Tópicos Especiais em Fundamentos da Computação
Shell Scripting

Parâmetros e Variáveis

Andrei Rimsa Álvares
andrei@cefetmg.br

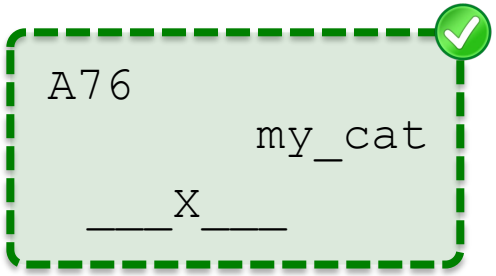
Sumário

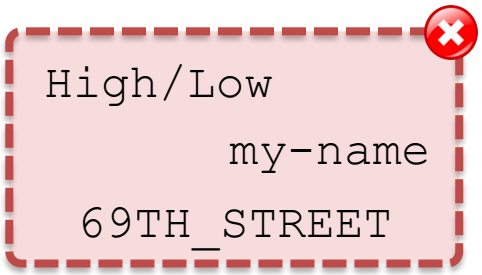
- Parâmetros e variáveis
  - Variáveis criadas pelo usuário
  - Atributos de variáveis
  - Variáveis reservadas
- Parâmetros e variáveis avançados
  - Arranjos (*arrays*)
  - Localidade
  - Especiais
  - Posicionais
  - Expansão

PARÂMETROS E VARIÁVEIS

Variáveis

- **Parâmetros da *shell*:** associado a valor que é acessível ao usuário
- Parâmetros cujos nomes consistem de letras, dígitos e/ou _ são comumente conhecidos como **variáveis de *shell*** (ou só **variáveis**)
- Uma variável deve começar com letra ou sublinhado, mas não com dígitos
- Exemplos





Tipos de Variáveis

- Existem vários tipos de variáveis de *shell*
  - **Criadas pelo usuário:** o usuário nomeia e define valores
    - Podem ser modificadas a qualquer momento, ou marcadas como *readonly* para que seus valores não sejam modificados
    - Podem ser globais (**variáveis de ambiente**)
  - **Reservadas:** possuem significado especial para a *shell*
    - Normalmente possuem nomes pequenos (mnemônicos)
    - Pode-se mudar os valores da maioria dessas variáveis
  - **Posicional e especiais:** permitem acessar argumentos em linha de comando (`$1`, `$*`, `$#`, `$@`, ...) e interações com a *shell* (`$?`, `$$`, `$!`)
    - Não se pode definir valores para essas variáveis

VARIÁVEIS CRIADAS PELO USUÁRIO

Variáveis Criadas pelo Usuário

- O seguinte exemplo declara uma variável com nome `person` e a inicializa com o valor `max`

```
$ person=max
$ echo person
person
$ echo $person
max
```

Lembrete: a variável é expandida quando possui um `$` precedente

Convenção: nomes de variáveis globais são capitalizados

Cuidado: não deve ter espaços entre os nomes

Escapes e Espaços

- Pode-se evitar a expansão usando aspas simples ou escapando com barra invertida, mas não com aspas duplas

```
$ echo $person "$person" '$person' \$person
max max $person $person
```

- Para definir um valor a uma variável que contém **espaços** e **tabulações**, use aspas duplas entre o valor

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
bash: and: command not found
```

Dica: embora as aspas duplas não sejam necessárias em todos os casos, é um bom hábito usá-las

Por que deu o erro `command not found`?

Escopo Local

- Podem ser definidas apenas para um determinado programa na linha de comando (escopo local ao programa)

```
$ cat > my_script << EOF
> echo \$TEMPDIR
> EOF
$ chmod +x my_script
$ TEMPDIR=~/tmp ./my_script
/home/rimsa/tmp
$ echo $TEMPDIR
```

Mais sobre localidade de variáveis a seguir

Expansão de Arquivos em Variáveis

- Quando se executa um comando usando uma variável como argumento, a *shell* troca o nome pelo valor (expande); se o valor tiver caracteres especiais (como `*` ou `?`), a *shell* **pode ou não** expandir o valor da variável

- Exemplo

```
$ ls
max.report
max.summary
$ memo=max*
$ echo "$memo"
max*
$ echo $memo
max.report max.summary
```

Não foi expandido com aspas duplas

Foi expandido sem aspas duplas

`read`: Aceita Entradas do Usuário

- Um dos usos mais comuns para variáveis criadas por usuários é armazenar informações que o usuário digita no *prompt* de comandos
- Usando o comando `read` um script pode aceitar entradas do usuário e armazenar em variáveis
- Exemplo

Cuidado: a *shell* pode interpretar a entrada do usuário se a variável for usada sem aspas dupla; ex.: asterisco

read1

```
#!/bin/sh

echo -n "Go ahead: "
read firstline
echo "You entered: $firstline"
```

```
$ ./read1
Go ahead: This is a line.
You entered: This is a line.
```

`read`: Aceita Entradas do Usuário

- Usar o comando `-p` para mostrar uma mensagem

read2a

```
#!/bin/sh

read -p "Go ahead: " firstline
echo "You entered: $firstline"
```

```
$ ./read2a
Go ahead: This is a line.
You entered: This is a line.
```

- Se não definir uma variável, a *shell* coloca na variável `$REPLY`

read2b

```
#!/bin/sh

read -p "Go ahead: "
echo "You entered: $REPLY"
```

```
$ ./read2b
Go ahead: This is a line.
You entered: This is a line.
```

- Lendo várias entradas ao mesmo tempo

read2c

```
#!/bin/sh

read -p "Enter something: " word1 word2 word3
echo "Word 1 is: $word1"
echo "Word 2 is: $word2"
echo "Word 3 is: $word3"
```

```
$ ./read2c
Enter something: a b c d
Word 1 is: a
Word 2 is: b
Word 3 is: c d
```

Sintaxe Opcional

- A sintaxe `$VARIABLE` é um caso especial da sintaxe mais geral `${VARIABLE}`, na qual a variável é envolvida em chaves

- Exemplo: são necessários quando se deseja concatenar uma variável com uma *string*

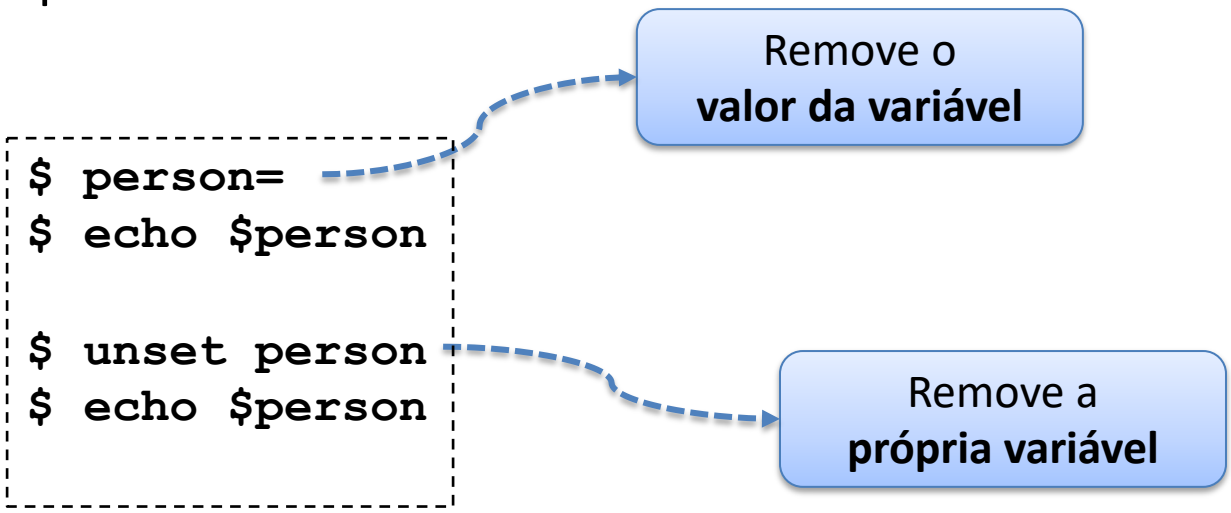
```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE
```

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
counterclockwise counterfeit
```

Removendo Variáveis

- A não ser que a remova explicitamente, uma variável continua viva enquanto a *shell* em que ela foi criada estiver em execução

- Exemplo



ATRIBUTOS DE VARIÁVEIS

Shell Scripting

Somente Leitura

- Pode-se usar o comando interno `readonly` para garantir que o valor da variável não pode ser alterado
- Deve-se definir um valor antes de torná-la `readonly`, depois não se pode alterar nem remover a variável
- Exemplo

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
bash: person: readonly variable
```

Dica: execute o comando `readonly` sem parâmetros para obter as variáveis somente leitura

Cuidado: não é possível remover uma variável depois de defini-la como somente leitura♣

♣: http://stackoverflow.com/questions/17397069/unset-readonly-variable-in-bash

Definir Atributos a Variáveis

- Os comandos internos `declare` e `typeset` (dois nomes para o mesmo comando) definem atributos para variáveis de *shell*

- Atributos que variáveis podem ter

Opção	Significado
`-a`	Declarar uma variável como um arranjo
`-f`	Declarar uma variável como um nome de função
`-i`	Declarar uma variável de tipo inteiro
`-r`	Declarar uma variável somente leitura (comando `readonly`)
`-x`	Declarar uma variável global (comando `export`)

Definir Atributos a Variáveis

- Definindo atributos usando o símbolo −

```
$ declare person1=max
$ declare -r person2=zach
$ declare -rx person3=helen
$ declare -x person4
```

- Retirando atributos usando o símbolo +

```
$ declare +x person3
```

Cuidado: não funciona para o atributo **somente leitura**

Listando os Atributos

- Pode-se listar todas as variáveis que possuem propriedades através do comando a seguir

```
$ declare -r
declare -ar BASH_VERSINFO='([0]="3" [1]="2" [2]="39" [3]="1" ... )'
declare -ir EUID="500"
declare -ir PPID="936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:..."
declare -ir UID="500"
declare -r person2="zach"
declare -rx person3="helen"
```

Variáveis Inteiras

- Por padrão, os valores de variáveis são armazenados internamente como *strings*
- Ao realizar operações aritméticas, a *shell* converte a variável em um número, executa a operação e depois a converte de volta para *string*
- Contudo, pode-se adicionar o atributo inteiro (`-i`) a uma variável para que internamente seja armazenada como inteiro

```
$ declare -i COUNT
```

VARIÁVEIS RESERVADAS

Variáveis Reservadas

- Variáveis reservadas são tanto herdadas ou são declaradas e inicializadas pela *shell* antes de começar (através de arquivos de configuração com o `/etc/profile`)

- Pode-se definir valores para essas variáveis a partir da linha de comando ou de um arquivo inicialização

- Tipicamente, deseja-se que alterações nessas variáveis sejam aplicadas em sua *login shell* e todas as *subshells* derivadas
  - Para aquelas que não são automaticamente exportadas pode-se usar o comando `export` explicitamente

$HOME: Onde Fica seu Diretório Pessoal

- Por padrão, seu diretório pessoal é o diretório de trabalho quando você faz *login* no sistema
- Quando você faz *login*, a *shell* recebe o caminho para seu diretório pessoal e o associa na variável HOME
- Essa variável é usada pelo comando `cd` (sem parâmetros) para ir para seu diretório pessoal; e pela *shell* na expansão de til

```
$ pwd
/Users/rimsa/tmp
$ echo $HOME
/Users/rimsa
$ cd
$ pwd
/Users/rimsa
```

```
$ pwd
/Users/rimsa/tmp
$ echo $HOME
/Users/rimsa
$ echo ~
/Users/rimsa
```

`$PATH`: Onde a Shell Procura por Programas

- Quando se passa um comando com o **caminho relativo ou absoluto**, a *shell* procura o programa nesse diretório com nome especificado

- Contudo, se der um comando como um nome simples, a shell procura em certos diretórios (*search path*) pelo programa que se deseja executar
  - A variável `$PATH` controla esses diretórios

- O valor padrão dessa variável é determinado quando a *shell* é compilada, mas pode ser modificado posteriormente por arquivos
  - Incluir normalmente diretórios que guardam programas, como `/bin` e `/usr/bin` e outros diretórios apropriados ao sistema

$PATH: Onde a Shell Procura por Programas

- A variável $PATH especifica os diretórios na ordem em que a *shell* deve procurar os programas, em que cada diretório é separado por um dois pontos (:)

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

- Pode-se alterar a variável $PATH para definir outros diretórios
  - Para usar o diretório de trabalho pode-se usar ponto (.); dois pontos (:) no começo, no final (como no exemplo abaixo) ou dois pontos consecutivos (: :)

```
$ export PATH=/usr/bin:/bin:/usr/sbin:/sbin:~/bin:
$ export PATH=$PATH:/opt/local/bin
```

Cuidado: por questões de segurança não se deve definir o diretório de trabalho

`$PS1`: *Prompt* de Usuário (Primário)

- O *prompt* padrão do *bash* é um sinal de dólar (`$`) e quando executado como root um sinal de tralha (`#`)
- A variável `PS1` controla o formato do *prompt*, que pode ser modificado para customizar sua aparência
- Exemplos

```
$ PS1="[\u@\h \W \!]$ "
[rimsa@rimsa ~ 510]$ echo test
test
$ PS1="$(hostname): "
bravo.example.com: echo test
test
```

`$PS1`: *Prompt* de Usuário (Primário)

- Símbolos especiais que podem ser usados para customizar o *prompt*

Símbolo	Mostrado no prompt
`\$`	Se executado como root `#`, caso contrário `$`
`\w`	Nome do caminho do diretório de trabalho
`\W`	Nome base do diretório de trabalho
`\!`	Número do último comando (*event*)
`\d`	Data no formato: dia da semana, mês, dia do mês
`\h`	*Hostname* da máquina, sem o domínio
`\H`	*Hostname* da máquina completo, incluindo o domínio
`\u`	Nome do usuário atual
`\@`	Hora atual no formato de `12` horas `AM/PM`
`\T`	Hora atual no formato de `12` horas `HH:MM:SS`
`\A`	Hora atual no formato de `24` horas `HH:MM`
`\t`	Hora atual no formato de `24` horas `HH:MM:SS`

`$PS2`: *Prompt* de Usuário (Secundário)

- A variável `PS2` armazena o *prompt* secundário (por padrão `>`) que é exibido quando se passa um comando incompleto para a *shell*
  - É usado para indicar que a *shell* espera que se termine o comando para que seja executado

- Exemplo: não fechar a aspas duplas em um comando `echo`

```
$ echo "primeira linha
> segunda linha"
primeira linha
segunda linha
```

- Alterando o `PS2` para mostrar dois pontos (`:`) ao invés de maior (`>`)

```
$ PS2=": "
$ echo "primeira linha
: 
```

$IFS: Separa Campos de Entrada (*Word Splitting*)

- A variável IFS (*Internal Field Separator*) especifica os caracteres que podem ser usados para separar argumentos em linha de comando
  - Por padrão tem o valor de **espaço**, **tab** e **nova linha**; independente do IFS, argumentos ainda podem ser separados por espaço ou tab
  - Quando definidos caracteres para IFS, esses também podem separar campos, mas somente em caso de expansão

- Exemplo

```
$ a=w:x:y:z
$ cat $a
cat: w:x:y:z: No such file or directory
$ IFS=":"
$ cat $a
cat: w: No such file or directory
cat: x: No such file or directory
cat: y: No such file or directory
cat: z: No such file or directory
```

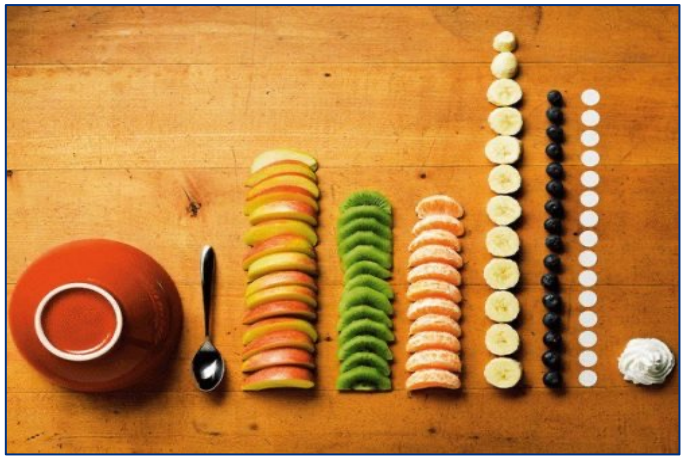
PARÂMETROS E VARIÁVEIS AVANÇADOS

Shell Scripting

Parâmetros e Variáveis Avançados

- Além do que já foi visto, serão introduzido os seguintes parâmetros e variáveis
  - Arranjos (*array*)
  - Localidade (variáveis globais/locais)
  - Parâmetros especiais
  - Parâmetros posicionais
  - Expansão de variáveis

ARRANJOS (*ARRAYS*)



Shell Scripting

Arranjos (*Arrays*)

- O *bash shell* suporta variáveis que são *arrays* unidimensionais, tal que os índices são inteiros que começam com `0` (zero)

```
name=(element1 element2 ...)
```

- Exemplo

```
$ NAMES=(max helen sam zach)
$ echo ${NAMES[0]}
max
$ echo ${NAMES[2]}
sam
```

Índices Especiais [@] e [*]

- Ambos os índices [@] e [*] extraem todo o arranjo

```
$ echo ${NAMES[@]}
max helen sam zach
```

```
$ echo ${NAMES[*]}
max helen sam zach
```

- Mas funcionam de forma diferente quando usados entre aspas duplas
  - [@]: produz um arranjo que é uma cópia do original
  - [*]: produz um único elemento de um array que guarda todos os elementos do *array* separados pelo primeiro caractere da variável IFS (normalmente um espaço)

```
$ A=("${NAMES[*]}")
$ B=("${NAMES[@]}")
$ declare -a
declare -a A='([0]="max helen sam zach")'
declare -a B='([0]="max" [1]="helen" [2]="sam" [3]="zach")'
...
```

Atribuição ao Arranjo

- Pode-se usar índices do lado esquerdo de uma atribuição para trocar o valor do elemento selecionado do arranjo

```
$ echo ${NAMES[*]}
max helen sam zach
$ NAMES[1]=max
$ echo ${NAMES[*]}
max max sam zach
```

LOCALIDADE DE VARIÁVEIS

Shell Scripting

Localidade de Variáveis

- Por padrão, variáveis são locais ao processo no qual são declaradas

- Portanto, um *shell script* não tem acesso às variáveis declaradas na *login shell*, a não ser que as torne explicitamente disponíveis (global)

showname

```
#!/bin/sh

echo "My name is: $NAME"
```

```
$ NAME="Andrei"
$ ./showname
My name is:
```

- O comando `export` torna uma variável disponível aos processos filho

```
$ export NAME="Andrei"
$ ./showname
My name is: Andrei
```

Dica: pode-se transformar uma variável local em global chamando `export` com o nome da variável

export

- Quando se define uma variável global (através do `export`), a *shell* coloca o valor da variável no ambiente de chamada do processo filho
- A variável é passada **por valor**, ou seja, cada processo filho recebe uma **cópia** da variável para seu próprio uso
- Exemplo

updatename

```
#!/bin/sh

echo "My name is: $NAME"
NAME="new name"
echo "New name is: $NAME"
```

```
$ export NAME="Andrei"
$ ./updatename
My name is: Andrei
New name is: new name
$ echo $NAME
Andrei
```

Funções

- Como funções são executadas no mesmo ambiente que a *shell* que as chamam, variáveis são implicitamente compartilhadas entre ela

```
$ function nam() {
>   echo $myname
>   myname=zach
> }
$ myname=sam
$ nam
sam
$ echo $myname
zach
```

Variáveis Locais a Funções

- Variáveis locais são úteis em funções de propósito geral
- Como funções podem ser chamadas por vários *scripts* diferentes, deve-se garantir que o nome das variáveis usadas dentro da função não entrem em conflito (duplicada) com nomes de variáveis nos programas que chamam essas funções
- Variáveis locais eliminam esse problema; o comando interno `typeset`, quando declarado dentro de uma função, torna o escopo da variável local a função em que é definida

Variáveis Locais a Funções

- Exemplo

```
$ function count_down() {
>   typeset count
>   count=$1
>   while [ $count -gt 0 ]; do
>     echo "$count..."
>     ((count=count-1))
>     sleep 1
>   done
>   echo "Blast Off."
> }
```

```
$ count=10
$ count_down 4
4...
3...
2...
1...
Blast Off.
$ echo $count
10
```

Dica: declarar a variável como `local` tem o mesmo efeito

PARÂMETROS ESPECIAIS

Parâmetros Especiais

- Parâmetros especiais permitem que se acesse valores pertinentes aos argumentos em linha de comando e a execução de programas

- Para acessar um parâmetro especial, basta preceder o símbolo de `$` antes do caractere especiais

  - `$$`: Identificador do processo atual
  - `$!`: Identificador do último processo no plano de fundo
  - `$?`: Status de saída
  - `$RANDOM`: Gerador de números aleatórios

Cuidado: não se pode alterar esses parâmetros especiais

`$$`: Identificador do Processo Atual

- A *shell* armazena no parâmetro especial `$$`, o identificador do processo (*PID*) do programa em execução

- Exemplo: para obter o *PID* da *shell*

```
$ echo $$
5209
$ ps
   PID TTY          TIME CMD
  5209 pts/1    00:00:00 bash
  6015 pts/1    00:00:00 ps
```

Repare que é exibido o *PID* da *shell*, e não do comando `echo`, já que a expansão é feita antes de executar o comando

$$: Identificador do Processo Atual

- Outro exemplo: a *shell* cria um novo processo quando executa um *shell script*

id2

```
#!/bin/sh

echo "id2 PID = $$"
```

```
$ echo $$
8232
$ ./id2
id2 PID= 8362
$ echo $$
8232
```

`$!`: Identificador do Último Processo no Plano de Fundo

- A *shell* armazena o identificador do último processo que executou no plano de fundo (*background*) no parâmetro especial `$!`

- Exemplo: executa o comando `sleep` em *background* e mostra o seu identificador de processo

```
$ sleep 60 &
[1] 8376
$ echo $!
8376
$ ps | grep sleep
  PID TTY          TIME CMD
 8376 pts/1    00:00:00 sleep 60
```

`$?`: Status de Saída

- Quando um processo termina sua execução por qualquer motivo, ele retorna um **status de saída** (código de retorno ou condição de saída) para o processo pai (aquele que executou o programa)

- O parâmetro especial `$?` armazena o status de saída do último comando executado
  - status **zero**: indica uma execução **com sucesso**
  - status **não-zero**: indica uma execução **sem sucesso**

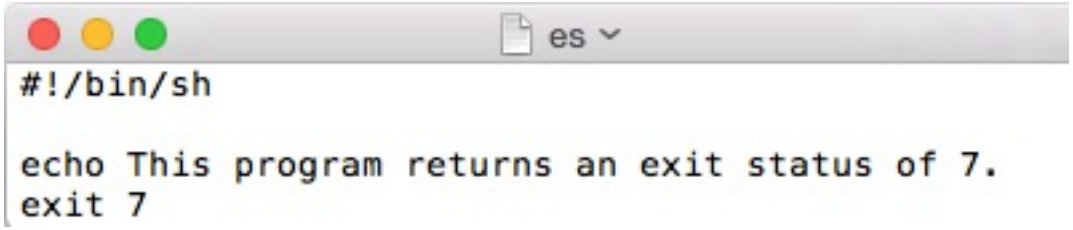
- Exemplo

```
$ ls es
es
$ echo $?
0
```

```
$ ls xxx
ls: xxx: No such file or directory
$ echo $?
1
```

`$?`: Status de Saída

- Pode-se especificar o status de saída usando o comando interno `exit`, seguido de um número que indica o código de retorno




```
$ ./es
This program returns an exit status of 7.
$ echo $?
7
$ echo $?
0
```

Qual será a saída se executarmos o comando `echo $?` novamente?

$RANDOM: Gerador de Números Aleatórios

- Toda vez que se usa essa variável, a *shell* a expande para um valor inteiro entre 0 e 32767

```
$ echo $RANDOM
31812
$ echo $RANDOM
22588
```

- Pode ser usado em operações aritméticas

```
$ echo $((RANDOM % 10))
4
```

Cuidado: se remover essa variável (`unset`) ela perde sua propriedade especial

PARÂMETROS POSICIONAIS

Shell Scripting

Parâmetros Posicionais

- **Parâmetros posicionais** incluem o nome do comando e os argumentos de linha de comando

- Esses parâmetros são chamados **posicionais** porque, em um *shell script*, se referencia eles através de suas posições na linha de comando

- Parâmetros posicionais
  - `$#`: número de parâmetros na linha de comando
  - `$0`: nome do programa chamado
  - `$1` até `$n`: argumentos em linha de comando
  - `$*` e `$@`: representam todos os argumentos

Apenas os comandos internos `set` e `shift` podem alterar esses parâmetros

$#: Número de Parâmetros

- O parâmetro $# armazena o número de argumentos na linha de comando (parâmetros posicionais), sem incluir o próprio comando

- Exemplo: mostrar a quantidade de argumentos na linha de comando

num_args

```
#!/bin/sh

echo "This script was called with $# arguments."
```

```
$ ./num_args sam max zach
This script was called with 3 arguments.
```

`$0`: Nome do Programa Chamado

- A *shell* armazena o nome do comando utilizado para chamar o programa no parâmetro `$0`
  - Esse parâmetro é nomeado zero porque aparece antes do primeiro argumento da linha de comando
- Exemplo: mostrar o programa executado

Dica: pode-se usar o comando `$(basename $0)` para extrair apenas o nome do comando

abc

```
#!/bin/sh

echo "The command used to run this script is $0"
```

```
$ ./abc
The command used to run this script is ./abc
$ ~rimsa/abc
The command used to run this script is /home/rimsa/abc
```

$1 até $n: Argumentos em Linha de Comando

- O primeiro argumento da linha de comando é representado pelo parâmetro $1, o segundo argumento por $2 e assim por diante
- Para valores maiores que 9, deve-se colocar o número envolvido em chaves (ex.: ${12} é o décimo segundo argumento)
- Exemplo: mostrar os cinco primeiros argumentos

display_5args

```
#/bin/sh

echo First 5 arguments are $1 $2 $3 $4 $5
```

A *shell* associa o valor null para parâmetros posicionais cujo argumento não estão presentes (ex.: $4 e $5)

```
$ ./display_5args zach max helen
First 5 arguments are zach max helen
```

`$*` e `$@`: Argumentos em Linha de Comando

- Os parâmetros `$*` e `$@` representam todos os argumentos em linha de comando

- Exemplo: mostrando todos os parâmetros

display_all

```
#!/bin/sh

echo All arguments are $*
```

```
$ ./display_all a b c d e f g h i j k l m n o p
All arguments are a b c d e f g h i j k l m n o p
```

$* e $@: Argumentos em Linha de Comando

- Considere o exemplo a seguir

showargs

```
#!/bin/sh

echo "$0 was called with $# arguments, the first is ($1)."
```

```
$ ./showargs a b c
./showargs was called with 3 arguments, the first is (a).
```

- É uma boa ideia envolver referências a parâmetros posicionais entre aspas duplas, principalmente na linha de comando

```
$ echo $xx

$ ./showargs $xx a b c
./showargs was called with 3 arguments, the first is (a).
$ ./showargs "$xx" a b c
./showargs was called with 4 arguments, the first is ().
```

$* vs $@

- Os parâmetros $* e $@ funcionam da mesma forma, a não ser quando envoltos em aspas duplas; $* resulta em um único argumento (separados por espaço ou primeiro caractere de IFS), enquanto $@ produz uma lista em que cada parâmetro posicional é um argumento

bb1

```
#!/bin/sh

set "$*"
echo $# parameters with '"$*"'
echo 1: $1
echo 2: $2
echo 3: $3
```

bb2

```
#!/bin/sh

set "$@"
echo $# parameters with '"$@"'
echo 1: $1
echo 2: $2
echo 3: $3
```

```
$ ./bb1 a b c
1 parameters with "$*"
1: a b c
2:
3:
```

```
$ ./bb2 a b c
3 parameters with "$@"
1: a
2: b
3: c
```

Dica: na prática, $@ é mais útil que $*

`shift`: Promove Argumentos

- O comando interno `shift` promove cada argumento; o primeiro é descartado, o segundo (que era `$2`) se torna o primeiro (agora `$1`), o terceiro se torna o segundo e assim por diante

demo_shift — Edited

```
#!/bin/bash

echo "arg1=$1      arg2=$2      arg3=$3"
shift
echo "arg1=$1      arg2=$2      arg3=$3"
shift
echo "arg1=$1      arg2=$2      arg3=$3"
shift
echo "arg1=$1      arg2=$2      arg3=$3"
shift
```

```
$ ./demo_shift alice helen zach
arg1=alice    arg2=helen    arg3=zach
arg1=helen    arg2=zach     arg3=
arg1=zach     arg2=         arg3=
arg1=         arg2=         arg3=
```

Cuidado: como não existe o comando `unshift`, não se pode recuperar argumentos que foram descartados

Dica: `shift` pode ser usado com um argumento que especifica quantos parâmetros arredar (padrão: `1`)

`set`: Inicializa Argumentos em Linha de Comando

- Quando o comando interno `set` é chamado com um ou mais argumentos, ele associa os valores aos parâmetros posicionais (começando em `$1`)

- Exemplo: associando valores aos parâmetros `$1`, `$2` e `$3`

set_it

```
#!/bin/sh

set this is it
echo $3 $2 $1
```

```
$ ./set_it
it is this
```

`set`: Inicializa Argumentos em Linha de Comando

- Combinar substituição de comando com o comando interno `set` é uma forma conveniente de formatar a saída padrão que pode ser facilmente manipulado pelo *shell script* posteriormente

dateset

```
#!/bin/sh

set $(date)
echo $*
echo
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
echo "Argument 6: $6"
echo
echo "$2 $3, $6"
```

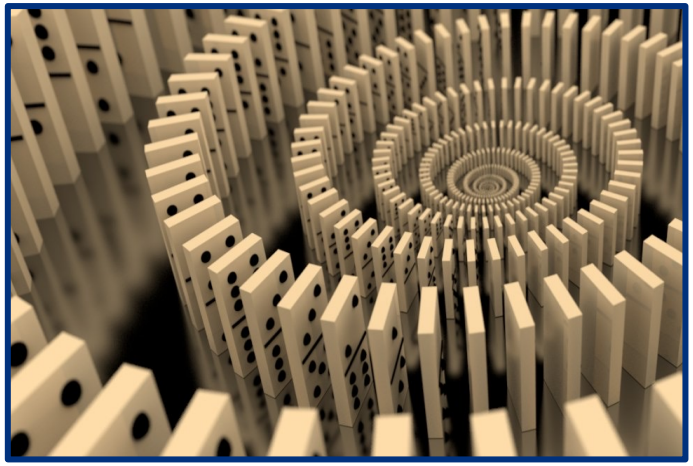
```
$ date
Wed Aug 13 17:35:29 PDT 2008
$ ./dateset
Wed Aug 13 17:35:34 PDT 2008

Argument 1: Wed
Argument 2: Aug
Argument 3: 13
Argument 6: 2008

Aug 13, 2008
```

Dica: pode-se usar os formatadores (`+format`) do comando `date`

EXPANSÃO DE VARIÁVEIS



Shell Scripting

Expansão de Variáveis

- A expressão `${name}` (ou somente `$name` se não for ambíguo) expande para o valor da variável `name`
  - Se `name` não estiver definida ou for `null`, a *shell* expande para a string `null`

- A *shell* provê as seguintes alternativas para a expansão da string `null` como o valor da variável
  - Usar um valor padrão (`:-`)
  - Usar um valor padrão e atribuí-lo a variável (`:=`)
  - Mostrar uma mensagem de erro e abortar o programa (`:?`)
  - Expandir padrão se existir (`:+`)

Dica: pode-se usar `set -o nounset` para fazer com que o *bash* mostre uma mensagem de erro e aborte o *script* sempre que uma variável indefinida é referenciada

Usar um Valor Padrão (:-)

- O modificador `:-` usa o valor padrão no lugar de variáveis `null` ou indefinidas, enquanto permite que uma variável não nula represente a si própria

  `${name:-default}`

  - A *shell* interpreta `:-` como: se `name` é `null` ou indefinida, expanda `default` e use esse valor, caso contrário expanda o conteúdo de `name`

- Exemplos: listar o conteúdo do diretório na variável `LIT`; caso não exista usar o diretório `/home/max/literature`

```
$ ls ${LIT:-/home/max/literature}
...
$ ls ${LIT:-$HOME/literature}
...
```

Dica: pode usar variáveis dentro do default

Definir um Valor Padrão (`:=`)

- O modificador `:-` não altera o valor da variável; contudo, é possível modificar o valor de uma variável `null` ou indefinida para seu valor padrão em um *script* usando o modificador `:=`

```
${name:=default}
```

  - A *shell* expande a expressão `${name:=default}` do mesmo jeito que `${name:-default}`, mas também define o valor de `name` para o valor expandido `default`

- Exemplo

```
$ ls ${LIT:=/home/max/literature}
...
```

O Comando Interno : (null)

- *Shell scripts* frequente começam com o comando interno : (null) seguido na mesma linha da expansão de variáveis com := para definir qualquer variável que pode ser null ou indefinida

- O comando interno : avalia cada *token* do comando que se segue, mas não executa nenhum comando
  - Se não tivesse o comando : no começo, a *shell* avaliaria a expansão e tentaria executar o comando resultante dessa avaliação

- Exemplo: definir a variável TMPDIR para /tmp se não estiver definida previamente

```
: ${TEMPDIR:=/tmp}
```

Mostrar uma Mensagem de Erro (`:?`)

- Às vezes um *script* precisa de uma variável, mas não é capaz de prover um valor padrão razoável para ser usado

- Se a variável for `null` ou indefinida, o modificador `:?` faz com que o script mostra uma mensagem e termine com um status de saída `1`

```
${name:?message}
```

  - Se omitir a mensagem, a *shell* apresenta uma mensagem padrão

- Exemplo

```
$ cd ${TESTDIR:?$(date +%T) error, variable not set.}
bash: TESTDIR: 16:16:14 error, variable not set.
```

Expandir Padrão Se Existir (`:+`)

- Se a variável `name` não for vazia (definida ou não `null`), então expande-se para o valor `default` (padrão)

```
${name:+default}
```

- Exemplo

```
$ VAR=test
$ echo ${VAR:+unknown}
unknown
$ unset VAR
$ echo ${VAR:+unknown}
```

Para quê serve isso?

Outras Expansões

- Existem ainda outras expansões de variáveis
  - `${parameter:offset}` e `${parameter:offset:length}`
  - `${#parameter}`
  - `${!parameter}`
  - `${parameter#word}` e `${parameter##word}`
  - `${parameter%word}` e `${parameter%%word}`
  - `${parameter/pattern/string}` e `${parameter//pattern/string}`

`${parameter:offset}`
`${parameter:offset:length}`

- Expansão de *substring*
  - `offset` indica o caractere de início
  - `length` indica a quantidade de caracteres; se não for especificado expande-se até o final

- Exemplo

```
$ PHONE="(31) 4321-5678"
$ echo ${PHONE:5}
4321-5678
$ echo ${PHONE:0:4}
(31)
```

${#parameter}

- Usado para obter o tamanho (*length*) de valor de uma variável

```
$ echo ${PWD}
/tmp
$ echo ${#PWD}
4
```

- Pode-se obter também o tamanho de um arranjo (quantidade de elementos), expandindo o arranjo com índices * ou @

```
$ echo ${NAMES[*]}
max helen sam zach
$ echo ${#NAMES[*]}
4
```

  - Para obter o tamanho de um elemento do arranjo, basta referenciar pelo índice

```
$ echo ${#NAMES[1]}
5
```

${!parameter}

- Adiciona um nível de indireção a expansão

```
$ foo=bar
$ bar=abc
$ echo ${!foo}
abc
```

- Exceto em dois casos
  - Quando se usa o asterisco após um prefixo: `${!prefix*}`

```
$ myName=John
$ myLastName=Doe
$ echo ${!my*}
myLastName myName
```

  - Ou quando se usa com um arranjo: `${!array[@]}`

```
$ msg=(hello my friend)
$ echo ${!msg[*]}
0 1 2
$ for idx in ${!msg[*]}; do
>   echo ${msg[$idx]};
> done
hello
my
friend
```

O comando `for` será visto em mais detalhes posteriormente

${parameter#word}
${parameter##word}

- O operador # remove o **prefixo casado mínimo**, enquanto ## remove o **prefixo casado máximo**, tal que `word` é expandido como expansão de arquivos

- Exemplo

```
$ SOURCEFILE="/usr/local/src/prog.c"
$ echo ${SOURCEFILE#/*/}
local/src/prog.c
$ echo ${SOURCEFILE##/*/}
prog.c
```

`${parameter%word}`
`${parameter%%word}`

- O operador `%` remove o **sufixo casado mínimo**, enquanto `%%` remove o **sufixo casado máximo**, tal que `word` é expandido como expansão de arquivos

- Exemplo

```
$ SOURCEFILE="/usr/local/src/prog.c"
$ echo ${SOURCEFILE%/*}
/usr/local/src
$ echo ${SOURCEFILE%%/*}

$ echo ${SOURCEFILE%.c}
/usr/local/src/prog
```

${parameter/pattern/string}

- O `pattern` é expandido para encontrar o **primeiro padrão** assim como em expansão de arquivos, mas substitui `string` com o maior casamento da expansão de `parameter`

- Exemplo

```
$ SOURCEFILE="/usr/local/src/prog.c"
$ echo ${SOURCEFILE/.c/.cpp}
/usr/local/src/prog.cpp
$ echo ${SOURCEFILE/\/usr\/local/.}
./src/prog.c
$ echo ${SOURCEFILE/*\//}
prog.c
```

${parameter//pattern/string}

- Similar ao anterior, mas as `//` antes do `pattern` indicam que deve-se substituir **todas as ocorrências** do padrão casada por `string` para a expansão do `parameter`

- Exemplo: substituir todas as ocorrências de `:` por espaços em branco

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
$ echo ${PATH/:/ }
/usr/local/bin /usr/bin:/bin:/usr/sbin:/sbin
$ echo ${PATH//:/ }
/usr/local/bin /usr/bin /bin /usr/sbin /sbin
```

ISSO É TUDO, PESSOAL!