais o que ele é capaz de fazer executando: `help test` e `help [[`.*

O comando `test` pode ser executado das seguintes formas:

```
[[ $? -eq 0 ]]
```

Ou então...

```
test $? -eq 0
```

Mas, a primeira forma, além de ter alguns recursos a mais, só funciona no Bash.

Ao lado do teste, nós utilizamos os *operadores de concatenação condicional* `&&` (AND/E) e `||` (OR/OU).

Sua principal função é executar ou não o comando seguinte de acordo com o status de saída do comando que os preceder. Se o comando antes de `&&` retornar um status `0` (sucesso), o comando seguinte será executado. Caso contrário, o comando a ser executado será aquele que vier depois do operador `||`:

```
comando1 && comando_sucesso || comando_erro
```

Por isso, no script `checa_builtin`, nós utilizamos...

```
[[ $? -eq 0 ]] && echo "ls é interno!" || echo "ls não é interno!"
```

Ou seja, se o comando `test` retornar "sucesso" (se `$?` for mesmo igual a zero), o comando a ser executado será...

```
echo "ls é interno!"
```

Caso retorne "erro", o comando executado será...

```
echo "ls não é interno!"
```

Entendido o funcionamento do script, vamos executá-lo.

```
:~$ ./checa_builtin
ls não é interno!
```

O que era de se esperar.

5.3 - Passando argumentos para o scripts

Mas seria muito chato ter que editar o script toda vez que quiséssemos testar se um comando é ou não é interno! E se fosse possível executar o script `checa_builtin` informando, na própria linha de comando, o comando que queremos testar?

Pois bem, isso é possível, graças à variável especial `$n`, onde `n` é um número inteiro a partir de `0`.

> *Nós estamos utilizando o termo "argumentos" por uma questão de similaridade com outras linguagens, mas o termo correto para isso no Bash é **parâmetros** (ou "valores"). É por este motivo que, tecnicamente, essas variáveis especiais são chamadas de **parâmatros posicionais**.*

Aliás, nós utilizamos essa variável especial na primeira aula, quando queríamos descobrir qual era o shell em execução, lembra?

```
:~$ echo $0
/bin/bash
```

Na ocasião, nós vimos que `$0` armazena o nome do programa/script em execução (no

caso, era o bash). Os demais números, começando do `1`, irão representar, em ordem de ocorrência, os argumentos passados para o script. Desta forma:

```
:~$ script argumento_1 argumento_2 ... argumento_N

$0 => "script"
$1 => "argumento_1"
$2 => "argumento_2"
...
$n => "argumento_N"
```

Então, de volta ao script `checa_builtin`, nós podemos combinar que ele será executado assim:

```
:~$ ./checa_builtin comando_a_ser_testado
```

O próprio shell cuidará de armazenar `comando_a_ser_testado` na variável especial `$1`, e nós poderemos utilizá-la dentro do nosso script.

Então, vamos às alterações!

Para isso, nós vamos substituir todas as ocorrências de `ls` por `$1`...

```
#!/usr/bin/env bash

# Aqui nós executamos o comando "help"...
help $1 &> /dev/null

# E aqui nós testamos a saída com o comando "test"...
[[ $? -eq 0 ]] && echo "$1 é interno!" || echo "$1 não é interno!"

exit 0
```

Vamos ver o que acontece quando testamos os comandos (será?) `cat` e `cd`?

```
:~$ ./checa_builtin cat
cat não é interno!

:~$ ./checa_builtin cd
cd é interno!
```

5.4 - Contando o número de argumentos

Nós sabemos que o nosso script depende do argumento `comando_a_ser_testado` para funcionar corretamente. Mas, e se ele cair nas mãos de um usuário desavisado?

Por isso, nós precisamos testar, no mínimo, se a quantidade de argumentos passados na execução do script corresponde ao número de argumentos que nós esperamos (no caso, apenas um).

Felizmente, o shell também armazena essa informação na variável especial `$#`.

No nosso caso, a quantidade esperada é de apenas um argumento, e nós podemos utilizar novamente o comando `test` para verificar se essa quantidade bate...

```
[[ $# -ne 1 ]] && mensagem_de_erro && exit 1
```

Nesse trecho, nós utilizamos o operador de comparação numérica `-ne` (de *not equal*, ou "diferente"). Ou seja, se a quantidade de argumentos (`$#`) for diferente de `1`, uma mensagem de erro deve ser enviada e, em seguida, o script deve ser interrompido com o status de saída `1` (indicando um erro).

Então, vamos alterar o script. Mas, para que as linhas não fiquem longas demais, eu vou armazenar a mensagem de erro numa variável chamada `msg`...

```
#!/usr/bin/env bash

# Mensagem de erro...
msg="É preciso informar um comando válido!"

# Teste de quantidade de argumentos...
[[ $# -ne 1 ]] && echo $msg && exit 1

# Aqui nós executamos o comando "help"...
help $1 &> /dev/null

# E aqui nós testamos a saída com o comando "test"...
[[ $? -eq 0 ]] && echo "$1 é interno!" || echo "$1 não é interno!"

exit 0
```

Fazendo alguns testes...

```
:~/scripts$ ./checa_builtin
É preciso informar um comando válido!

:~/scripts$ ./checa_builtin cat
cat não é interno!

:~/scripts$ ./checa_builtin test
test é interno!
```

5.5 - Um pequeno resumo

Ainda existem algumas outras variáveis especiais, mas vamos com calma, porque você já tem novidades demais para assimilar. Então, por enquanto, vamos apenas relembrar as variáveis do shell que nós vimos até aqui.

Variável	Descrição
`$?`	Armazena o status de saída do último comando executado.
`$$`	Armazena o PID da sessão corrente do shell.
`$0`	Armazena o nome do arquivo do programa ou do script sendo executado.
`$n`	(a partir de 1) Armazena os valores (parâmetros) passados para o script, onde “n” é um número inteiro, positivo, correspondente à posição dos parâmetros na linha de comando (daí “posicionais”).
`$#`	Armazena o número de parâmetros passados para o script.

Aula 6 – Vetores

6.1 - Um nome, muitos valores

Como nós já vimos, para criar uma variável no bash/shell, nós escolhemos um nome e atribuímos a ele um valor. Esse valor pode mudar ao longo do tempo enquanto o script é executado, mas o nome sempre representará apenas um valor de cada vez.

```
:~$ fruta="banana"
:~$ echo $fruta
banana
:~$ fruta="laranja"
:~$ echo $fruta
laranja
```

E se nós quiséssemos representar, não uma fruta, mas um conjunto de frutas? Uma ideia seria atribuir cada uma das frutas a um nome diferente:

```
:~$ fruta01="banana"
:~$ fruta02="laranja"
:~$ fruta03="abacate"
:~$ fruta04="goiaba"
```

Apesar de ser uma solução até intuitiva, isso não seria muito prático, principalmente se quiséssemos percorrer todos os valores correspondentes ao conjunto de “frutas”. Felizmente, porém, o Bash nos permite criar variáveis onde um único nome pode apontar para diversos valores, e esse tipo de varável é chamada de **variável vetorial**, ou **array**.

6.2 – Criando vetores indexados

O processo de criação de vetores indexados é tão simples quanto a criação de variáveis escalares. A sintaxe geral é a seguinte:

```
nome[índice]=valor
```

Onde **nome** é o nome da array e **índice** pode ser um número inteiro positivo ou uma string. Por exemplo:

```
fruta[0]="banana"
fruta[1]="laranja"
fruta[2]="abacate"
```

Ou ainda...

```
fruta[doce]="banana"
fruta[azeda]="laranja"
fruta[verde]="abacate"
```

Quando o índice é um número inteiro, nós chamamos esse tipo de variável **vetor indexado**, quando for uma string, nós teremos um **vetor associativo**.

Especificamente no Bash, uma array indexada pode ser criada da seguinte forma:

```
fruta=("banana" "laranja" "abacate")
```

Onde cada valor receberá automaticamente um índice correspondente à sua ordem de ocorrência, sempre contando a partir de `0` (zero).

Finalmente, caso você queira apenas inicializar uma array antes de atribuir qualquer valor a ela, basta utilizar uma das sintaxes abaixo:

```
# Arrays indexadas
declare -a nome_da_array

# Arrays associativas
declare -A nome_da_array
```

> *Declarar arrays indexadas é opcional, mas **declarar arrays associativas é obrigatório**.*

Por exemplo:

```
:~$ declare -A fruta
```

```
:~$ fruta[verde]=abacate
:~$ fruta[vermelha]=cereja
:~$ fruta[amarela]=banana
```

Se uma variável associativa não for declarada, ao acessá-la, apenas o valor do último elemento será retornado, pois toda a parte entre colchetes será ignorada no processo de atribuição dos valores.

Exemplo:

```
:~$ fruta[verde]=abacate
:~$ fruta[vermelha]=cereja
:~$ fruta[amarela]=banana
:~$ echo ${fruta[verde]}
banana
```

6.3 – Acessando os valores da array

O processo de leitura dos valores em uma array é um pouco diferente do que nós vimos até aqui. Por se tratar de uma identificação composta por um nome e um índice, o shell precisará de algo que indique que estamos falando de todo um conjunto de identificadores, e não apenas de um nome seguindo de uma string.

Veja o exemplo...

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo $fruta
banana
:~$ echo $fruta[2]
banana[2]
```

Invocando apenas `$fruta`, o shell retorna somente o valor correspondente ao primeiro elemento da array (`banana`). Quanto tentamos ser mais específicos fazendo referência ao índice da fruta que nos interessa, `$fruta[2]`, o retorno foi o mesmo de antes acrescido da string `[2]`.

Para que o shell entenda que estamos falando de um nome e um índice, todo esse conjunto deve estar entre chaves, como na sintaxe geral abaixo:

```
${nome[índice]}
```

Aplicando ao exemplo...

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo ${fruta[2]}
abacate
```

6.4 – Lendo todos os valores de uma array

Nós podemos obter todos os valores de uma array de duas formas: como uma única string ou como strings separadas.

Para obter uma string composta de todos os elementos de uma array, nós utilizamos o caractere asterisco (`*`) no índice...

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo ${fruta[*]}
banana laranja abacate
```

Neste caso, `banana laranja abacate` formam uma única string. Se quisermos obter os mesmos valores, mas como strings separadas, nós utilizamos o caractere arroba (`@`) no índice...

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo ${fruta[@]}
banana laranja abacate
```

Embora não seja possível notar na saída do terminal, neste caso estamos diante de

três strings distintas, e isso será muito útil quando precisarmos percorrer os valores de uma array, o que nós veremos mais adiante no curso, quando falarmos das estruturas de repetição.

Se, em vez dos valores, nós quisermos os índices de cada elemento, basta acrescentar o caractere de exclamação (`!`) antes do nome da array:

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo ${!fruta[@]}
0 1 2
```

6.5 – Listando os valores da array por faixas de índices

É possível listar os valores numa array a partir de um determinado ponto utilizando a sintaxe...

```
${nome[@]:início}
```

Onde **início** é o índice a partir do qual queremos obter a lista de valores.

Por exemplo:

```
:~$ fruta=("banana" "laranja" "abacate" "limão" "abacaxi")
:~$ echo ${fruta[@]:2}
abacate limão abacaxi
```

Também é possível obter uma faixa de valores indicando os índices de **início** e **fim**:

```
${nome[@]:início:fim}
```

Por exemplo...

```
:~$ fruta=("banana" "laranja" "abacate" "limão" "abacaxi")
:~$ echo ${fruta[@]:1:3}
laranja abacate limão
```

6.6 – Descobrindo o número de elementos de uma array

Nós podemos descobrir a quantidade de elementos que uma array possui, o que é feito incluindo o caractere cerquilha (`#`) antes do seu nome, por exemplo...

```
:~$ fruta[0]="banana"
:~$ fruta[1]="laranja"
:~$ fruta[2]="abacate"
:~$ echo ${#fruta[@]}
3
```

Aula 7 – Concatenação de Strings

7.1 - Expansão de parâmetros

Sob certas condições, seja no terminal ou em scripts, quando fazemos referência a uma variável acrescentando o prefixo `$` ao seu nome, o Bash "entende" que o que nós queremos é o valor que essa variável armazena e realiza uma **substituição**. Sendo mais exato, este processo é chamado de **expansão de parâmetros**.

Observe os exemplos abaixo:

```
:~$ cor="verde"

# Sem aspas...
:~$ echo $cor
verde

# Com aspas duplas...
:~$ echo "$cor"
verde

# Com aspas simples...
:~$ echo '$cor'
$cor

# Com aspas duplas e caractere de escape...
:~$ echo "\$cor"
$cor
```

Repare que, nos dois últimos casos, o Bash não expandiu a variável e tratou `$cor` literalmente como uma string. Isso acontece porque as aspas (ou a ausência delas) e o caractere de escape (`\`) também possuem um significado especial na interpretação de comandos, o que pode ser resumido em algumas "regrinhas" simples.

Condição	Comportamento
Sem Aspas	Variáveis são expandidas, mas caracteres invisíveis (quebras de linha, múltiplos espaços, tabulações, etc) são descartados e substituídos por um espaço.

	No terminal, se o histórico estiver habilitado, o caractere `!` (exclamação) no início da string será interpretado como um comando de busca de eventos e executado. É possível escapar `$`, o acento grave, `\`, aspas duplas e caracteres invisíveis. A exclamação no início da string pode ser escapada, mas o caractere de escape também seria retornado como parte dessa string.
Aspas Duplas	Variáveis são expandidas e caracteres invisíveis são preservados. No terminal, se o histórico estiver habilitado, o caractere `!` (exclamação) no início da string será interpretado como um comando de busca de eventos e executado. É possível escapar `$`, o acento grave, `\`, aspas duplas e caracteres invisíveis. A exclamação no início da string pode ser escapada, mas o caractere de escape também seria retornado como parte dessa string.
Aspas Simples	Variáveis não são expandidas, caracteres invisíveis não são preservados, e todos os caracteres especiais, inclusive a exclamação no início da string, são tratados como caracteres literais. As aspas simples não podem ocorrer dentro de aspas simples, pois não haveria como escapá-las.
Escape	O caractere de escape (`\`) faz com que os caracteres com significado especial para o Bash sejam interpretados literalmente, a não ser quando ocorre dentro de aspas simples ou ele mesmo é escapado.

7.2 - Inserindo strings em strings

Uma das formas mais comuns de concatenação, é quando nós utilizamos as variáveis para reservar o espaço (um *placeholder*) no local onde uma informação deve ser inserida.

Veja o exemplo:

```
:~$ objeto="celular"
:~$ echo "Maria tinha um $objeto."
Maria tinha um celular.

:~$ objeto="lápis"
:~$ echo "Maria tinha um $objeto."
Maria tinha um lápis.

:~$ objeto="carneirinho"
:~$ echo "Maria tinha um $objeto."
Maria tinha um carneirinho.
```

Nos três casos, a variável referenciada (`$objeto`) reservou o espaço onde os valores `celular`, `lápis` e `carneirinho` deveriam ser inseridos.

Veja outro exemplo, agora em um script que nós vamos chamar de `piada-ruim`:

```
#!/usr/bin/env bash

ben=10
leonardo=20
joaosinho=30

# Observe o caractere de escape (\) usado para impedir que a
# quebra de linha seja interpretada pelo shell!

frase="No rateio da conta, o Leonardo dá $leonardo, \
o Ben $ben, e o Joãosinho $joaosinho."

echo $frase
```

Quando executado, você veria isso no terminal:

```
:~$ ./piada-ruim
No rateio da conta, o Leonardo dá 20, o Ben 10, e o Joãosinho 30.
```

7.3 – O operador de concatenação

De forma geral, como vimos até aqui, não precisamos de nenhum operador para concatenar strings. Mas, em muitos casos, pode ser necessário anexar valores **ao final** do conteúdo de uma variável. Observe a seguinte situação:

```
:~$ fruta="banana"
:~$ cor="verde"
:~$ echo $fruta $cor
banana verde
```

Veja que bastou utilizar as duas variáveis separadas por um espaço como parâmetro do comando `echo` para que a string `banana verde` fosse exibida no terminal.

Ocorre, porém, o caso em que nós só queremos atualizar o valor armazenado na variável antes de utilizarmos alguma forma de exibi-la. Isso pode ser feito recorrendo ao operador de concatenação `+=`.

Por exemplo...

```
:~$ fruta="banana"
:~$ fruta+=" verde"
:~$ echo $fruta
banana verde
:~$ fruta+=" baratinha"
:~$ echo $fruta
banana verde baratinha
```

> ***Importante!*** *O operador `+=` só serve para fazer concatenações quando estamos trabalhando com strings! Se a variável for (de fato) numérica, ele funcionará como um* ***operador de incremento****!*

Exemplo:

```
# String...
:~$ valor=1; valor+=1
:~$ echo $valor
11

# Número inteiro...
:~$ declare -i valor; valor=1; valor+=1
:~$ echo $valor
2
```

Aula 8 – Operações Aritméticas

8.1 - As operações básicas

O Bash é capaz de realizar as operações aritméticas básicas de quatro formas, todas elas limitadas a números inteiros:

- Comando interno: `let expressão`
- Comando composto: `(( expressão ))`
- Expansão aritmética: `$(( expressão ))`
- Com valores definidos como inteiros: `declare -i nome`

Além disso, no interior dos colchetes (`[...]`) do índice de um vetor indexado, os valores são tratados como inteiros pelo Bash, o que nos permite realizar operações aritméticas com eles. Isso, por exemplo, seria perfeitamente válido:

```
vetor[n++]
```

Todos esses métodos são capazes de lidar com as operações aritméticas básicas, mas funcionam de formas muito diferentes, como veremos adiante. Antes, porém, nós precisamos conhecer os símbolos que, independente do método, irão representar as operações aritméticas que veremos neste curso.

8.2 - Operadores aritméticos

Operador	Descrição
`+`	Soma
`-`	Subtração
`*`	Multiplicação
`/`	Divisão
`%`	Módulo (resto)

`**`	Potência (exponenciação)

8.3 - Operadores de atribuição

Operador	**Descrição**
`nome=valor`	Atribui um valor a “nome”
`nome+=valor`	Soma um valor ao valor atual em “nome”
`nome-=valor`	Subtrai um valor do valor atual em “nome”
`nome*=valor`	Multiplica o valor atual em “nome” por outro valor
`nome/=valor`	Divide o valor atual em “nome” por outro valor
`nome%=valor`	Substitui o valor atual em “nome” pelo resto da divisão por outro valor
`nome++`	Pós-incremento: retorna o valor atual em “nome” e soma 1
`nome--`	Pós-decremento: retorna o valor atual em “nome” e subtrai 1
`++nome`	Pré-incremento: atribui a “nome” o seu valor atual mais 1
`--nome`	Pré-decremento: atribui a “nome” o seu valor atual menos 1

Exemplos:

```
numero=10    # Atribui o valor 10 à variável ‘numero’ (10)
numero+=3    # Soma 3 ao valor já definido em ‘numero’ (13)
numero*=2    # Multiplica o valor em ‘numero’ por 2 (26)
```

O que seria o mesmo que...

```
numero=10          # Atribui o valor 10 à variável ‘numero’ (10)
numero=numero+3    # Soma 3 ao valor já definido em ‘numero’ (13)
numero=numero*2    # Multiplica o valor em ‘numero’ por 2 (26)
```

Os operadores de pré e pós incremento e de pré e pós decremento, por sua vez, sempre efetuam a soma ou a subtração de `1` sobre o valor atual em `nome`. A diferença

entre o “pré” e o “pós” diz respeito ao momento em que o novo valor estará disponível. Veja no exemplo:

```
:~$ nota=8
:~$ echo $((nota++)) # pós-incremento
8
:~$ echo $nota
9
```

Repare que, no momento que a expansão aritmética aconteceu (quando executamos o `echo`), o valor em `nota` ainda era o valor anterior (`8`). Mas, logo em seguida, a operação foi efeutada e o valor passou a ser `9`.

No caso do pré incremento ou decremento, a operação é efetuada antes da expansão acontecer:

```
:~$ nota=8
:~$ echo $((++nota))
9
:~$ echo $nota
9
```

8.4 - Precedência

Quanto à ordem de precedência (quem é executado primeiro),os operadores aritméticos seguem as mesmas regras da matemática:

- As operações são efetuadas da esqueda para a direita;
- Primeiro efetua-se o que está entre parêntesis;
- Depois as exponenciações;
- Depois as multiplicações, divisões e módulos;
- Por último, as somas e subtrações.

8.5 – O problema do 'declare -i'

Toda variável no Bash é criada como sendo de **tipo indefinido** e seu valor é tratado como uma **string**. Além disso, com exceção das arrays associativas, nós não somos

obrigados a declarar explicitamente o tipo de valor que será armazenado numa variável. Porém, quando trabalhamos com valores numéricos, em algumas situações pode ser interessante explicitar que o valor é um inteiro. Então, nós utilizamos o comando builtin `declare`:

```
declare -i nome[=valor] # Onde “=valor” é opcional.
```

Isso ativa o **atributo de inteiro** da variável, fazendo com que ela seja tratada como um valor numérico e, portanto, capaz de participar de algumas operações aritméticas.

Por exemplo, sem o atributo de inteiro ligado...

```
:~$ nota=6
:~$ nota=$nota+2
:~$ echo $nota
6+2
```

Mas, ativando o atributo de inteiro da variável `nota`...

```
:~$ declare -i nota
:~$ nota=6
:~$ nota=$nota+2
:~$ echo $nota
8
```

Nós também poderíamos reatribuir o valor em ''nota'' incrementando seu valor:

```
:~$ declare -i nota
:~$ nota=6
:~$ nota+=2
:~$ echo $nota
8
```

Mas, os problemas começam quando tentamos utilizar outros operadores de (re)atribuição, por exemplo:

```
:~$ declare -i nota
:~$ nota=6
:~$ nota=$nota-2
```

```
:~$ echo $nota
4
:~$ nota-=2
bash: nota-=2: comando não encontrado
```

Ou seja, o único operador de (re)atribuição compacto que podemos utilizar é o `+=`, porque somente ele existe (do ponto de vista do Bash) como um operador válido para qualquer tipo de valor na linha do comando – se o valor for uma string, acontecerá uma concatenação, se for um inteiro, haverá um incremento.

Essa inconsistência (em relação às outras formas de realizar operações aritméticas) já é um sinal de que declarar o valor como inteiro não é algo trivial e merece uma atenção toda especial.

Outro problema do uso indiscriminado do `declare -i` com a única finalidade de permitir cálculos aritméticos nos scripts, é que isso afeta a legibilidade do código e, portanto, dificulta seu entendimento e a correção de erros. O principal problema está na presunção de um comportamento padrão, por exemplo:

```
:~$ nota=2+2
:~$ echo $nota
2+2
```

Isso é o esperado quando nós lemos a linha do comando acima, ou seja, o valor em `nota` é a **string 2+2**, a menos que, em algum lugar, ou em algum momento, o valor da variável tenha sido declarado como um inteiro...

```
declare -i nota

...um monte de código...

nota=2+2
echo $nota    # resultaria em '4'...
```

Essas inconsistências devem ser sempre evitadas nos nossos scripts e, além disso, o Bash tem formas muito mais interessantes de efetuar cálculos sem que os atributos de uma variável sejam alterados.

8.6 - O comando interno 'let'

O comando interno `let` (no sentido de “tornar”, em inglês), é utilizado para criar variáveis numéricas e realizar operações aritméticas com elas. O que ele faz é tratar strings numéricas que representem valores inteiros como inteiros literais sem alterar os atributos das variáveis.

Por exemplo:

```
:~$ let nota=2
:~$ echo $nota
2
:~$ nota=$nota+5
:~$ echo $nota
2+5    # O valor em 'nota' continua sendo tratado como string...
```

O mais importante aqui é você perceber que os nomes, valores e operadores utilizados com o `let` são seus **parâmetros**, e isso significa que a interpretação das expressões é feita internamente pelo próprio comando, não pelo shell.

Observe:

```
:~$ let nota=2
:~$ echo $nota
2
:~$ let nota=nota+2
:~$ echo $nota
4
```

Reparou que, no `let`, nós não precisamos utilizar o `$` para expandir o valor em `nota`? Isso só é possível porque não é o shell que está interpretando a expressão. Mas, isso também é válido:

```
:~$ let nota=2
:~$ echo $nota
2
:~$ let nota=$nota+2
:~$ echo $nota
4
```

Porque, independente do comando que recebe os parâmetros, o shell sempre realizará expansões antes de executar a linha do comando. Quer dizer, quando o `let` foi efetivamente executado, a expressão que ele recebeu como parâmetro era...

```
let nota=2+2
```

Efetuando múltiplas expressões

Você também pode passar várias expressões de uma vez para o `let`:

```
:~$ let nota=2 nota=nota+2 nota-=3
:~$ echo $nota
1
```

Cada uma dessas três expressões é um parâmetro do comando `let`.

Outra característica do comando `let`, é que ele permite a construção de expressões com espaços. Mas, como os espaços são os separadores dos parâmetros que passamos para o comando (as expressões, como no exemplo anterior), esse tipo de construção deve ser feita entre aspas.

Por exemplo:

```
:~$ let 'nota = 5'
:~$ echo $nota
5
```

Então, para efetuar múltiplas expressões com um mesmo comando, nós temos duas alternativas:

Alternativa 1: cada expressão que contenha espaços deverá vir entre aspas...

```
:~$ let 'nota = 5' ++nota
:~$ echo $nota
6
```

Alternativa 2: tudo entre aspas, mas as expressões devem ser separada por uma vírgula...

```
:~$ let 'nota = 5, ++nota'
:~$ echo $nota
6
```

8.7 – O comando composto '((expressão))'

O comando composto `((...))` é a forma "padrão" de fazer o Bash interpretar expressões aritméticas. Seu comportamento é idêntico ao do comando interno `let`, com algumas poucas diferenças. Por exemplo, diferente do `let`, múltiplas expressões devem **sempre** ser separadas por vírgulas:

```
:~$ ((nota=2, nota=nota+2, nota-=3))
:~$ echo $nota
1
```

E também não precisamos de aspas para usar espaços nas expressões:

```
:~$ ((nota = 2, nota = nota + 2, nota -= 3))
:~$ echo $nota
1
```

8.8 - A expansão aritmética '$((expressão))'

As expansões do Bash serão assunto da próxima aula, mas nós já tivemos uma prévia de algumas delas em vários momentos até aqui e está na hora de falar de mais uma: a **expansão aritmética**.

Diferente dos comandos `let` e `((...))`, de quem ela herda todas as características vistas até aqui, a expansão aritmética efetua as expressões e fornece um valor como retorno na saída padrão, o que é muito prático e poderoso.

Veja o exemplo:

```
:~$ echo $((nota = 2, nota = nota + 2, nota -= 3))
1
:~$ echo $nota
```

```
1
```

8.9 - E os números não inteiros?

Infelizmente, o Bash não dá suporte a operações com números não inteiros. Para isso, nós sempre teremos que recorrer a alguma outra ferramenta, como a calculadora programável `bc`, ou a linguagem de processamento de dados tabulares `awk` (minha solução preferida), mas isso foge do escopo deste curso. Porém, como o `bc` é uma solução muito popular, aqui está uma pequena amostra de como ele funciona.

```
# O ‘bc’ lê um arquivo ou uma string na entrada padrão
# e efetua a operação. Por padrão, ele retorna a parte
# inteira do resultado...

:~$ bc <<< '7 / 3'
2

# Para ver as casas decimais, nós utilizamos a opção '-l'...

:~$ bc -l <<< "7 / 3"
2.33333333333333333333

# Se quisermos um número fixo de casas decimais, nós temos
# que passar o argumento 'scale' junto com a operação...

:~$ bc <<< "scale=2; 7 / 3"
2.33
```

O `bc` não vem instalado por padrão em todas as distribuições GNU/Linux, mas não deve ser difícil descobrir como instalá-lo. No Debian e derivados, o comando para instalação é:

```
sudo apt install bc
```

Você encontra mais informações sobre ele, suas opções e seus operadores no manual:

```
:~$ man bc
```

Aula 9 – Expansões do Shell

9.1 - O que são expansões

Antes de ser executada, uma linha de comando passa por uma análise completa. Nessa análise, o shell procura por palavras, símbolos e operadores para dividir o comando em vários pedaços (chamados *tokens*) e determinar o que deverá ser feito com cada um deles. Em um dado momento neste processo, alguns desses pedaços serão identificados como elementos que deverão ser substituídos por algum tipo de valor, e são essas substituições que nós chamamos de **expansões**.

Através das suas expansões, o Bash oferece um poderoso conjunto de símbolos e elementos sintáticos que podem facilitar muito a nossa vida e tornar os nossos códigos mais limpos e rápidos.

Expansão	**Descrição**
Expansão de caminhos	Simplifica a referência a caminhos.
Expansão de nomes de arquivos	Símbolos para representar padrões de caracteres nos nomes de arquivos.
Expansão de chaves	Permite a geração de strings a partir de um padrão.
Substituição de comandos	Permite que a saída de um comando torne-se um valor.
Expansão aritmética	Efetua expressões aritméticas e disponibiliza o resultado como um valor.
Expansão de parâmetros	Retorna o valor de um parâmetro (variável) e ainda oferece diversas formas de manipulá-lo.
Quebra de palavras	Percorre resultados de outras expansões e decide como tratar as strings que não estão entre aspas a partir de um caractere de separação.
Remoção de aspas	Remove todas as ocorrências dos caracteres `\`, `'` e `"` que não resultem de uma expansão.

Substituição de processos	Permite enviar a saída de um comando para a entrada de outro comando que só aceitaria arquivos como argumento. **Não está disponível no shell "sh"!**

Nesta aula, nós veremos algumas das expansões de uso mais comum, e a próxima aula será totalmente dedicada à **expansão de parâmetros**.

Da lista acima, nós já conhecemos: as **substituições de comando**, as **expansões aritméticas**. Além disso, todas as expansões de variáveis escalares e vetoriais vistas até aqui são parte das chamadas **expansões de parâmetros**.

9.2 - Expansão de caminhos

Uma expansão de caminho acontece quando o shell interpreta o caractere til (`~`) como o caminho absoluto da pasta do usuário atual ou de um usuário especificado.

Exemplo 1: exibindo a pasta do usuário logado...

```
:~$ echo ~
/home/user
```

Exemplo 2: entrando na pasta 'Documentos' do usuário logado...

```
:~$ cd ~/Documentos
:~/Documentos$ pwd
/home/user/Documentos
```

Exemplo 3: exibindo a pasta do usuário 'root...

```
:~$ echo ~root
/root
```

O Bourne Shell (sh) não expande o til!

9.3 - Expansão de nomes de arquivos

Na ocorrência de metacaracetres formadores de padrões de nomes de arquivos, o shell fará uma busca na lista de arquivos da pasta indicada tentando encontrar os nomes que correspondam ao padrão. Os metacaracteres são:

Meta	Descrição
`*`	Representa zero ou mais caracteres, exceto o ponto (`.`) no início dos nomes de arquivos ocultos.
`?`	Representa um único caractere existente, exceto o ponto (`.`) no início dos nomes de arquivos ocultos.
`[...]`	Representam um conjunto ou uma faixa de caracteres que podem ocupar uma única posição no nome do arquivo.

Exemplo 1: lista todos os arquivos com nomes terminados em '.txt'...

```
:~$ ls *.txt
```

Exemplo2: lista todos os arquivos que começam com o caractere 'a'...

```
:~$ ls a*
```

Exemplo 3: lista todos os arquivos que contenham um ponto...

```
:~$ ls *.*
```

Exemplo 4: lista todos os arquivos ocultos...

```
:~$ ls .*
```

Lembre-se de que diretórios também são arquivos!

Exemplo 5: lista todos os arquivos com nomes iniciados com caracteres na faixa de 'a' até 'j'...

```
:~$ ls [a-j]*
```

Exemplo 6: lista todos os arquivos que começam apenas com os caracteres 'a' e 'j'...

```
:~$ ls [aj]*
```

Também é importante notar que, ao expandir nomes de arquivos, o shell está gerando uma lista de valores separados por espaço que serão passados como argumentos para o comando em uso (`ls`, nos exemplos). Portanto, supondo que o comando abaixo tenha encontrado três correspondências...

```
:~$ ls dir*
```

Isso seria o mesmo que...

```
:~$ ls dir1 dir2 dir3
```

O que faria o utilitário `ls` listar o conteúdo dessas três pastas.

Outra possibilidade interessante decorrente dessa expansão, é que não precisamos do utilitário `ls` para listar arquivos. Afinal, a lista de arquivos já está gerada, e nós só precisamos exibi-la (com um `echo` por exemplo):

```
:~$ echo dir*
dir1 dir2 dir3
```

9.4 - Expansão de chaves

A expansão de chaves (`{ }`)é um mecanismo para gerar strings a partir de padrões.

Exemplos:

```
:~$ echo a{3,5,7}
a3 a5 a7

:~$ echo a{0..9}
a1 a2 a3 a4 a5 a6 a7 a8 a9
```

```
:~$ echo b{a..z}nana
banana bbnana bcnana ... bxnana bynana bznana

:~$ touch teste-{01,02,03}.txt
:~$ ls
teste-01.txt teste-02.txt teste-03.txt
```

Também podemos definir "saltos" nas sequências geradas:

```
:~$ echo {0..9..2}
0 2 4 6 8
```

> ***Importante!*** *A expansão de chaves acontece antes da expansão de parâmetros, por isso não podemos utilizá-las dentro das chaves...*

```
:~$ inicio=2
:~$ echo {$inicio..20}
{2..20}    # A expansão é inválida como expansão de chaves...
```

9.5 - Quebra (split) de palavras

Depois de processar uma expansão de parâmetro, uma substituição de comando e/ou uma expansão aritmética que não tenha ocorrido entre aspas, o shell utilizará o caractere definido na variável especial `IFS` (*internal field separator*) como separador para quebrar a saída em campos (palavras).

Se a variável `IFS` não tiver sido redefinida, por padrão, sequências dos caracteres **espaço**, **tabulação** e **nova linha** no começo ou no fim da string, serão ignoradas e qualquer ocorrência deles no interior da string será utilizada como delimitador de campos. Se o valor de `IFS` for nulo, não haverá quebra de palavras.

Veja o exemplo:

```
alunos=(João Maria "Luis Carlos")
for aluno in ${alunos[*]}; do
    echo $aluno
```

```
done
```

Aqui, o comando composto `for` irá percorrer todos os elementos do vetor indexado `alunos`, armazenando temporariamente cada um dos nomes em `aluno` que, por sua vez, será exibido com o comando `echo`.

O resultado seria...

```
João
Maria
Luis
Carlos
```

Mas “Luis Carlos” deveria aparecer numa única linha, e isso não ocorreu por conta de dois problemas:

- A expansão da variável ''alunos'' foi feita sem aspas.
- A expansão foi feita usando o caractere `*` no índice da array.

Observe este outro exemplo:

```
alunos=(João Maria "Luis Carlos")
for aluno in "${alunos[@]}"; do
    echo $aluno
done
```

Aqui nós utilizamos o caractere `@` no índice e envolvemos a array com aspas, fazendo com que o resultado fosse o esperado:

```
João
Maria
Luis Carlos
```

Apenas para você entender a diferença, com as aspas e usando o asterisco no índice da array...

```
alunos=(João Maria "Luis Carlos")
for aluno in "${alunos[*]}"; do
    echo $aluno
```

```
done
```

A saída seria:

```
João Maria Luis Carlos
```

Portanto, a solução mais simples e correta neste caso é usar aspas e o caractere "@", mas também seria possível chegar ao mesmo resultado do código original alterando o caractere delimitador de campos na variável `IFS`:

```
IFS=$'\n'
alunos=(João Maria "Luis Carlos")
for aluno in ${alunos[*]}; do
    echo $aluno
done
```

E a saída seria:

```
João
Maria
Luis Carlos
```

> *Alterar o valor do `IFS` é um procedimento comum, mas deve ser feito com muito critério. Aqui, no nosso curso básico, a minha intenção é apenas demonstrar a expansão de quebra de palavras e mostrar que a variável `IFS` existe, razão pela qual nós não nos aprofundaremos no assunto por enquanto.*

9.6 - Substituição de comandos

Essa expansão já pode ser considerada uma velha conhecida e, portanto, não há muito mais o que falar sobre ela. Então, nós vamos nos limitar a alguns detalhes que ainda não mencionamos.

Novas e antigas sintaxes das substituições de comandos

Antiga sintaxe (ainda funciona, mas é considerada obsoleta):

```
`comando`
```

Nova sintaxe (preferível):

```
$( comando )
```

- Com a **antiga sintaxe**, a barra invertida (`\`) mantém o seu significado literal, a não ser que venha seguida dos caracteres `$`, acento grave, ou `\`.
- Na **nova sintaxe**, todos os caracteres dentro dos parêntesis compõem o comando e nada é tratado de forma especial.

Substituições de comandos podem ser aninhadas

Antiga sintaxe:

```
comando \outro comando
```

Nova sintaxe:

```
$( comando $(outro comando) )
```

Variáveis compartilhadas por sessões diferentes

Apesar das substituições de comandos serem executadas em suas próprias sessões do shell (em *subshells*), elas recebem cópias das variáveis do ambiente de execução da sessão da qual elas derivam...

```
:~$ fruta=banana
:~$ saida=$(echo Macacos comem $fruta)
:~$ echo $saida
Macacos comem banana
```

9.7 - Remoção de aspas

Depois do processamento de qualquer expansão do shell, todas as ocorrências dos

caracteres sem aspas `\'`, `'`, e `"` **que não resultarem da expansão** serão removidos.

Exemplo:

```
:~$ fruta=banana
:~$ echo "$fruta"
fruta
:~$ texto="Minha casa é 'velha', mas nem tanto."
:~$ echo "$texto"
Minha casa é 'velha', mas nem tanto.
```

Aula 10 – Expansões de Parâmetros

10.1 - Trocando o nome pela pessoa

A expansão de parâmetros é o que acontece quando o shell substitui a ocorrência de um nome de variável prefixado com um cifrão (`$`) pelo valor armazenado na variável, como nós temos visto desde a Aula 4:

```
:~$ fruta=banana
:~$ alunos=(João Maria Sofia José)
:~$ echo $fruta
banana
:~$ echo ${aluno[2]}
Sofia
```

Note que o cifrão antes de **substituições de comando** e **expansões aritméticas** também tem, de certo o modo, a mesma função dos casos acima: indicar para o shell que ele deve substituir o que vem em seguida por um valor. A diferença é que este valor, em vez de estar armazenado numa variável, é o resultado de uma expressão ou a saída de um comando.

A rigor, o nome da variável deveria ser escrito entre chaves (`${nome }`), mas isso é opcional no caso das variáveis escalares:

```
:~$ fruta=banana
:~$ echo ${fruta}
banana
```

Na prática, quando se trata de variáveis escalares, as chaves só são utilizadas quando nós queremos separar o nome da variável do restante da string com que ela será concatenada. Por exemplo:

```
:~$ prefixo=im
:~$ echo ${prefixo}possível
impossível
```

Aqui, sem as chaves, o nome da variável a ser expandida ficaria `$prefixopossível`, e não é isso que nós queremos.

As chaves também não são exatamente obrigatórias quando trabalhamos com arrays, mas o valor expandido será apenas o do primeiro elemento (índice "0"), quando forem arrays indexadas, ou o último a ser atribuído, nas arrays associativas.

```
:~$ alunos=(João Maria José)   # Array indexada...
:~$ echo $alunos
João

:~$ declare -A notas    # Array associativa...
:~$ notas[jonas]=8
:~$ notas[joana]=10
:~$ notas[lucas]=9
:~$ echo $notas
9
```

No caso das arrays, as chaves servem para dizer ao shell que o conteúdo dos colchetes deve ser interpretado como o índice do elemento que contém o valor que nós queremos...

```
:~$ echo ${alunos[1]}
Maria
:~$ echo ${notas[joana]}
10
```

10.2 - Indireções

Outra expansão que nós já vimos no curso, só não ligamos o nome à pessoa, foi quando falamos de arrays e explicamos como fazer para listar os índices em vez dos valores de cada elemento. Por exemplo:

```
# Retornando valores...

:~$ alunos=(João Maria José)
:~$ echo ${alunos[@]}
João Maria José
```

```
# Retornando os índices...

:~$ alunos=(João Maria José)
:~$ echo ${!alunos[@]}
0 1 2
```

Mas, quando o caractere exclamação (`!`) aparece antes do nome de uma **variável escalar entre chaves**, ele faz com que o Bash execute a chamada **expansão indireta** (ou *indireção*), que é quando o valor de uma variável corresponde ao nome de outra variável, como no exemplo abaixo:

```
:~$ fruta=banana
:~$ banana=amarela
:~$ echo ${fruta}
banana
:~$ echo ${!fruta}
amarela
```

Se o nome da variável escalar estiver seguido dos caracteres `*` ou `@`, o comportamento da expansão se altera para retornar uma lista de nomes de variáveis que correspondem a um padrão inicial. Por exemplo:

```
# Encontrando variáveis pelo padrão no nome...

:~$ aluno_1=João
:~$ aluno_2=Maria
:~$ aluno_3=José
:~$ echo ${!aluno*}
aluno_1 aluno_2 aluno_3
```

Fica bem claro, portanto, que a expansão de parâmetros é muito mais poderosa do que a simples substituição de um nome por um valor. Dentro das chaves, nós podemos utilizar diversos outros símbolos para processar de muitas formas os valores armazenados em uma variável.

10.3 - Substrings

Não é raro nós precisarmos apenas de um “pedaço” de uma string em vez de todo o

seu conteúdo. Em programação, esse “pedaço” é chamado de **substring**.

Considere a string abaixo:

```
:~$ minha_string=1234567890abcdefghij
```

Se eu quiser apenas tudo que vier **depois** do sétimo caractere, basta colocar dois pontos (`:`) após o nome da variável e o número correspondente à última posição que deve ser descartada (no caso, 7):

```
:~$ echo ${minha_string:7}
890abcdefghij
```

Nós também podemos indicar quantos caracteres queremos ver a partir da posição indicada. Basta incluir novamente os dois pontos (":") seguido da quantidade desejada. Veja os exemplos:

```
:~$ echo ${minha_string:7:0}    # Não retorna nada...

:~$ echo ${minha_string:7:1}
8
:~$ echo ${minha_string:7:2}
89
:~$ echo ${minha_string:7:3}
890
```

Também podemos iniciar a contagem da posição a partir do fim da string...

```
:~$ minha_string=1234567890abcdefghij
:~$ echo ${minha_string: -7}
defghij
```

Repare em duas coisas no exemplo:

- Antes do `-7`, **existe um espaço**;
- A substring retornada **contém** os 7 últimos caracteres da string.

Ou seja, quando contamos a partir do fim, estamos definindo a quantidade de caracteres **que nos interessam**, e não a quantidade de caracteres da parte **que não nos interessa**, como fizemos antes.

Reforçando a ideia:

```
# Descarta os 7 primeiros caracteres e retorna o restante da string...
:~$ echo ${minha_string:7}
890abcdefghij

# Retorna os 7 últimos caracteres...
:~$ echo ${minha_string: -7}
defghij
```

As quantidades de caracteres também podem ser negativas. Por exemplo:

```
:~$ minha_string=1234567890abcdefghij
:~$ echo ${minha_string:7:-2}
890abcdefgh
```

Aqui, o valor foi expandido para todos os caracteres **depois** da posição 7 **menos** os dois últimos caracteres.

Outros exemplos:

```
:~$ minha_string=1234567890abcdefghij
:~$ echo ${minha_string: -7} # Retorna os 7 últimos caracteres
defghij
:~$ echo ${minha_string: -7:2} # Dos 7 últimos caracteres, retorna apenas os
2 primeiros...
de
:~$ echo ${minha_string: -7:-2} # Dos 7 últimos caracteres, descarta os 2
últimos...
defgh
```

Importante! *O espaço antes de uma posição negativa é obrigatório e necessário para diferenciar a indicação de uma contagem negativa (`: -7`) do símbolo de uma outra expansão de parâmetro (`:-`), utilizada para definir o valor padrão de uma variável caso ela não tenha sido definida.*

Por exemplo:

```
:~$ minha_var=teste
```

```
:~$ echo ${minha_var:-banana}
teste
:~$ unset minha_var
:~$ echo ${minha_var:-banana}
banana
```

Mas isso é assunto para outro tópico mais adiante.

10.4 - Comprimento de strings e número de elementos de arrays

Para saber o número de caracteres de uma string, nós utilizamos a cerquilha (`#`) antes do nome da variável:

```
:~$ fruta=banana
:~$ echo ${#fruta}
6
```

Mas, se estivermos trabalhando com uma array, o shell expandirá o número de elementos que ela contém:

```
:~$ echo ${#alunos[@]}
3
```

*A título de exercício, você sabe explicar por que o código abaixo retorna `4`? Uma pista: pense no que está **realmente** sendo retornado pela expansão abaixo.*

```
:~$ alunos=(João Maria "Luis Carlos")
:~$ echo ${#alunos}
4
```

10.5 - Testando variáveis

Através das expansões de parâmetros, nós podemos testar se uma variável foi ou não definida e dizer ao shell o que fazer em cada caso. É aí que entram os símbolos abaixo:

Expansão	Descrição
`${nome:-valor_padrao}`	Caso `nome` não exista ou seja nulo, o shell expande `valor_padrao`.
`${nome:=valor_padrao}`	Caso `nome` não exista ou seja nulo, o shell expande `valor_padrao` e atribui `valor_padrao` a `nome`.
`${nome:?mensagem}`	Caso `nome` não exista ou seja nulo, o shell escreve `mensagem` na saída de erro e termina a execução se estiver no modo não-interativo.
`${nome:+valor_padrao}`	Caso `nome` não exista ou seja nulo, nada é expandido. Se `nome` existir, `valor padrão` é expandido.

Essas quatro expansões cobrem muitos problemas que nós resolveríamos com o comando `test` ou com uma estrutura de decisão `if`.

Exemplo 1: a variável `minha_var` não existe...

```
:~$ echo ${minha_var:-Este é o padrão}
Este é o padrão
:~$ echo $minha_var

:~$
```

Exemplo 2: a variável `minha_var` *não existia*...

```
:~$ echo ${minha_var:=Este é o padrão}
Este é o padrão
:~$ echo $minha_var
Este é o padrão
```

Exemplo 3: gera um erro se a variável `minha_var` não existir...

```
:~$ echo ${minha_var:?"A variável 'minha_var' não existe!"}
bash: minha_var: A variável 'minha_var' não existe!
```

Exemplo 4: a variável `minha_var` existe!

```
:~$ minha_var=teste
:~$ echo ${minha_var:+Este é o padrão}
Este é o padrão
:~$ echo $minha_var
teste
```

10.6 – Maiúsculas e minúsculas

É muito fácil alterar a caixa (alta ou baixa) dos caracteres de uma string com as expansões de parâmetros.

Expansão	Descrição
`${variavel^}`	Primeiro caractere em caixa alta.
`${variavel^^}`	Todos os caracteres em caixa alta.
`${variavel,}`	Primeiro caractere em caixa baixa.
`${variavel,,}`	Todos os caracteres em caixa baixa.

Exemplos:

```
# Caixa alta...

:~$ fruta=banana
:~$ echo ${fruta^}
Banana
:~$ echo ${fruta^^}*
BANANA

# Caixa baixa...

:~$ fruta=LARANJA
:~$ echo ${fruta,}
lARANJA
:~$ echo ${fruta,,}
laranja
```

10.7 – Aparando strings

Além das substrings, nós também podemos aparar (*trim*) o início ou o fim de uma string a partir de padrões de correspondência. Isso é possível com o uso dos símbolos `#` ou `%`.

Expansão	**Descrição**
`${variavel#padrão}`	Todo o início da string será aparado até a **primeira** ocorrência do padrão.
`${variavel##padrão}`	Todo o início da string será aparado até a **última**ocorrência do padrão.
`${variavel%padrão}`	Todo o fim da string será aparado até a **primeira** ocorrência do padrão de trás para frente.
`${variavel%%padrão}`	Todo o fim da string será aparado até a **última** ocorrência do padrão de trás para frente.

Nos exemplos abaixo, observe que o padrão é `*ba`, significando: "caracteres `b` e `a` antecedidos de qualquer ou nenhum caractere". Observe também que a string em `frutas` possui duas ocorrências da sequência `ba`.

Portanto...

```
# O padrão casa com tudo até a primeira ocorrência de 'ba'...

:~$ frutas='banana laranja abacate'
:~$ echo ${frutas#*ba}
nana laranja abacate

# O padrão casa com tudo até a última ocorrência de 'ba'...

:~$ frutas='banana laranja abacate'
:~$ echo ${frutas##*ba}
cate
```

Sem o metacarectere `*`, porém, nós teríamos resultados iguais...

```
:~$ frutas='banana laranja abacate'
:~$ echo ${frutas#ba}
nana laranja abacate
:~$ echo ${frutas##ba}
nana laranja abacate
```

Isso porque, na segunda expansão, `ba` só casa com os dois primeiros caracteres da string. Por isso nós utilizamos os meta caracteres para representar os padrões. Dentre eles, os mais importantes no momento são:

Meta	Descrição
`*`	Casa com qualquer quantidade de caracteres ou nenhum caractere.
`?`	Casa com um único caractere qualquer existente.
`[...]`	Casa com um único caractere existente do listado entre os colchetes.

Nos exemplos abaixo, nós veremos como aparar o final de uma string a partir de uma busca reversa pelo padrão:

```
# O padrão casa com a mínima correspondência reversa iniciada com 'an'...

:~$ frutas="banana laranja abacate"
:~$ echo ${frutas%an*}
banana lar

# O padrão casa com a máxima correspondência reversa iniciada com 'an'...

:~$ frutas="banana laranja abacate"
:~$ echo ${frutas%%an*}
b
```

10.8 - Busca e substituição de padrões

Existem ótimas ferramentas disponíveis na linha de comandos para realizar buscas e substituições de padrões de strings, como os utilitários `sed`, `awk` e `tr`, por exemplo. Mas o Bash oferece uma solução bem mais rápida para esse tipo de operação com as suas expansões.

Expansão	**Descrição**
`${variavel/padrão/string}`	Substitui a **primeira** ocorrência de 'padrão' por 'string'.
`${variavel//padrão/string}`	Substitui **todas** as ocorrências de 'padrão' por 'string'.

Por exemplo:

```
:~$ mensagem="Use Windows e seja feliz!"

# Casa com 'Windows'

:~$ echo ${mensagem/Windows/Linux}
Use Linux e seja feliz!

# Casa com 'W', qualquer coisa, e o 's' seguido de espaço...

:~$ echo ${mensagem/W*s /Linux }
Use Linux e seja feliz!

# Casa com 'W' seguido de qualquer coisa até o último espaço...

:~$ echo ${mensagem/W* /Linux }
Use Linux feliz!

# Substituindo todas as ocorrências do padrão 'te' pela string 'tche'...

:~$ compras="leite mate chocolate"
:~$ echo ${compras//te/tche}
 leitche matche chocolatche
```

Aula 11 – O loop 'for'

11.1 - Comandos compostos

O loop `for` faz parte de uma categoria de comandos chamada de **comandos compostos**, que são estruturas de linguagem que agrupam comandos ou são compostas por um blocos de controle e execução de comandos:

- Agrupamento de comandos com parêntesis
- Agrupamento de comandos com chaves
- Estruturas de repetição (`for`, `while`, `until`)
- O menu `select`
- Estruturas de decisão (`if` e `case`)

Estando no grupo das estruturas de repetição, a função do `for` é percorrer sequencialmente cada elemento de uma lista executando comandos. Durante cada iteração (cada ciclo), o loop `for` armazena o valor da lista que está sendo lido no momento em uma variável.

11.2 - Sintaxe

A sintaxe geral do loop `for` é:

```
for NOME [in LISTA]; do
    COMANDOS
done
```

Onde `LISTA` pode ser uma expansão de uma string com elementos separados por espaços, arrays, expansões de nomes de arquivos ou expansões de chaves. Caso a expressão `in LISTA` seja omitida, o Bash presumirá a sintaxe abaixo:

```
for NOME in "$@"; do
    COMANDOS
done
```

Onde `$@` são todos os parâmetros posicionais passados na linha de comando (`$1`, `$2`, `$3`...).

Alternativamente, o loop `for` pode trabalhar avaliando expressões, como na linguagem C:

```
for (( expressão1; expressão2; expressão3 )); do
    COMANDOS
done
```

Onde...

- **expressão1** é uma atribuição de um valor inicial;
- **expressão2** é uma expressão condicional que espera uma avaliação falsa para indicar o fim do loop;
- **expressão3** é uma atualização do valor inicial.

Por exemplo:

```
for (( n = 1; n <= 5; n++ )); do
    echo $n
done
```

Neste caso, o valor inicial de `n` é `1` e, a cada ciclo, ele será incrementado em `+1` até a avaliação expressão condicional resultar em **falso**. Deste modo, o comando `echo` será executado enquanto o valor de `n` for menor ou igual a `5`, resultando em:

```
1
2
3
4
5
```

11.3 - Percorrendo as palavras em uma string

Qualquer string com elementos separados por espaços pode ser passada para o loop `for` como uma `LISTA`.

Por exemplo:

```
:~$ alunos='João Maria Luis Carlos'
:~$ for nome in $alunos; do echo $nome; done
João
Maria
Luis
Carlos
```

Isso teria o mesmo efeito de...

```
:~$ for nome in João Maria Luis Carlos; do echo $nome; done
João
Maria
Luis
Carlos
```

Para utilizar outros separadores que não sejam espaços, é necessário alterar temporariamente o valor na variável `IFS` (o separador interno de campos). Por exemplo, se quisermos passar uma string separada por vírgulas como `LISTA`:

```
alunos='João,Maria,Luis Carlos'

# Salvando o IFS para restauração...
IFS_ORIGINAL=$IFS

# Alterando o IFS...
IFS=','

# Executando o loop...
for nome in $alunos; do echo $nome; done

# Restaurando o IFS original...
IFS=$IFS_ORIGINAL
```

A saída seria:

```
João
Maria
Luis Carlos
```

> *Nos scripts, a menos que existam outros blocos de código dependendo dos valores originais, não é necessário salvar e restaurar o valor de IFS, já que qualquer alteração dos separadores originais ficará restrita apenas à sessão de execução do script.*

Quando a string possui quebras de linha (caractere de controle `\n`) como separadores de campos, o IFS pode ser alterado utilizando uma expansão de aspas de que nós ainda não falamos, e que expande o significado de caracteres ANSI-C em uma string...

```
$'...' --> Expande caracteres no padrão ANSI-C
```

Esses caracteres são sequências iniciadas com a barra invertida e representam, em boa parte, os chamados caracteres de controle, como a quebra de linha (`\n`), a tabulação horizontal (`\t`), e outros caracteres escapados.

Veja o exemplo:

```
:~$ alunos=$'João\nMaria\nLuis Carlos'
```

Neste caso, as quebras de linha seriam expandidas pelo shell, o que pode ser verificado fazendo...

```
:~$ echo "$alunos"
João
Maria
Luis Carlos

# Sem aspas, não seria possível perceber a diferença...
:~$ echo $alunos
João Maria Luis Carlos

# Obviamente, com aspas simples não haveria a expansão do '$'...
:~$ echo '$alunos'
$alunos
```

Ainda neste mesmo caso, a variável `IFS` deveria ser alterada para `\n` antes de podermos utilizar `alunos` como a `LISTA` de um loop `for`...

```
:~$ IFS=$'\n' # Alterando o IFS...
```

```
:~$ for nome in "$alunos"; do echo "$nome"; done
João
Maria
Luis Carlos
```

11.4 - Percorrendo elementos de uma array

Se `LISTA` for uma array, o loop `for` percorrerá cada um de seus elementos armazenando seus valores ou seus índices na variável temporária.

Por exemplo:

```
:~$ frutas=("banana" "laranja" "mamão papaya")

# Armazenando os valores em ‘fruta’...

:~$ for fruta in "${frutas[@]}"; do echo $fruta; done
banana
laranja
mamão papaya

# Armazenando os índices em ‘fruta’...

:~$ for fruta in "${!frutas[@]}"; do echo $fruta; done
0
1
2
```

Também podemos percorrer os elementos de uma array associativa...

```
:~$ declare -A carros
:~$ carros[vw]="Fusca"
:~$ carros[fiat]="Palio"
:~$ carros[ford]="Corcel"

# Armazenando os valores em ‘carro’...

:~$ for carro in "${carros[@]}"; do echo $carro; done
Fusca
Palio
```

```
Corcel

# Armazenando os índices em 'carro'...

:~$ for carro in "${!carros[@]}"; do echo $carro; done
vw
fiat
ford
```

11.5 - Percorrendo nomes de arquivos

As **expansões de nomes de arquivos** também podem ser usadas como uma `LISTA` para o loop `for`:

```
:~$ for arquivo in *.txt; echo $arquivo; done
teste1.txt
teste2.txt
teste3.txt
```

11.6 - Percorrendo faixas numéricas e alfabéticas

As listas podem ser geradas a partir de **expansões de chaves** que resultem em faixas numéricas ou alfabéticas, como nos exemplos abaixo:

```
:~$ for n in {1..5}; do echo $n; done
1
2
3
4
5

:~$ for s in {a..e}; do echo $s; done
a
b
c
d
e
```

```
# Incrementando em 2 passos...

:~$ for n in {1..5..2}; do echo $n; done
1
3
5

# Incrementando em 3 passos...

:~$ for s in {a..j..3}; do echo $s; done
a
d
g
j
```

Como podemos ver, graças à expansão de chaves, não precisamos trabalhar com expressões aritméticas quando o que queremos é, por exemplo, fazer com que o loop `for` dure um determinado número de ciclos.

11.7 - Controlando a execução do loop 'for'

Se quisermos, a partir do resultado de um teste lógico, por exemplo, interromper a execução de um loop, basta utilizar o comando interno `break`:

```
for n in {1..10}; do
    [[ $n -gt 5 ]] && break || echo $n
done
```

Neste exemplo, o comando `break` é invocado caso o valor de `n` seja maior do que `5`.

A saída seria:

```
1
2
3
4
5
```

Na situação inversa, quando queremos que o loop continue, mas sem executar os demais comandos do bloco, nós podemos utilizar o comando interno `continue`:

```
for n in {1..10}; do
    [[ $n -lt 6 ]] && continue
    echo $n
done
```

Aqui, o comando `continue` será invocado em todos os ciclos onde `n` é menor do que `6`, o que faz com que o loop continue sem a execução do comando `echo` enquanto `n` não for maior ou igual a `6`, resultando em...

```
6
7
8
9
10
```

Aula 12 – Loops 'while' e 'until'

12.1 - Estruturas de repetição condicional

Na aula sobre o loop `for`, nós vimos uma estrutura que executava um bloco de comandos à medida em que percorria sequencialmente cada elemento de uma lista. Mas existem outros dois tipos de loop no Bash que, em vez de percorrer valores, irão testar determinadas condições para decidir se continuam executando um bloco de comandos ou não. São os loops `while` e `until`.

As palavras *while* e *until* significam "enquanto" e "até", respectivamente. Portanto, fica fácil deduzir que os blocos de comandos desses loops continuarão sendo executados "enquanto" uma condição for verdadeira (`while`) ou "até" uma condição tornar-se verdadeira (`until`).

12.2 - Sintaxe

Toda sintaxe aplicável ao loop `while` também é aplicável ao loop `until`, o que muda é apenas a forma como cada um deles reage às condições.

```
# Lopp while...

while CONDIÇÃO; do
    COMANDOS
done

while CONDIÇÃO; do COMANDOS; done

# Loop until...

until CONDIÇÃO; do
    COMANDOS
done

until CONDIÇÃO; do COMANDOS; done
```

> ***Importante!*** *A `CONDIÇÃO` não precisa ser necessariamente um comando `test`. Na verdade, qualquer expressão que retorne um estado `0` (sucesso/verdadeiro) ou `1` (erro/falso) pode ser utilizada, inclusive os comandos internos `true`, `false` e o comando nulo `:`.*

12.3 - O loop 'while'

Como dissemos, o loop `while` executa repetidamente um bloco de comandos **enquanto** uma determinada condição for verdadeira:

```
n=0
while [[ $n -lt 5 ]]; do
    echo $n
    ((n++))
done
```

Neste caso, nós definimos o valor inicial de `n` fora do bloco de comandos. A cada ciclo nós verificamos se o seu valor ainda é menor do que 5, incrementando o valor original em `+1` em seguida. Quando o valor de `n`, depois de ser incrementado, passar a ser igual a 5, o teste sairá com status `1` (falso), encerrando o loop. Sendo assim, cada um dos comandos no bloco é executado 5 vezes, resultando nesta saída no terminal:

```
0
1
2
3
4
```

> ***Importante!*** *Os comandos sempre serão executados durante os ciclos, e não ao final de todos os ciclos. O que delimita o bloco de comandos é a estrutura `do COMANDOS; done`.*

12.4 - O loop 'until'

O loop `until` segue a lógica inversa do loop `while`. Para que um ciclo seja executado, o teste tem que sair com status `1` (falso), e o loop será interrompido assim que o teste sair com status `0` (verdadeiro).

Por exemplo, para obter o mesmo resultado do exemplo anterior...

```
n=0
until [[ $n -eq 5 ]]; do
    echo $n
    ((n++))
done
```

Repare que, já no início, o teste sairia com status `1` (falso), e esta é a condição lógica que o loop `until` espera para liberar a execução do bloco de comandos. À medida em que o valor de `n` é incrementado, eventualmente ele será igual a `5`, fazendo com que o teste retorne status `0` (verdadeiro) e os clicos sejam interrompidos.

A saída no terminal seria a mesma:

```
0
1
2
3
4
```

12.5 - Loops infinitos

Muitas vezes, existe a necessidade de executar loops indefinidamente, deixando por conta de alguma interferência do usuário ou de um teste no bloco de comandos a sua interrupção. Nesses casos, em vez de uma expressão a ser avaliada, nós utilizamos comandos internos que retornam imediatamente um estado de saída:

Comando	**Descrição**
`true`	Retorna sempre o status `0` (sucesso)
`false`	Retorna sempre o status `1` (erro)

`:`	Comando nulo, não faz nada e sempre retorna `0` (sucesso)

A interrupção de um loop infinito pode ser feita através de alguma intervenção do usuário, seja através do comando **Ctrl+C** no terminal, da espera pela entrada de um valor de um comando `read`, ou através de instruções que resultem na chamada dos comandos `break` ou `exit`.

Exemplo 1: loop `while` infinito...

```
while true; do
    COMANDOS
done
```

Ou ainda...

```
while : ; do
    COMANDOS
done
```

Exemplo 2: loop `until` infinito...

```
until false; do
    COMANDOS
done
```

12.6 - Interrompendo loops infinitos

Não existe uma fórmula definitiva para interromper um loop infinito pelo código, mas nós podemos entender o conceito geral analisando alguns exemplos. Aqui, nós utilizaremos apenas o loop `while`, mas, como se trata de um loop infinito, pouco importa se vamos utilizar a lógica `until false` ou `while true`, pois é o que acontece no bloco de comandos que nos interessa.

Exemplo1: utilizando o comando `break`...

```
n=1
while true; do
```

```
  [[ $n -lt 5 ]] && echo "Ciclo $n" || break
  ((n++))
done
```

Como no primeiro exemplo, nós atribuímos o valor `5` à variável `n`. Só que, agora, o teste é feito no bloco de comandos, e não na declaração do `while`. Se o valor de `n` for maior do que `5`, o teste retornará erro (`1`) e o comando `break` será executado, interrompendo a execução do loop.

Exemplo 2: controlando o loop com o comando `read`...

```
while true; do
  read -p "Digite qualquer coisa ou Q para sair: " str
  [[ "$str" = [qQ] ]] && echo Saindo... && break \
                      || echo "Você digitou $str"
done
```

Desta vez, o comando `read` irá pausar o ciclo até o usuário entrar com algum texto. O texto digitado será armazenado na variável `str` e testado para ver se corresponde ao padrão `q` ou `Q`. Se isso acontecer, o loop será interrompido. Este é um método muito utilizado na construção de menus, assunto da nossa próxima aula.

Aula 13 – O menu 'select'

13.1 - Um menu simplificado

O quarto tipo de loop que o Bash nos oferece combina a capacidade de percorrer os valores de uma lista alimentando uma variável (como no loop `for`) com os loops infinitos que fizemos com o `while` e o `until` – trata-se do menu `select`.

O `select` é um loop especialmente criado para gerar menus. Na verdade, ele é tão especializado nisso, que tem até o seu próprio prompt no shell: o prompt `PS3`.

13.2 - Sintaxe

Sua sintaxe geral é idêntica à do loop `for`:

```
select NOME [in LISTA]; do
    COMANDOS
done
```

Ou, numa linha...

```
select NOME [in LISTA]; do COMANDOS; done
```

Cada elemento da `LISTA` será utilizado para a exibição de um menu numerado seguido de um prompt que espera pela opção do usuário.

Por exemplo:

```
:~$ select fruta in banana laranja abacate; do echo $fruta && break; done
1) banana
2) laranja
3) abacate
#? _
```

O valor padrão do prompt `PS3` é `#?`. Quando o usuário digita o número correspondente à sua opção, o valor da string correspondente ao número escolhido é

atribuído à variável `NOME` (`fruta`, no exemplo), o bloco de comandos é executado e um novo ciclo é iniciado, a menos que um dos comandos do bloco encerre o loop (com um `break`) ou a sessão do shell (com um `exit`).

> *Se não houver nenhum valor no prompt quando o usuário teclar **Enter**, a lista de opções e o prompt serão exibidos novamente. Sem um comando de saída, cada escolha feita levará a uma nova exibição do prompt.*

13.3 - O prompt 'PS3'

O Bash trabalha com cinco prompts diferentes, cada um deles armazenado em uma variável de ambiente:

Variável	Descrição
`PS0`	Contém um string que é expandida após a execução de um comando e exibida antes de qualquer saída do comando.
`PS1`	É o prompt de comandos do terminal, por exemplo... `:~$`
`PS2`	Prompt de continuação, exibido no terminal quando o nosso comando está incompleto. Geralmente, seu valor é `>`.
`PS3`	É o prompt do loop `select` e seu valor padrão é `#?`.
`PS4`	Define o prefixo das saídas que estão sendo rastreadas em um script (por exemplo, quando utilizamos o parâmetro de execução `set -x` no script). Seu valor padrão geralmente é `+`.

No caso, a variável que nos interessa é a `PS3`, que pode ser definida antes da execução do loop `select`. Observe no exemplo:

```
PS3="Escolha um número ou tecle '4' para sair: "
select fruta in banana laranja abacate sair; do
    [[ $fruta = "sair" ]] && break
    echo "Você escolheu $fruta."
    break
done
```

Executando o script, esta seria a saída no terminal:

```
1) banana
2) laranja
3) abacate
4) sair
Escolha um número ou tecle '4' para sair:
```

Repare também que nós definimos dois pontos de saída para o loop: o primeiro, caso o usuário escolha `4`, atribuindo `sair` ao valor de `fruta`, e o segundo após a execução dos demais comandos do bloco.

> ***Importante!*** *A variável `NOME` recebe o valor da string associada ao número da lista, e não o número que foi digitado! Para obter o número digitado, nós utilizamos uma variável que o shell sempre gera após um comando `read` ou o menu `select`: a variável `REPLY`.*

13.4 - Menus com 'while' e 'until'

Mesmo tendo aplicações muito mais genéricas, os loops `while` e `until` são bastante utilizados na criação de menus. Na verdade, os menus com o loop `select`, embora práticos em algumas situações, podem ser um pouco limitados, especialmente quando precisamos de um maior grau de comunicação com o usuário.

De forma geral, todo menu com `while` ou `until` seguirá essa estrutura:

```
fruta=(banana laranja abacate)

while true; do
   clear
   echo "1. Banana"
   echo "2. Laranja"
   echo "3. Abacate"
   echo "4. Sair"
   read -p "Escolha o número da sua opção: " opt
   [[ $opt -eq 4 ]] && break
   echo "Você escolheu ${fruta[$(($opt - 1))]}."
   read -p "Tecle algo para continuar..." continua
```

```
done
```

Como o comando `read` já nos oferece um prompt, o `echo` das opções também poderia ser dispensado:

```
fruta=(banana laranja abacate)

while true; do
   clear
   read -p "
1. Banana
2. Laranja
3. Abacate
4. Sair

Escolha o número da sua opção: " opt
   [[ $opt -eq 4 ]] && break
   echo "Você escolheu ${fruta[$(($opt - 1))]}."
   read -p "Tecle algo para continuar..." continua
done
```

Evidentemente, seria preciso tratar a entrada do usuário para garantir que ele só digitou números de 1 a 4, mas esse é o esquema geral de um menu. Basicamente, nós sempre teremos uma lista de opções, um prompt para a entrada da escolha, e um bloco de comandos onde a escolha do usuário será tratada e direcionada de alguma forma para uma ação correspondente.

É neste ponto que entram as estruturas de decisão do Bash, como as declarações `case` e `if`, por exemplo, que serão vistas na próxima aula.

Aula 14 – Estruturas de decisão ‘if’ e ‘case’

14.1 - A estrutura 'if', 'elif', 'then', 'else'

O comando composto `if`, através de seus componentes, as palavras-chave `if` e `elif`, observa o status de saída de um comando. Se o comando observado terminar com status de saída `0` (sucesso), um bloco de comandos iniciado pela palavra-chave `then` será executado. Caso contrário, se houver um bloco de comandos opcional iniciado pela palavra-chave `else`, este é que será executado. Se não houver um bloco `else`, nem outros comandos a serem observados, o comando composto `if` é terminado.

A sintaxe geral é a seguinte:

```
if COMANDO 1
then
    BLOCO DE COMANDOS 1
elif COMANDO 2
then
    BLOCO DE COMANDOS 2
...
elif COMANDO N
then
    BLOCO DE COMANDOS N
else
    BLOCO DE COMANDOS ALTERNATIVOS
fi
```

Nessa estrutura, é muito importante observar que:

- Ela sempre terá, no mínimo, a palavra chave `if`, um comando observado, o bloco de comandos iniciado pelo `then` e o terminador da estrutura `fi`;
- Só pode haver uma linha `if` e ela sempre será a primeira linha da estrutura;
- Todas as linhas `if` e `elif` terão que ser seguidas pelos seus respectivos blocos `then`;
- Os blocos `then` só são executados se os comandos observados nas linhas `if` ou `elif` que os antecederem terminarem com sucesso;

- Só pode haver um bloco de comandos alternativos iniciado pelo `else`;
- O bloco `else`só será executado se nenhuma das linhas `if` ou `elif` observar um comando que termine com sucesso;
- O bloco `else` não pode vir sem que um bloco `then` o anteceda;
- O bloco `then` só é executado se as linhas `if` e `elif` encontrarem sucesso;
- O bloco `else` só é executado se as linhas `if` e `elif` encontrarem erro.

Além disso, esteja atento ao fato de que uma linha de comando iniciada com um `if` ou um `elif` é uma linha de comando como todas as outras, no sentido de que o comando observado será executado normalmente, como se não houvesse nada antes dele. As diferenças só vão acontecer depois da execução do comando observado, que é quando o shell decide se irá ou não executar o bloco de comandos seguinte (iniciado com a palavra-chave `then`).

Então, fique atento:

- O `if` **não testa** afirmações lógicas;
- O `if` **não testa** comandos;
- O `if` testa **saídas** de comandos!

14.2 - Um engano muito comum

Mas, e a estrutura abaixo?

```
if [[ expressão ]]
then
    COMANDOS SE SUCESSO
else
    COMANDOS SE ERRO
fi
```

Para quem não conhece o shell, parece que o teste de expressões faz parte da estrutura do `if`, mas isso não é verdade. Os testes de expressões `[[...]]` e `[...]`, assim como a expressão aritmética `((...))`, como vimos neste curso, também são comandos. Ou seja, eles também podem ter suas saídas testadas pelo `if`, mas não fazem parte da estrutura.

Aliás, de longe, estes são os comandos mais testados com o `if` e o `elif`, e qualquer

busca em tutoriais e exemplos na internet pode comprovar isso. Mas, o maior poder do `if` no Bash é a sua capacidade de testar a saída de qualquer comando.

Por exemplo:

```
if grep ^$USER /etc/passwd &>/dev/null; then
    echo "Usuário encontrado"
else
    echo "Usuário não encontrado"
fi
```

Aqui, nós testamos o status de saída do comando `grep`, que fez a busca do nome do usuário logado no sistema (`$USER`) no arquivo `/etc/passwd`, que é um arquivo que armazena dados dos usuários do sistema. Normalmente, o `grep` retornaria a linha encontrada no arquivo e encerraria com status `0`. Para evitar a exibição da linha do arquivo, nós redirecionamos todas as saídas (`&>`) para o "limbo" do sistema, o arquivo `/dev/null`.

14.3 - Operadores de encadeamento condicional

Também conhecidos (erroneamente) como "conectores lógicos" ou (mais corretamente) "conectores condicionais", os operadores `&&` e `||` funcionam mais ou menos como o comando composto `if`, pelo menos no que diz respeito ao fato de que ambas as construções testam saídas de comandos, mas existem diferenças importantes!

Observe este exemplo:

```
:~$ true && echo verdadeiro; false || echo falso
```

Consegue adivinhar a saída?

Isso mesmo!

```
verdadeiro
falso
```

O problema é que os conectores condicionais sempre avaliam a saída do último

comando executado antes deles, ou seja, não existe a ideia de um "bloco de comandos", eles só fazem a ligação entre um comando e o comando seguinte, daí o nome: *"operadores de encadeamento"*.

Com o comando composto `if`, a coisa é diferente nesse aspecto:

```
:~$ if true; then echo verdadeiro; false; else echo falso; fi
verdadeiro
```

Enquanto não aparecer um `elif`, um `else` ou um `fi`, tudo que vier depois do `then` será executado e, em seguida, a estrutura toda será encerrada.

14.4 – A estrutura 'case'

O comando composto `case` executa seletivamente um bloco de comandos de acordo com o casamento de um padrão com o valor de uma palavra.

A sintaxe geral é a seguinte:

```
case PALAVRA in
   PADRÃO 1) COMANDOS;;
   PADRÃO 2) COMANDOS;;
   ...
esac
```

Aqui, o `PADRÃO` pode ser qualquer string literal, uma lista de strings separadas por uma barra vertical (`|`), que pode ser lida como "ou", ou composto pelos mesmos símbolos utilizados para representar padrões de nomes de arquivos: `[ ... ]`, `?`, e `*`.

Pode haver qualquer número de blocos de padrões e comandos numa declaração `case`, mas apenas o bloco que corresponder ao padrão buscado será executado se a lista de comandos correspondente a um padrão terminar com um par de ponto e vírgulas (`;;`).

> *Existem outros delimitadores de blocos de comandos além do `;;`, mas este basta para um primeiro contato.*

Também é muito comum utilizar o asterisco (`*`), que casa com qualquer coisa, no

último bloco de comandos para definir uma ação padrão, caso não exista correspondência com nenhum dos padrões anteriores.

Aqui está um exemplo típico de uso:

```
read -p "Digite o nome de uma fruta: " fruta

case ${fruta,,} in
   banana|laranja|maracuj[aá]) echo "É uma fruta amarela"  ;;
      abacate|melancia|limão*) echo "É uma fruta verde"    ;;
              morango|pitanga) echo "É uma fruta vermelha" ;;
                           *) echo "Eu não sei a cor dessa fruta..."
                              exit 1
                              ;;
esac

exit 0
```

No geral, como podemos ver, a ideia do comando é bem simples, mas alguns detalhes precisam ser observados.

1 - Tratamento da string que está sendo passada para o comando 'case'

```
case ${fruta,,} in
```

Repare que aqui nós utilizamos uma expansão de parâmetros para deixar todo o valor da variável `fruta` em minúsculas. Isso facilita o casamento dos padrões, já que não temos que nos preocupar com maiúsculas e minúsculas quando o usuário digitar os dados solicitados.

2 – Criando padrões que contemplam várias possibilidades de digitação

```
banana|laranja|maracuj[aá]) echo "É uma fruta amarela";;
```

Como o usuário poderia ter digitado "maracujá" sem o acento, nós utilizamos uma lista (`[...]`) para informar os caracteres válidos na última posição da string. No caso, "a" ou "á". Observe também que, no total, nós "autorizamos" a execução do bloco de comandos com o casamento de 4 strings possíveis: "banana", "laranja", "maracuja" ou "maracujá".

3 - Definindo uma opção padrão

```
        *) echo "Eu não sei a cor dessa fruta..."
           exit 1
           ;;
```

O caractere coringa `*` casa com qualquer string, inclusive com strings vazias, que seria o caso do usuário teclar **Enter** sem digitar nada quando solicitado. Portanto, esta seria a condição padrão, caso não fosse encontrada nenhuma correspondência com as opções anteriores.

Repare também que, neste último bloco de comandos, nós incluímos o comando `exit 1`. Isso não é obrigatório, mas serve para ilustrar que é possível fazer o script retornar um status diferente do status do último comando executado, que é o retorno padrão do comando composto `case`. Comparando com os demais blocos de comando, neste último, nós também aproveitamos para mostrar que os comandos podem ser dispostos em várias linhas.

Aula 15 – Funções

15.1 - Conceito

Você pode imaginar uma função como um pequeno script dentro do seu script. Elas são pequenos blocos de comandos que recebem um nome pelo qual podem ser chamados. Com as funções, nós evitamos escrever os mesmos comandos e instruções várias vezes no script. Além disso, elas podem tornar o código mais legível, já que o nome da função (se bem escolhido) dá uma dica da finalidade daquele bloco de comandos.

15.2 - Sintaxe geral

No Bash, esses blocos de comandos podem ser qualquer comando composto (qualquer um mesmo!), mas o mais comum é utilizarmo o agrupamento de comandos com chaves.

```
nome_da_função() {
    COMANDOS
}
```

Ou, em uma linha...

```
nome_da_função() { COMANDOS; }
```

> ***Importante!*** *Como delimitadores de um agrupamento, as chaves são tratadas como palavras-chaves pelo shell, o que nos obriga a seperá-las de outras palavras da linha de comando. Além disso, no caso da escrita em uma linha, o último comando do agrupamento precisa terminar com o ponto e vírgula*

Também podemos criar funções utilizando a palavra reservada `function`...

```
function nome_da_função {
```

```
    COMANDOS
}

function nome_da_função { COMANDOS; }
```

A sintaxe é simples, mas alguns detalhes devem ser observados:

- Criar uma função não faz com que ela seja executada. Para isso, é necessário invocá-la (nós dizemos "chamar") a partir de seu nome em algum ponto do script ou pelo terminal.
- As funções devem aparecer no script `antes` dos pontos em que são chamadas, a menos que sejam chamadas por outras funções.
- As regras para os nomes das funções são as mesmas dos nomes das variáveis e nós devemos sempre escolher nomes que descrevam bem a finalidade da função.
- Em outras linguagens, nós podemos declarar argumentos dentro dos parêntesis que vêm depois do nome da função. No Bash, isso não é possível, e eles servem apenas para indicar que estamos definindo uma função quando não usamos a palavra reservada `function`.

15.3 - Nossa primeira função

Vamos criar um script chamado `exemplo.sh`:

```
#!/usr/bin/env bash

oi() {
   echo "Oi, eu sou uma função!"
}

oi
```

Aqui, nós criamos uma função chamada `oi`. Entre as chaves, nós definimos que ela executará apenas o comando `echo`, mas poderia ser qualquer quantidade de comandos. Depois de definida, mais adiante no código, nós podemos chamar a função `oi` quantas vezes quisermos simplesmente escrevendo seu nome nos pontos

em que ela precisar ser chamada.

Executando o script...

```
:~$ ./exemplo.sh
Oi, eu sou uma função!
```

15.4 - Passando "argumentos"

Se a nossa função precisar processar informações, será necessário passar para ela os dados que deverão ser processados, e isso é feito através de parâmetros posicionais. Como já dissemos, nós podemos imaginar as funções como pequenos scripts dentro do nosso script, e essa analogia também engloba a forma como os scripts recebem os parâmetros posicionais.

Então, vamos relembrar: como você passa os parâmetros para um script?

```
:~$ ./script.sh param1 param2 param3 ... paramN
```

Internamente, cada um desses parâmetros seria automaticamente passado para as variáveis especiais `$1`, `$2`, `$3` ... `$N` respectivamente.

No caso das funções, acontece exatamente a mesma coisa.

Veja o exemplo:

```
#!/usr/bin/env bash

oi() {
echo "Oi, eu sou $1!"
}

oi Goku
oi Vegeta
oi "a Bulma"
```

Executando...

```
:~$ ./exemplo.sh
Oi, eu sou Goku!
```

```
Oi, eu sou Vegeta!
Oi, eu sou a Bulma!
:~$
```

Também como nos scripts...

- `$#` - Armazena o número de argumentos passados para a função
- `$*` e `$@`- Armazenam todos os argumentos passados para a função

> *Quando uma função é executada, os parâmetros posicionais (menos o `$0`) e os parâmetros `$#`, `$*` e `$@` são temporariamente redefinidos para receberem os argumentos passados para ela. Quando sua execução termina, esses parâmetros são restaurados para os valores originais da sessão do shell.*

15.5 - Retornando valores

A maioria das linguagens de programação trabalha com o conceito de funções que retornam um valor. As funções do Bash não permitem isso, mas nós podemos fazer com que as nossas funções retornem um status de saída (limitado ao inteiros de `0` a `255`) através do comando interno `return`, o qual pode ser lido posteriormente utilizando a variável especial `$?`.

O comando `return` é análogo ao comando `exit`. A única diferença é que o `return` encerra apenas a execução do bloco de comandos da função, enquanto o `exit` encerra a execução de todo o script.

Voltando ao nosso exemplo...

```
#!/usr/bin/env bash

oi() {
}
    echo "Oi, eu sou $1!"
    [[ -n $1 ]] && return 0 || return 1

oi Goku
echo "Status: $?"
oi
```

```
echo "Status: $?"
```

Executando...

```
:~$ ./exemplo.sh
Oi, eu sou Goku!
Status: 0
Oi, eu sou
Status: 1
:~$
```

Porém, a forma mais correta de obter valores a partir da execução de uma função é através das nossas velhas amigas **substituições de comandos**.

Veja o exemplo:

```
#!/usr/bin/env bash

oi() {
   echo "Oi, eu sou $1!"
}
mensagem=$(oi $1)

echo $mensagem
```

Repare que a variável `$1` aparece duas vezes: dentro da função, recebendo o valor do argumento passado na chamada da função e, no corpo do script, recebendo o valor do argumento passado na linha de comando.

Executando...

```
:~$ ./exemplo.sh Goku
Oi, eu sou Goku!
:~$ ./exemplo.sh Vegeta
Oi, eu sou Vegeta!
```

Além das substituições de comandos, nós podemos nos valer o fato de que **todas as variáveis no script são tratadas como globais, a menos que seja definido o contrário**. Desta forma, em vez de tentarmos obter da função o retorno de um valor, nós podemos fazer com que ela defina (ou redefina) variáveis.

Por exemplo:

```
#!/usr/bin/env bash

oi() {
    mensagem="Oi, eu sou $1!"
}

oi $1
echo $mensagem
```

Executando...

```
:~$ ./exemplo.sh Goku
Oi, eu sou Goku!
:~$ ./exemplo.sh Vegeta
Oi, eu sou Vegeta!
```

15.6 - Escopo de variáveis

Repetindo: *todas as variáveis no script são tratadas como globais, a menos que seja definido o contrário*, e só é possível definir variáveis locais dentro de funções, o que fazemos utilizando a palavra reservada `local` antes do nome da variável.

```
nome_da_função() {
    ...
    local NOME=VALOR
    ...
}
```

Por exemplo:

```
#!/usr/bin/env bash

msg="Oi, eu sou a Bulma!"

oi() {
   local msg="Oi, eu sou $1!"
   echo $msg
```

```
}

oi $1
echo $msg
```

Executando:

```
:~$ ./exemplo.sh Goku
Oi, eu sou Goku!
Oi, eu sou a Bulma!
:~$
```

Repare que, mesmo existindo duas variáveis com o mesmo nome (`msg`), seus valores não se misturam, porque uma está definida como local dentro do bloco de comandos da função. Portanto, o comando `echo` da função "sabe" que estamos nos referindo à variável local `msg`, e não à sua "xará" global.

15.7 - A variável 'FUNCNAME'

A nossa analogia das funções como *"pequenos scripts dentro de um script"* tem uma falha importante quando se trata de retornar o nome da função. A variável especial `$0` armazena o nome do script em execução, e isso não vai mudar se ele for acessado de dentro de uma função. Contudo, o Bash nos oferece uma variável interna que armazena o nome de todas as funções em execução, trata-se da array `FUNCNAME`.

Em `FUNCNAME`, o elemento de índice `0` é o nome da função atualmente em execução, e a variável só existe enquanto houver alguma função sendo executada no shell.

Por exemplo:

```
#!/usr/bin/env bash

Goku() {
    echo "Oi, eu sou ${FUNCNAME[0]}!"
}

Vegeta() {
    echo "Eu sou ${FUNCNAME[0]}, seu verme!"
}
```

```
Goku
Vegeta
```

Executando:

```
:~$ ./exemplo.sh
Oi, eu sou Goku!
Eu sou Vegeta, seu verme!
:~$
```

> ***Dica:*** *não é a forma mais correta, mas, como o valor do índice `0` de uma array indexada sempre será retornado quando a acessamos apenas com `$NOME_DA_ARRAY`, nós podemos obter o nome da função apenas com `$FUNCNAME`, sem as chaves e o `[0]`.*

15.8 - Diferenciando funções de comandos com o mesmo nome

É possível nomear funções com o mesmo nome de comandos que nós usamos no terminal. Na verdade, embora possa causar certa confusão dentro de scripts, isso é muito comum quando estamos definindo "aliases" e funções no arquivo `.bashrc` com o propósito de customizar o comportamento de comandos.

Quando isso acontece, é possível diferenciar as chamadas às funções das chamadas aos comandos originais com o builtin `command`.

Veja o exemplo:

```
#!/usr/bin/env bash

function ls {
   echo "Oi, eu sou uma função!"
}

ls
command ls
```

Que, resultaria em algo como...

```
:~$ ./exemplo.sh
Oi, eu sou uma função!
Desktop Documentos Downloads ...
:~$
```

Aqui, `ls` retornou a saída da função, mas `command ls` retornou os arquivos no diretório corrente.

> ***Importante!*** *Tome muito cuidado ao criar funções com nomes de comandos, pois isso pode causar grandes problemas!*