

# Dá para fazer em shell?

Por Julio Cezar Neves



# Sumário

---

Sumário.....	2
Prefácio.....	3
1. Sincronismo de processos assíncronos com named pipes.....	5
2. Substituição de Processos.....	6
3. Algumas implementações do Bash 4.0.....	8
3.1. 1. Novas substituições de parâmetros.....	8
3.2. 2. Substituição de chaves.....	10
4. Colunador.....	12
5. Criando Animações com ImageMagick.....	15
6. notify-send.....	17
7. YAD - Yet Another Dialog.....	23
8. Fatiando opções com o getopt.....	27
8.1. Manipulação de erros.....	28

## Prefácio

---

Muito me alegra o fato de você gostar de *Shell* e ter acompanhado (ou, pelo menos, tentado acompanhar) os vídeos que desenvolvi com algumas pequenas dicas desta linguagem.

Confesso que quando fiz os vídeos minha preocupação não era ensinar *Shell*, já que o assunto é muito vasto e não seria meia dúzia de vídeos que conseguiria passar um material tão vasto. Tinha três preocupações:

1. Mostrar comandos ou técnicas que em 90% dos casos que via, estavam usados de maneira indevida. Acho que o workshop que melhor representou esse meu desejo foi a trilogia **if + test + conectores**;
2. Mostrar soluções de problemas usando poucas linhas de código, como nos *workshops* sobre o comando **paste** e o de *one-liners*;
3. Mostrar comandos ou técnicas daqueles que as pessoas olham e torcem o nariz, porque não fazem a menor ideia do que aquele “troço”, como foi as dicas sobre **yad** e **xargs**.

Essa nova coleção de dicas continua sem a pretensão de ensinar *Shell*, tendo como diretriz ir mostrando comandos, técnicas e macetes pouco conhecidos como:

- A suíte Image Magick (formada por 11 utilitários, mas que os poucos que já ouviram falar dela, sabem usar um pouquinho só do utilitário **convert**);
- O yad, que é conhecido como o zenity com esteróides;
- O **notify-send**;
- O **getopts** que separa as opções passadas para o programa; coprocessos e
- Outros

Tem também algumas coisinhas que quebram o maior galho, tais como:

- *Named Pipes*;
- Substituição de processos;
- A variável **\$IFS**;
- Como receber dados via *pipe* e como mandar dados para a saída de erros (no programa **colunador.sh**).

E outras (já que o assunto é *Shell*) coisas mais...

Pessoal, espero que vocês gostem e aproveitem. Foi feito de muito boa vontade com o objetivo de divulgar essa excelente e pouquíssima conhecida linguagem *Shell*.

# 1. IFS: Inter Field Separator

---

O *Shell* tem uma variável interna chamada **IFS** - *Inter Field Separator* (será Tabajara? :) - cujo valor *default* podemos obter da seguinte forma:

```
$ echo "$IFS" | od -h
00000000 0920 0a0a
00000004
```

O programa **od** com a opção **-h** foi usado para gerar um *dump* hexadecimal da variável. E lá podemos ver:

Valor Hexadecimal	Significado
09	<TAB>
20	Espaço
0a	<ENTER>

Ou seja os separadores entre campos (tradução livre de **IFS**) *default* são o **<TAB>**, o espaço em branco e o **<ENTER>**. O **IFS** é usado em diversas instruções, mas seu uso é muito comum em par com o **for** e/ou com o **read**. Vejamos mais um exemplo:

```
$ cat script1.sh
#!/bin/bash
while read linha
do
    awk -F: '{print $1}' <<< $linha > /dev/null
done < /etc/passwd
```

Como podemos ver este *script* não faz nada (sua única saída foi enviada para **/dev/null** para não deturpar os tempos de execução), mas vamos usá-lo para avaliação dos tempos de execução.

Este *script* foi alterado, trocando o **awk** pelo **cut**, e ficando com a seguinte cara:

```
$ cat script2.sh
#!/bin/bash
while read linha
do
    echo $linha | cut -f1 -d: > /dev/null
done < /etc/passwd
```

Mais uma outra alteração, usando somente intrínsecos do *Shell* desta vez tomando partido do **IFS**:

```
$ cat script3.sh
#!/bin/bash
while IFS=: read user lixo
do
    echo $user> /dev/null
done < /etc/passwd
```

Neste último exemplo, transformamos o separador padrão em dois-pontos (:) e usamos sua propriedade em conjunto com o `read`, isto é, o primeiro campo veio para a variável `user` e o resto para a variável `lixo`.

É interessante notar que alguns comandos do *Shell* permitem que se troque o valor de algumas variável do sistema somente durante o tempo de sua execução e foi esse macete que apliquei no `IFS` que acabamos de ver, que ficou com o valor de dois pontos (:) somente durante a execução do `read`.

Quer ver outro caso? Então olha só:

```
$ echo $LANG          # Contém a linguagem em uso
pt_BR.UTF-8
$ date                # Em pt_BR.UTF-8
Sáb Mar 17 18:36:56 -03 2018
$ LANG=en_EN date    # Em inglês
Sat Mar 17 18:37:56 -03 2018
$ echo $LANG          # Voltou ao pt_BR.UTF-8?
pt_BR.UTF-8
```

Em seguida, fiz um *script* usando *Shell* puro. Repare a construção `${linha%%:*}`, que é um intrínseco (*built-in*) do *Shell* que serve para excluir da variável `linha` o maior casamento com o padrão especificado (`:*` - que significa "de dois-pontos em diante"), ou seja, excluiu de linha tudo a partir do último dois pontos, contando da direita para a esquerda.

```
$ cat script4.sh
#!/bin/bash
while read linha
do
    echo ${linha%%:*} > /dev/null
done < /etc/passwd
```

Para finalizar, adaptei o *script* escrito pelo incansável Rubens Queiroz que, exceto pelo `awk`, é *Shell* puro.

```
$ cat script5.sh
#!/bin/bash
for user in `awk -F: '{print $1}' /etc/passwd`
```

```
do
    echo $user > /dev/null
done
```

Vou criar mais dois *scripts*, que, no duro, são simples comandos porque o *loop* é desnecessário, mas para esse estudo são importantes:

```
$ cat script6.sh
#!/bin/bash
cut -f1 -d: /etc/passwd > /dev/null
$ cat script7.sh
#!/bin/bash
awk -F: '{ print $1; }' /etc/passwd > /dev/null
```

Agora, o mais importante: reparem os tempos da execução de cada um deles:

```
$ Vermelho=$(tput setaf 1)
$ Branco=$(tput setaf 7)
$ # Saída vermelha em fundo preto para realçar e homenagear meu mengão
;)
$ for i in script[1-7].sh
> do
>     echo $Vermelho===Tempos do $i==$Branco
>     time $i
> done
===Tempos do script1.sh===

real 0m0.065s
user 0m0.004s
sys 0m0.000s
===Tempos do script2.sh===

real 0m0.032s
user 0m0.008s
sys 0m0.004s
===Tempos do script3.sh===

real 0m0.002s
user 0m0.000s
sys 0m0.000s
===Tempos do script4.sh===

real 0m0.002s
user 0m0.000s
sys 0m0.000s
===Tempos do script5.sh===

real 0m0.003s
user 0m0.000s
sys 0m0.000s
===Tempos do script6.sh===

real 0m0.002s
user 0m0.000s
sys 0m0.000s
===Tempos do script7.sh===
```

```
real 0m0.003s
user 0m0.000s
sys 0m0.000s
```

Reparem que estas diferenças de tempo foram obtidas para um arquivo com somente 41 linhas.  
Veja:

```
$ wc -l /etc/passwd
41 /etc/passwd
```

Um outro uso interessante do **IFS** é o que vemos a seguir, primeiramente usando o IFS default que como vimos é **<TAB>**, Espaço e **<ENTER>**:

```
$ Frutas="Pera Uva Maçã"
$ set $Frutas
$ echo $1
Pera
$ echo $3
Maçã
```

Agora, vamos alterar o **IFS** para fazer o mesmo com uma variável qualquer, e para tal vamos continuar usando o famigerado **/etc/passwd**:

```
$ Root=$(head -1 /etc/passwd)
$ echo $Root
root:x:0:0:root:/root:/bin/bash
$ oIFS="$IFS"
$ IFS=:
$ set - $Root
$ echo $1
root
$ echo $7
/bin/bash
$ IFS="$oIFS"
```

Senhoras e Senhores, neste artigo pretendi mostrar duas coisas:

1. O uso do **IFS** que, infelizmente para nós, é uma variável pouco conhecida do *Shell* e
2. Mostrar que o *Shell* não é lento, normalmente é mal usado/programado. Quanto mais intrínsecos do *Shell* usamos, mais veloz e performático fica o *script*, igualando em tempo até um *loop* de instruções com simples e únicos comandos.



## 2. Sincronismo de processos assíncronos com named pipes

---

Vamos falar hoje em *named pipes*. Você sabia que pode sincronizar 2 ou mais processos assíncronos, trocando informações entre eles usando esta técnica? Deixa eu te mostrar: abra 2 terminais no mesmo diretório e em um deles faça:

```
$ mkfifo paipi
$ ls -l paipi
prw-r--r-- 1 julio julio 0 Nov  4 18:08 paipi
```

Viu!? É um arquivo do tipo p e se o seu **ls** for colorido, verá que seu nome tem uma cor de burro quando foge. Agora em um terminal escreva:

```
cat paipi
```

Calma, não se desespere! Ele não congelou (pinguim não congela, janelas congelam ;), ele está ouvindo uma ponta do *named pipe*, esperando que se fale algo na outra ponta. Então vamos para o outro terminal para falar. Redirecione qualquer saída para o *named pipe* que ela “miraculosamente” aparecerá no primeiro terminal, que a esta altura já não terá aparência de “congelado”. Por exemplo, faça:

```
ls -l > paipi
```

E dessa forma podemos trocar dados entre 2 processos. Genial, não é?

### 3. Substituição de Processos

---

O *Shell* também usa os *named pipes* de uma maneira bastante singular, que é a substituição de processos (*process substitution*). Uma substituição de processos ocorre quando você põe um `<` ou um `>` grudado na frente do parêntese da esquerda. Teclando-se o comando:

```
$ cat <(ls -l)
```

Resultará no comando `ls -l` executado em um *subShell* como é normal, porém redirecionará a saída para um *named pipe* temporário, que o *Shell* cria, nomeia e depois remove. Então o `cat` terá um nome de arquivo válido para ler (que será este *named pipe* e cujo dispositivo lógico associado é `/dev/fd/63`), e teremos a mesma saída que a gerada pela listagem do `ls -l`, porém dando um ou mais passos que o usual.

Como poderemos constatar isso? Fácil ... Veja o comando a seguir:

```
$ ls -l >(cat)
l-wx----- 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É... Realmente é um *named pipe*.

Você deve estar pensando que isto é uma maluquice de nerd, né? Então suponha que você tenha 2 diretórios: `dir` e `dir.bkp` e deseja saber se os dois estão iguais (aquela velha dúvida: será que meu backup está atualizado?). Basta comparar os dados dos arquivos dos diretórios com o comando `cmp`, fazendo:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) || echo backup furado
```

ou, melhor ainda:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) >/dev/null || echo backup furado
```

Este é um exemplo meramente didático, mas são tantos os comandos que produzem mais de uma linha de saída, que serve como guia para outros. Eu quero gerar uma listagem dos meus arquivos, numerando-os e ao final dar o total de arquivos do diretório corrente:

```
while read arq
do
```

```
((i++)) # assim nao eh necessario inicializar i
echo "$i: $arq"
done < <(ls)
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Tá legal, eu sei que existem outras formas de executar a mesma tarefa. Mas tente fazer usando **while**, sem usar substituição de processos que você verá que este método é muito melhor.

## 4. Algumas implementações do Bash 4.0

---

### 4.1. Novas substituições de parâmetros

---

```
${parâmetro:n}
```

Equivale a um `cut` com a opção `-c`, porém muito mais rápido. Preste atenção, pois neste caso, a origem da contagem é zero.

```
$ TimeBom=Flamengo
$ echo ${TimeBom:3}
mengo
```

Essa Expansão de Parâmetros que acabamos de ver, também pode extrair uma subcadeia, do fim para o princípio, desde que seu segundo argumento seja negativo.

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

Isto equivale a uma indireção, ou seja, devolve o valor apontado por uma variável cujo nome está armazenada em parâmetro.

### Exemplo

```
$ Ponteiro=VariavelApontada
$ VariavelApontada="Valor Indireto"
$ echo "A variável \${Ponteiro} aponta para \"\${Ponteiro}\"
> que indiretamente aponta para \"\${!Ponteiro}\""
```

A variável `$Ponteiro` aponta para `"VariavelApontada"` que indiretamente aponta para `"Valor Indireto"`

```
${!parâmetro@}
${!parâmetro*}
```

Ambas expandem para os nomes das variáveis prefixadas por parâmetro. Não notei nenhuma diferença no uso das duas sintaxes.

## Exemplos

Vamos listar as variáveis do sistema começadas com a cadeia **GNOME**:

```
$ echo ${!GNOME@}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
$ echo ${!GNOME*}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
${parâmetro^}
${parâmetro,}
```

Essas expansões foram introduzidas a partir do Bash 4.0 e modificam a caixa das letras do texto que está sendo expandido. Quando usamos circunflexo (^), a expansão é feita para maiúsculas e quando usamos vírgula (,), a expansão é feita para minúsculas.

## Exemplo

```
$ Nome="botelho"
$ echo ${Nome^}
Botelho
$ echo ${Nome^^}
BOTELHO
$ Nome="botelho carvalho"
$ echo ${Nome^}
Botelho carvalho      # Que pena...
```

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

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

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

No Windows, além dos vírus e da instabilidade, também são frequentes nomes de arquivos com espaços em branco e quase todos em maiúsculas. No exemplo anterior, vimos como trocar os espaços em branco por sublinha (\_), no próximo veremos como passá-los para minúsculas:

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

for Arq in *[A-Z]*      # Pelo menos 1 minúscula
do
    if [ -f "${Arq,}" ]  # Arq em minúsculas já existe?
    then
        echo ${Arq,,} já existe
    else
```

```
mv "$Arq" "${Arq,,}"  
fi  
done
```

## 4.2. Substituição de chaves

---

Elas são usadas para gerar cadeias arbitrárias, produzindo todas as combinações possíveis, levando em consideração os prefixos e sufixos.

Existiam 5 sintaxes distintas, porém o Bash 4.0 incorporou uma 6ª. Elas são escritas da seguinte forma:

1. `{lista}`, onde lista são cadeias separadas por vírgulas;
2. `{inicio..fim}`;
3. `prefixo{****}`, onde os asteriscos (`****`) podem ser substituídos por lista ou por um par `inicio..fim`;
4. `{****}sufixo`, onde os asteriscos (`****`) podem ser substituídos por lista ou por um par `inicio..fim`;
5. `prefixo{****}sufixo`, onde os asteriscos (`****`) podem ser substituídos por lista ou por um par `inicio..fim`;
6. `{inicio..fim..incr}`, onde `incr` é o incremento (ou razão, ou passo). Esta foi introduzida a partir do Bash 4.0.

```
$ echo {1..A}      # Letra e número não funfa  
{1..A}  
$ echo {0..15..3}   # Incremento de 3, só no Bash 4  
0 3 6 9 12 15  
$ echo {G..A..2} # Incremento de 2 decresc, só no Bash 4  
G E C A  
$ echo {000..100..10} # Zeros à esquerda, só no Bash 4  
000 010 020 030 040 050 060 070 080 090 100  
$ eval \>{a..c}.{ok,err}\;  
$ ls ?.*  
a.err a.ok b.err b.ok c.err c.ok
```

A sintaxe deste último exemplo pode parecer rebuscada, mas substitua o `eval` por `echo` e verá que aparece:

```
$ echo \>{a..c}.{ok,err}\;  
>a.ok; >a.err; >b.ok; >b.err; >c.ok; >c.err;
```

Ou seja o comando para o Bash criar os 6 arquivos. A função do `eval` é executar este comando que foi montado. O mesmo pode ser feito da seguinte maneira:

```
$ touch {a..z}.{ok,err}
```

Mas no primeiro caso, usamos Bash puro, o que torna esta forma pelo menos 100 vezes mais rápida que a segunda que usa um comando externo (`touch`).

## 5. Colunador

---

É muito raro vermos um *script Shell* que trabalhe devidamente com as opções de **stdin**, **stdout** e **stderr**, ou seja, entrada primária, saída primária e saída de erros primária. O *script* a seguir, habilita a captura dos dados de entrada pelo *pipe* (**|**), por um redirecionamento da entrada (**<**) ou por passagem de parâmetro e manda os erros para a saída de erros padrão, o que permite redirecionar os erros para onde você desejar, ou até trancar a saída de erro (**2>&-**) para evitar essas mensagens.

```
$ cat colunador.sh
#!/bin/bash
# Recebe dados via pipe, arquivo ou passagem de parâmetros e
#+ os coloca em coluna numerando-os

if [[ -t 0 ]]          # Testa se tem dado em stdin
then
    (($# == 0)) || { # Testa se tem parâmetro
        echo Passe os dados via pipe, arquivo ou passagem de
parâmetros >&2 # Redirecionando para stderr
        exit 1
    }
    Params="$@"
else
    Params=$(cat -)    # Seta parâmetros com conteúdo de stdin
fi
set $Params
for ((i=1; i<="$#"; i++))
{
    Lista=$(for ((i=1; i<="$#"; i++)); { printf "%0${##}i %s\n" $i "${!
i}"; } )
}
echo "$Lista" | column -c $(tput cols)
$ echo {A..Z} | colunador.sh
01 A    05 E    09 I    12 L    15 O    18 R    21 U    24 X
02 B    06 F    10 J    13 M    16 P    19 S    22 V    25 Y
03 C    07 G    11 K    14 N    17 Q    20 T    23 W    26 Z
04 D    08 H
```

Esse *script* começa testando se o **FD 0** (zero), que é o descritor da entrada primária, está aberto em um terminal e isso ocorrerá se o programa não estiver recebendo dados por um *pipe* (**|**) ou por um redirecionamento de entrada (**<**). Assim sendo vamos testar se a quantidade de parâmetros passados (**\$#**) é zero, ou seja: os dados não vieram por **stdin** nem por passagem de parâmetros quando então é indicado um erro, que é desviado para **stderr** (**>&2**).



Bem, agora vamos tratar os dados que recebemos, passando-os para a variável `$Parms`. Caso eles tenham vindo por passagem de parâmetros, aí é simples, é só atribuir a `$Parms` o valor de todos os parâmetros passados (`$@`).

Caso os dados tenham vindo de *stdin*, temos mais 2 macetes:

1. A quase totalidade dos utilitários do *Shell*, aceitam o hífen como representando os dados recebido de *stdin*. Veja isso:

```
$ cat arq
Isto é um teste
Outra linha de teste
$ echo Inserindo uma linha antes de arq | cat - arq
Inserindo uma linha antes de arq
Isto é um teste
Outra linha de teste
```

Ou seja, o comando primeiramente listou o que recebeu da entrada primária e em seguida o arquivo `arq`.

2. O outro macete é o comando `set`. Para entendê-lo vá para o *prompt* e faça:

```
$ set a b c
$ echo $1:$2:$3
a:b:c
```

Conforme você viu, este comando atribuiu os valores passados como os parâmetros posicionais.

Uma vez entendido isso, vimos que os valores recebidos de *stdin* foram colocados em `$Parms` que por sua vez teve seu conteúdo passado para os parâmetros posicionais, unificando dessa forma, os dados de qualquer tipo de entrada recebida.

O `for` terminará quando alcançar a quantidade de parâmetros passados (`$#`) e o comando `printf`, em seu primeiro parâmetro, (`%0${##}d`), diz que a linha será formatada (%) com um decimal (d) com `${##}` algarismos, preenchido com zeros à esquerda (0).

Veja isso para entender melhor: `${#var}` devolve o tamanho da variável `var` e também sabemos que `$#` retorna a quantidade de parâmetros passados. Juntando os dois, vemos que `${##}` devolve quantos algarismos têm a quantidade de parâmetros. Faça este teste para você ver:

```
$ set {a..k}          # Passando 11 parâmetros
$ echo ${##} Algarismos
2 Algarismos
$ set a b c           # Passando 3 parâmetros
$ echo ${##} Algarismos
1 Algarismos
```

Uma vez montada a lista, é só passá-la para o comando `tr`, cuja opção `-T` diz que a formatação não é para impressora e o `-8` é para dividir a saída em `8` colunas (trabalhar para que? O *Shell* já tem comando pronto para tudo!)

## 6. Criando Animações com ImageMagick

---

O pacote ImageMagick é extremamente poderoso e possui funcionalidades que muitos desconhecem. O script `entorta.sh`, cria uma animação simples, que pode ser visualizada em qualquer browser web.

### script `entorta.sh`

```
# cat entorta.sh
#!/bin/bash
# Montando uma animação no ImageMagick

# Vou fazer uma figura que servirá como base da animação.
#+ Ela será composta por 1 quadrado azul com 2 retângulos
#+ inscritos, formando a figura base.png
convert -size 150x150 xc:blue \
    -fill yellow -draw 'rectangle 5,5 145,72.5' \
    -fill yellow -draw 'rectangle 5,77.5 145,145' base.png

for ((i=1; i<=40; i++))
{
    # Gero 40 imagens de trabalho, torcendo (swirl)
    #+ a imagem base.png com incrementos de 35 graus
    convert -swirl $((35*i)) base.png Trab_$i.png

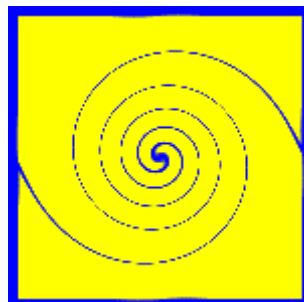
    # Concateno o nome de todas as imagens em Arqs

    Arqs="$Arqs Trab_$i.png"
}

# A animação a seguir é garantida pela opção -coalesce.
#+ A opção -dither é usada para diminuir a perda de
#+ qualidade com a redução da qtd de cores.
#+ A opção -colors 32 reduz a qtd de cores.
#+ A opção -layers optimize, usada com a anterior
#+ visam acelerar o processo.

convert -coalesce -dither -colors 32 -layers optimize $Arqs Anim.gif
# Agora, se vc abrir Anim.gif no browser, verá a animação.
```

A figura abaixo é o resultado final do processo:



O *Zenity* tem um sucessor, que evoluiu demais e atualmente é o que uso para criar interfaces gráficas para *scripts* feitos em *Shell*. Essa interface chama-se *Yad* e é super poderosa, fazendo praticamente tudo o que precisamos, e ao contrário do *Zenity*, não precisa criar uma caixa de diálogos para cada entrada ou saída do programa, já que uma de suas opções é para criar formulários com grau de complexidade bastante elevado.

Aconselho a todos que gostam de programar em *Shell* que instalem esse utilitário e divirtam-se.

O texto abaixo algumas vezes pode parecer fora de contexto, mas é porque ele foi copiado da parte referente ao *Yad* do meu livro **Bombando o Shell**.

## 7. notify-send

---

Essa dica é para aqueles que, como eu, conhecem *Zenity* e notaram que o diálogo `--notification` do YAD não possui a ferramenta `message`. Como acho este acessório do diálogo `--notification` muito útil, mandei um e-mail para o autor do YAD, que me disse que não implementaria esta facilidade pois ela já é contemplada pelo comando `notify-send` e que necessita da `libnotify` e ele queria que a única dependência do YAD fosse o GTK2.

Em para suprir esta falta, vou me afastar um pouco do objetivo e dar uma explicação rapidinha sobre a sintaxe do `notify-send`, que caso não esteja instalado no seu computador, você pode fazê-lo com a seguinte linha (para Debian e seus correlatos):

```
$ sudo apt-get install libnotify-bin
```

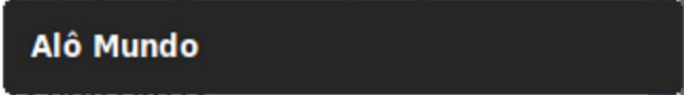
Bem a sintaxe é a seguinte:

```
notify-send [OPCOES...] <TITULO> [TEXT0]
```

Um exemplo bem bobo passando somente o título `TITULO`:

```
$ notify-send "Alô Mundo"
```

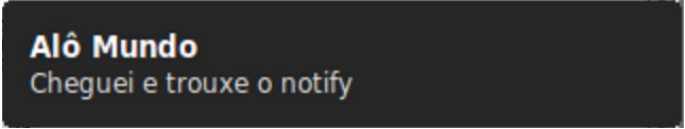
Fazendo isso aparecerá no canto superior esquerdo da tela a seguinte figura:



Alô Mundo

Mas poderíamos também preencher o `TEXTO`:

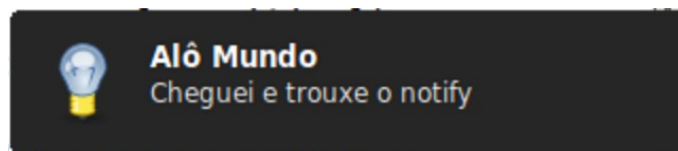
```
$ notify-send "Alô Mundo" "Cheguei e trouxe o notify"
```



Alô Mundo  
Cheguei e trouxe o notify

Podemos também colocar um ícone:

```
$ notify-send "Alô Mundo" "Cheguei e trouxe o notify" \  
-i gtk-dialog-info
```



O **notify-send** também pode ser usado para te avisar o fim de um *job* muito demorado. Para isso basta fazer um *script* que seja mais ou menos assim:

```
$ cat dispara_job.sh
#!/bin/bash
#  Avisa o término de um job, usando notification-send .
#  Esse script deverá ser executado em background
eval "$1"
notify-send -i terminal "Acabou" "\nFim do job $1"
```

Para testá-lo use algo como:

```
$ dispara_job.sh 'time find / > /dev/null 2>&1' &
```

Assim você lerá em *background* (**&**) o nome de todos os arquivos de seu computador mandando a saída primária (**stdout**) para o buraco negro (**> /dev/null**) e a saída de erros (**stderr**) também (**2>&1**). Ele também funciona em *foreground*, mas aí não te traria vantagem alguma, pois seu terminal ficaria preso até que ele terminasse.



Além dos ícones citados na definição do uso de **-button**, o **notify** (e o yad também) aceita(m) os ícones na tabela a seguir, mas note que esses nomes devem ser precedidos pela cadeia **notification-**. Por exemplo: não use **-i audio-next**, isso não funciona. Use **-i notification-audio-next**, da mesma forma **-i audio-play** deve ser substituído por **-i notification-audio-play**. Essa simplificação foi feita somente para tornar os nomes menores, permitindo que sejam tabulados em dois por linha, de forma a tomar menos espaço no artigo.

Inserir notification-	Inserir notification-
audio-next	audio-play
audio-previous	audio-volume-high
audio-volume-low	audio-volume-medium
audio-volume-muted	audio-volume-off
battery-low	device-eject
device-firewire	display-brightness-full
display-brightness-high	display-brightness-low
display-brightness-medium	display-brightness-off
GSM-3G-full	GSM-3G-high
GSM-3G-low	GSM-3G-medium
GSM-3G-none	GSM-disconnected
GSM-EDGE-full	GSM-EDGE-high
GSM-EDGE-low	GSM-EDGE-medium
GSM-EDGE-none	GSM-full
GSM-H-full	GSM-H-high
GSM-high	GSM-H-low
GSM-H-medium	GSM-H-none
GSM-low	GSM-medium
GSM-none	keyboard-brightness-full
keyboard-brightness-high	keyboard-brightness-low
keyboard-brightness-medium	keyboard-brightness-off
message-email	message-IM
network-ethernet-connected	network-ethernet-disconnected
network-wireless-disconnected	network-wireless-full
network-wireless-high	network-wireless-low
network-wireless-medium	network-wireless-none
power-disconnected	

Se quiser, ainda é possível temporizar, mas nesse caso, tenho uma observação a fazer:

Usando o parâmetro de urgência (**-u**) como **critical**, o diálogo só sai da tela quando clicado;

Experimente fazer:

```
$ notify-send "Alô Mundo" "\nCheguei e trouxe o notify" \
-i gtk-dialog-info -u critical
```

Ainda existem outras opções que não exploramos, por estarem saindo do propósito desta publicação. Abaixo uma tabela com essas opções, que espero que você explore-as, são que este utilitário é muito útil, podendo inclusive passar avisos sobre eventos tais como alterações no volume, chegada de e-mail, *instant messages*, eventos de rede,... Você encontrará todos os detalhes em <http://www.galago-project.org/specs/notification/>.

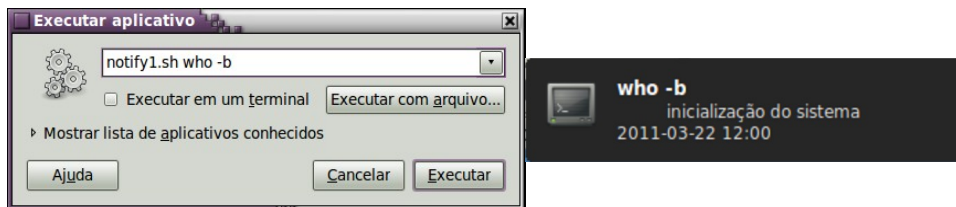
Opções	Ações
<b>-u NIVEL</b>	Especifica o nível de urgência ( <b>low</b> , <b>normal</b> , <b>critical</b> )
<b>-c TIPO[,TIPO...]</b>	Especifica a categoria da notificação
<b>-h TIPO:NOME:VALOR</b>	Especifica base de dados extra para passar. Os tipos válidos são <b>int</b> , <b>double</b> , <b>string</b> e <b>bytes</b>

```
$ cat notify1.sh
#!/bin/bash
# Executa um comando dando a resposta num
#+ balão do tipo notify-send. Ideal para
#+ usar com <ALT>+F2, sem abrir um terminal
#+ Thanks Karthik

Saida=$(eval "$*" 2>/dev/null) || {
    yad --text "\"Comando \"$*\" errado\" --button gtk-ok
    exit 1
}
notify-send -t $((1000+300*$(echo -n $Saida | wc -w))) -u low -i gtk-
dialog-info "$*" "$Saida"
```

Se você não quiser abrir um terminal para executar um comando, basta você usar este programa. Como fazê-lo? É simples: após colocar este *script* no seu diretório *home*, pressione **<ALT>+F2** que aparecerá a caixa "**Executar aplicativo**". Nesta caixa digite o nome do *script* (no caso **notify1.sh**) seguido do comando que você deseja executar e que terá sua saída apresentada em um menu *pop-up* gerado pelo **notify-send**. Vejamos sua utilização para o comando **who -b** que informa a data e hora do último *boot*.





Um programa para você verificar a carga da bateria de seu *notebook*.

```
$ cat notify2.sh
#!/bin/bash
# Para pegar a carga da bateria
Carga=$(acpi -b | sed 's/.* //;s/%//')
# Criando o índice do ícone já que o nome do
#+ ícone é no formato notification-battery-xxx
#+ e xxx deve ser 000, 020, 040, 060, 080,
#+ ou 100, dependendo da carga da bateria
Nivel=$(case $Carga in
    ([01][0-9]) echo 000;;
    ([23][0-9]) echo 020;;
    ([45][0-9]) echo 040;;
    ([67][0-9]) echo 060;;
    ([89][0-9]) echo 080;;
    (*) echo 100;;
esac)
notify-send -i notification-battery-$Nivel "Bateria" \
    "\n0 nível de carga da bateria está em $Carga%"
```

Veja como é a saída do **acpi -b**, que é quem me informa a carga remanescente na bateria:

```
$ acpi -b
Battery 0: Unknown, 98%
```

O **sed** foi usado para limpar essa saída, deixando somente o **98**.

Experimente fazer um *script* para pegar a temperatura de seu *notebook*. Para fazer isso, use o seguinte comando:

```
$ acpi -t
Thermal 0: ok, 29.8 degrees C
Thermal 1: ok, 46.0 degrees C
```

Veja o *script* que fiz para mudar o estado do *touchpad* do meu *notebook*:

```
$ cat tp-on-off
#!/bin/bash
# O meu note não tem hot key para Habilitar/Desabilitar
#+ o touchpad, então eu fiz esse bacalho

if [ $(synclient | sed '/touchpad/I !d; s/^\.*= //') -eq 0 ]
then
    notify-send -i touchpad-disabled.svg "TouchPad OFF" \
```

```

        '<b><big>Desliguei</big></b>\to touchpad'
    synclient TouchpadOff=1
else
    notify-send -i input-touchpad "TouchPad ON" \
        '<b><big>Liguei</big></b>\to touchpad'
    synclient TouchpadOff=0
fi

```

O comando **sed** só não deleta (**!d**) a linha que possui a cadeia *touchpad*, ignorando a caixa das letras (**I**). Veja:

```

$ synclient | sed '/touchpad/I!d'
    TouchpadOff          = 0

```

e ainda no **sed**, foi substituído (**s**) tudo que existia antes do estado do *touchpad* (**s/^.\*=**  
**/**) por vazio (**//**), restando desta forma somente o seu estado atual (que no exemplo dado era zero (**0**))

## 8. YAD - Yet Another Dialog

---

Sou fã de carteirinha do *Shell* e frequentemente preciso melhorar a interface de um *script* com o usuário. Para isso, invariavelmente usava o *Zenity* que é extremamente fácil e melhora muito a apresentação.

Minha satisfação como o *Zenity* só não era total porque todos que conhecem um pouquinho de GTK 2, sabem que se pode tirar muito mais dessa linguagem do que o *Zenity* tira. Sempre esperei que a qualquer momento o *gnome* lançaria uma nova versão deste software que seria arrasadora, viria com todos os incrementos que eu (e acho que todos) aguardavam.

Estava enganado, acompanhei o lançamento de diversas revisões do software mas elas simplesmente tratavam *bugs* e apresentação. Infelizmente nada de inovação. Porém um dia descobri o YAD.

YAD significa *Yet Another Dialog*, mas sua tradução do russo é "veneno" (e essa é a inspiração de seu ícone) e se define como um *fork* do *Zenity*. Porém o YAD, distancia-se um pouco dele por não utilizar bibliotecas descontinuadas como *libglade* ou *gnome-canvas*. Utiliza somente o GTK 2, o que lhe dá uma maior portabilidade dentro do ambiente \*n?x.

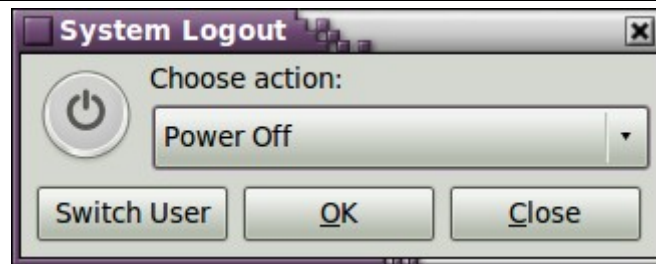
Poderia defini-lo como um *Zenity* com muitos aperfeiçoamentos, como botões customizados, diálogos adicionais, menu *pop-up* no ícone do **notification** (sim é o mesmo **notification** do *Zenity*, porém mais incrementado), diálogo para montar formulários complexos e muito mais.

Este artigo não é para mostrar como se instala, isso eu sei que vocês sabem se virar. O que quero é mostrar algumas (infelizmente muito poucas) novidades que o YAD traz para trabalharmos com interface GUI em *Shell*. Vamos ao que interessa:

Este é um dos poucos exemplos que achei pela internet e está no [wiki do YAD](#). Lá ele coloca o código completo para dar *shutdown*, *reboot*, ... Mas aqui executaremos somente a parte do YAD direto na linha de comandos.

```
$ yad --width 300 --entry --title "System Logout" \
```

```
--image=gnome-shutdown \
--button="_Switch User:2" \
--button="gtk-ok:0" --button="gtk-close:1" \
--text "Choose action:" \
--entry-text \
"Power Off" "Reboot" "Suspend" "Logout"
```



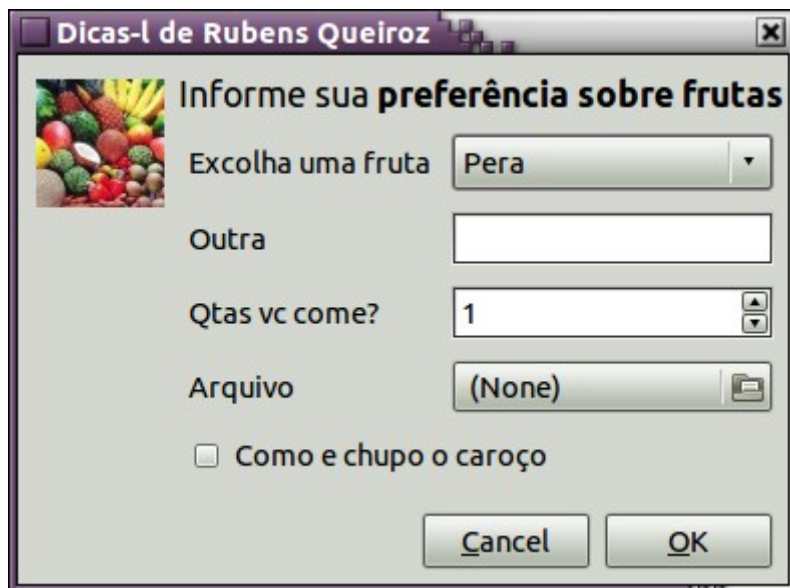
Aqui você pode ver uma ComboBox, diversos botões (um deles *customizado*) e um código extremamente simples. Repare que após a declaração do nome de cada botão, tem um `:N`, onde `N` é o valor que retorna em `$?` quando este botão for clicado. Simples, não?

Um outro exemplo usando o diálogo `--form`:

```
$ yad --form --title "Dicas-l de Rubens Queiroz" \
--text "Informe sua preferência sobre frutas" \
--image Frutas.jpg \
--field "Excolha uma fruta:CB" \
--field Outra \
--field "Qtas vc come?:NUM" \
--field "Arquivo:FL" \
--field "Como e chupo o caroço:CHK" \
'Pera!Uva!Maçã!Manga' \
"" \
'1!10!1'
```

Com certeza você sabe que as contrabarras foram colocadas somente para tornar o texto mais visível e fácil de comentar, mas poderia ter sido escrito em somente uma linha.

O diálogo `--form` permite que eu use uma opção `--field` que aceita um monte de modificadores. Aqui usei, na ordem, `:CB`, nenhum, `:NUM`, `:FL` e `:CHK` que são, respectivamente, *ComboBox*, Entrada de texto, *SpinBox* numérico, Seletor de arquivos e uma *CheckBox* e para finalizar passei os valores das 3 primeiras. Veja o que gerou:



Ainda quero fazer uns comentários:

- Repare no título (`--text`) que usei *tags* de formatação;
- Veja a facilidade de colocar qualquer imagem no diálogo usando a opção `--image`;
- Especifiquei os textos do `ComboBox` separando-os com um ponto de exclamação (!). Este é o *default*, mas pode ser alterado;
- O campo `Outra`, por não ter tido nenhum valor especificado assume que é uma simples entrada de dados.
- O campo `Qtas vc come?` É um *SpinBox* e seu valor foi especificado com `Mínimo!Máximo!Incremento`.
- Quando o campo `Arquivo` for clicado, abrirá um super diálogo para seleção de arquivo;
- O campo `Arquivo` e o `CheckBox` não foram inicializados por serem os últimos e por isso poderem assumir seus valores *default* (`Nome` e `FALSE`, respectivamente).

Fora esses pequenos exemplos ainda existem muitos outros diálogos que não foram implementados no *Zenity* como diálogos *drag'ndrop*, diálogos com ícones para escolha de

aplicativos, dialogo para escolha de fonte, diálogo para escolha de cores. E mesmo muitos dos diálogos existentes no *Zenity*, no *YAD* passam a ter mais opções.

Para finalizar, experimente fazer:

```
$ ls -l | yad --text-info --fontname "mono 10" --width 600 --height 300
$ ls -l | zenity --text-info --width 600 --height 300
```

e veja a falta que a opção `--fontname` faz no *Zenity*.

Só para clarear o que falei, existe um diálogo `--fonte` para escolha de fontes e uma opção `--fontname` para especificar qual fonte um determinado diálogo usará. Agora, para finalizar mesmo vou mostrar o exemplo anterior, porém com escolha da fonte:

```
#!/bin/bash
# Exemplo de uso dos diálogos --font
#+ e --text-info

Fonte=$(yad --font --title "Escolha de fontes" \
  --text "Escolha uma fonte
Dê preferência a fontes monoespacejadas" \
  --height=300 --width 600) || exit 1

ls -l | eval yad --text-info \
  --width=600 --height 300 \
  --title "\"Listagem do diretório $PWD\"" \
  --fontname \"'$Fonte'\"
```

Espero que você instale o *YAD* e teste esses exemplos que dei. Tenho certeza que você se amarrará no software.

## 9. Fatiando opções com o getopt

---

Analisar uma linha de comando e quebrá-la em cada uma de suas opções é uma tarefa complicada, pois, só falando em duas opções (**-A** e **-B**) sendo que **-B** pode ou não ter argumentos (**ARG**), teríamos de pesquisar por:

-AB	-ABARG	-AB ARG
-A -B	-A -BARG	-a -B ARG
-BA	--	--
-B -A	-BARG -A	-B ARG -A

Isso sem falar que, se uma das opções for facultativa, a gama da análise (provavelmente em um comando **case**) seria o dobro dessa. Em virtude dessa complexidade e para aliviar a vida do programador, foi desenvolvido o nosso amigo **getopts**, que, por ser um intrínseco (*builtin*), é bastante veloz. Ele foi feito para suceder a antiga implementação que chamava-se **getopt** (sem o **s**), que não era intrínseco e tinha alguns bugs, mas seu único pecado é não aceitar opções longas dos comandos, previstas pelo GNU (como em **CMD --help**, p.ex.).

Só para melhorar o entendimento unificando a terminologia que será usada, pelos parágrafos anteriores você já deve ter reparado que chamamos de opção uma letra precedida de um sinal de menos ou traço (**-**) e argumentos são os textos requeridos por algumas opções ou simplesmente passados para a linha de comando.

### Exemplo:

```
cut -f 2 -s -d : /etc/passwd
```

Trecho	Terminologia
<b>cut</b>	Programa
<b>-f -s -d</b>	Opções
<b>2</b>	Argumento da opção -f
<b>:</b>	Argumento da opção -d
<b>/etc/passwd</b>	Argumento do programa

## Sintaxe:

```
getopts CADOPTS VAR [ARGS]
```

Onde:

<b>CADOPTS</b>	contém a cadeia de opções e suas eventuais necessidades de argumento. Quando uma opção aceita um argumento, ela deverá vir seguida de um dois pontos (:). Como veremos adiante esta variável receberá um ponto de interrogação (?) em caso de erro. Assim sendo, o ponto de interrogação (?) e os dois pontos (:) são especiais para o <b>getopts</b> e em virtude disso esses dois caracteres não podem ser usados como opções.
<b>VAR</b>	O comando terá de ser executado (normalmente dentro de um <i>loop</i> de <b>while</b> ) tantas vezes quantas foram as opções válidas. A variável <b>VAR</b> , receberá em cada passada a opção (tirada de <b>CADOPTS</b> ) que será analisada, ou terá uma sinalização (flag) de erro.
<b>ARGS</b>	Normalmente são pesquisadas as opções passadas em <b>VAR</b> , mas se os argumentos forem passados em <b>ARGS</b> , eles é que serão analisados, ou seja, a existência desse parâmetro diz ao <b>getopts</b> para analisar o conteúdo de <b>ARGS</b> em vez dos parâmetros posicionais.

Variáveis usadas pelo **getopts**:

<b>OPTARG</b>	Contém cada argumento das opções descobertas em cada volta do loop do <b>getopts</b> ou uma marca de erro que analisaremos à frente;
<b>OPTIND</b>	Contém o índice do parâmetro posicional em análise que você gerenciará para saber, pelo valor de <b>OPTIND</b> a opção que está sendo trabalhada;
<b>OPTERR</b>	É uma variável que quando tem valor 1 indica para Shell que os erros irão para a tela e este é o padrão pois é sempre inicializada com valor 1. Com valor zero, os erros são assinalados, mas não são exibidos.

### 9.1. Manipulação de erros

Como já vimos o **getopts** pode ser executado de dois modos:

Modo silencioso	Quando a variável do sistema <b>\$OPTERR</b> for zero ou quando a cadeia que você passou em <b>OPTARG</b> começa por dois pontos (:);
Modo falador	Este é o normal. <b>OPTERR</b> é igual a um e a cadeia passada em <b>OPTARG</b> não começa por dois pontos (:).

Se for achada uma opção inválida, **getopts** põe um ponto de interrogação (?) em **VAR**. Se não estiver em modo silencioso dá uma mensagem de erro e esvazia **OPTARG**. Se estiver silencioso, o caractere que gerou a opção inválida é colocado em **OPTARG**.

Se um argumento requerido não for informado e **getopts** não estiver em modo silencioso, um ponto de interrogação (?) é colocado em **VAR**, **OPTARG** é esvaziada e é dada uma mensagem de



erro. Caso esteja em modo silencioso, um dois pontos (:) é colocado em VAR e OPTARG recebe o caractere da opção da qual não foi informado o argumento requerido

Quando o assunto é programação em bash, aconselho que, em seus scripts, sempre use o modo silencioso e monitore eventuais erros.

Como esse cara funciona?

Vamos fazer essa análise por exemplos, que é muito mais simples de aprender:

## Exemplo:

Suponha que um programa possa receber a opção -C. O script, para análise das opções deveria ser do tipo:

```
$ cat cut1.sh
#!/bin/bash

while getopts c Opc
do
    case $Opc in
        c) echo Recebi a opção -c
           ;;
    esac
done
```

Como vamos trabalhar diversos exemplos em cima deste mesmo script, preferi já começá-lo com um case, quando nesse caso bastaria um if.

Vamos vê-lo funcionando:

1. Executando-o sem nenhuma opção, ele não produz nenhuma saída;
2. Usando a opção -c, que é o esperado:

```
$ cut1.sh -c
```

Recebi a opção -C

3. Usando a opção -Z, que é inválida:

```
$ cut1.sh -z
./cut1.sh: opção ilegal -- z
```

Xiii, quem deu a mensagem foi o *Shell* e eu não soube de nada. Mesmo com erro, o programa continuaria em execução, porque o erro não é tratado pelo programa.

Então vamos alterá-lo colocando-o em modo silencioso, o que, como já vimos, pode ser feito iniciando a cadeia de opções (**CADOPTS**) com um dois pontos ou atribuindo zero à variável **\$OPTERR**.

```
$ cat cut2.sh
#!/bin/bash

while getopts :c Opc
do
    case $Opc in
        c) echo Recebi a opção -c
           ;;
        \?) echo Caractere $OPTARG inválido
           exit 1
    esac
done
```

Agora coloquei um dois pontos à frente de **CADOPTS** e passei a monitorar um ponto de interrogação (?) no **case**, porque quando é achado um caractere inválido, o **getopts**, como já vimos, coloca um ponto de interrogação (?) na variável **\$Opc**. Desta forma, listei **\$OPTARG** e em seguida dei **exit**.

Note que antes da interrogação coloquei uma contrabarra (\), para que o Shell não o expandisse para todos os arquivos que contém somente um caractere no nome.

4. Agora que já tenho o ambiente sob meu controle, vamos executá-lo novamente com uma opção não prevista:

```
$ cut2.sh -z
Caractere z inválido
$ echo $?
1
```

Agora aconteceu o que queríamos: deu a nossa mensagem e o programa abortou, passando 1 como código de retorno (**\$?**).

Bem, já examinamos todas as possibilidades que a passagem de opções pode ter. Vamos agora esmiuçar o caso que uma opção que tenha parâmetro associado. Para dizer que uma opção pode ter um argumento, basta colocar um dois pontos (:) após a letra da opção.

Então, ainda evoluindo o programa de teste que estamos fazendo vamos supor que a opção -c requeresse um argumento. Deveríamos então fazer algo como:

```
$ cat cut3.sh
#!/bin/bash
while getopts :c: Opc
do
    case $Opc in
        c)  echo Recebi a opção -c
            echo Parâmetro passado para a opção -c foi $OPTARG
            ;;
        \?) echo Caractere $OPTARG inválido
            exit 1
            ;;
        :)  echo -c precisa de um argumento
            exit 1
    esac
done
```

Agora introduzimos o caractere dois pontos (:) no case porque, como já vimos, quando um parâmetro não é localizado, o `getopts` em modo silencioso coloca um dois pontos (:) em `$Opc`, caso contrário, a falta de argumento é sinalizada, com um ponto de interrogação (?) nesta mesma variável.

Vamos então analisar todas as possibilidades:

1. Executando-o sem nenhuma opção, ele não produz nenhuma saída;
2. Passando a opção -C acompanhada de seu parâmetro, que é o esperado:

```
$ cut3.sh -c 2-5
Recebi a opção -c
Parâmetro passado para a opção -c foi 2-5
```

3. Passando a opção correta, porém omitindo o parâmetro requerido:

```
$ cut3.sh -c
-c precisa de um argumento
$ echo $?
1
```

Para finalizar esta série, vejamos um caso interessante. Desde o início, venho simulando nesse exemplo a sintaxe do comando `cut` com a opção `-c`. Para ficar igualzinho à sintaxe do `cut`, só falta receber o nome do arquivo. Vejamos como fazê-lo:

```
$ cat cut4.sh
#!/bin/bash

# Inicializar OPTIND é necessário caso o script tenha
#+ usado getopt antes. OPTIND mantém seu valor
OPTIND=1
while getopt :c: Opc
do
    case $Opc in
        c) echo Recebi a opção -c
            echo Parâmetro passado para a opção -c foi $OPTARG
            ;;
        \?) echo Caractere $OPTARG inválido
            exit 1
            ;;
        :) echo -c precisa de um argumento
            exit 1
    esac
    shift $((--OPTIND))
    Args="$@"
    echo "Recebi o(s) seguinte(s) argumento(s) extra(s): $Args"
done
```

E executando-o vem:

```
$ cut4.sh -c 2-5 /caminho/do/arquivo
```

Recebi a opção `-c`

Parâmetro passado para a opção `-c` foi `2-5`

Recebi o(s) seguinte(s) argumento(s) extra(s): `/caminho/do/arquivo`

Agora vamos dar um mergulho num exemplo um bem completo e analisá-lo. Para esse script interessam somente as opções `-f`, `-uargumento` e `-C`. Veja o código (mas este ainda não está 100%).

```
#!/bin/bash
printf "%29s%10s%10s%10s%10s\n" Comentário Passada Char OPTARG OPTIND
while getopt ":fu:C" VAR
do
    case $VAR in
        f) Coment="Achei a opção -f"
            ;;
```

```

    u) Coment="Achei a opção -u $OPTARG"
    ;;
    C) Coment="Achei a opção -C"
    ;;
    \?) Coment="Achei uma opção inválida -$OPTARG"
    ;;
    :) Coment="Faltou argumento da opção -u"
esac
printf "%30s%10s%10s%10s%10s\n" "$Coment" $((++i)) "$VAR" "$OPTARG"
"$OPTARG"
done

```

Agora vejamos a sua execução passando todas as opções juntas e sem passar o parâmetro requerido pela opção **-u** (repare os dois pontos (:)) que seguem o u na chamada do **getopts** neste exemplo).

```
$ getop.sh -fCxu
```

Comentário	Passada	Char	OPTARG	OPTIND
Achei a opção -f	1	f		1
Achei a opção -C	2	C		1
Achei uma opção inválida -x	3	?	x	1
Faltou argumento da opção -u	4	:	u	2

Repare que a variável **\$OPTIND** não foi incrementada. Vejamos então o mesmo exemplo, porém passando as opções separadas:

```
$ getop.sh -f -C -x -u
```

Comentário	Passada	Char	OPTARG	OPTIND
Achei a opção -f	1	f		2
Achei a opção -C	2	C		3
Achei uma opção inválida -x	3	?	x	4
Faltou argumento da opção -u	4	:	u	5

Ah, agora sim! **\$OPTIND** passou a ser incrementado. Para fechar, vejamos um exemplo sem opção inválida e no qual passamos o parâmetro requerido pela opção **-u**:

```
$ getop.sh -f -C -u Util
```

Comentário	Passada	Char	OPTARG	OPTIND
Achei a opção -f	1	f		2
Achei a opção -C	2	C		3
Achei a opção -u Util	3	u	Util	5

## Algumas observações:

1. Vimos que mesmo após encontrar erro, o **getopts** continuou analisando as opções, isso se dá porque o que foi encontrado pode ser um parâmetro de uma opção e não um erro;
2. Então como distinguir um erro de um parâmetro? Fácil: se a opção em análise requer argumento, o conteúdo de **\$OPTARG** é ele, caso contrário é um erro;
3. Quando encontramos um erro devemos encerrar o programa pois o **getopts** só aborta sua execução quando:
  1. Encontra um parâmetro que não começa por menos (-);
  2. Quando encontra um erro (como uma opção não reconhecida).

O parâmetro especial **--** marca o fim das opções, mas isso é uma convenção para todos os comandos que interagem com o Shell.

Então a nossa versão final do programa seria:

```
$ cat getop.sh
#!/bin/bash
function Uso
{
    echo "    $Coment
    Uso: $0 -f -C -u parâmetro" >&2
    exit 1
}
(($#==0)) && { Coment="Faltou parâmetro"; Uso; }

printf "%29s%10s%10s%10s%10s\n" Comentário Passada Char OPTARG OPTIND
while getopts ":fu:C" VAR
do
    case $VAR in
        f) Coment="Achei a opção -f"
           ;;
        u) Coment="Achei a opção -u $OPTARG"
           ;;
        C) Coment="Achei a opção -C"
           ;;
        \?) Coment="Achei uma opção invalida -$OPTARG"
            Uso
            ;;
        :) Coment="Faltou argumento da opção -u"
           ;;
    esac
done
```

```
    printf "%30s%10s%10s%10s%10s\n" "$Coment" $((++i)) "$VAR" \  
"$OPTARG" "$OPTIND"  
done
```