Ficha 2.1 – Introdução á linguagem de script PERL

Tópicos abordados:

- Formato de um ficheiro PERL
- Opções da linha de comandos e output do script
- Variáveis
- Leitura da entrada padrão
- Operadores
- Opções da linha de comandos
- Funções numéricas e de strings
- Estruturas de decisão
- Estruturas de repetição
- Vetores (arrays)
- Execução de um programa externo num script
- Passagem de valores pela linha de comandos
- Listas associativas (hashes)
- Variáveis de ambiente
- Ficheiros
- Exercícios

O que é um *script?*

Um *script* é um ficheiro de texto que contém uma série de instruções, que se pode executar em linhas de comandos, e que são executadas sequencialmente. Nesse sentido, são como os ficheiros de extensão BAT do MS-DOS, sendo que como nos sistemas UNIX, não existe o conceito de extensão, os ficheiros *scripts* podem ter qualquer nome. Contudo, por forma a rapidamente identificar os ficheiros *script* PERL, esses terão todos a extensão **.pl** (convenção quase universal, no que respeita a ficheiro PERL).

Num ambiente UNIX, é possível obter mais informações sobre PERL ou as suas funções, recorrendo ao utilitário "man":

```
man PERL
```

Como poderá constatar, a versão eletrónica do manual, estende-se por várias secções (e.g. PERL, Perldata, Perlfaq, etc.).

Na Internet, uma das referências mais importantes do PERL é o sítio http://www.PERL.com. Aí é possível, encontrar versões de PERL para várias plataformas (Linux, Windows, MacOS etc.), seja no formato de código fonte, ou no formato binário. Contudo, para o Windows, a versão recomendada do PERL é da ActiveState (http://www.activestate.com).

1. Formato de um ficheiro PERL

1.1. As primeiras linhas

As primeiras linhas de um ficheiro PERL devem ser constituídas por:

```
#!/usr/bin/perl
use strict;
use warnings;
```

Listagem 1: As primeiras linhas de um ficheiro PERL

A primeira linha indica que o ficheiro contém comandos que devem ser executados pelo interpretador de *PERL*, localizado na diretoria */usr/bin*. A segunda linha obriga o interpretador a utilizar as regras mais restritas sobre o código. Esta opção tem por finalidade ajudar o programador a encontrar os erros no seu código de forma mais fácil. A linha "use warnings;" ativa os avisos (*warnings*) do interpretador, levando a que situações menos claras sejam detetadas e indicadas na execução do script. É sempre conveniente ter os avisos ativos, pois muitas vezes um aviso pode alertar o programador para um comportamento que não é o pretendido. Uma alternativa equivalente ao uso da linha "use warnings;" é a indicação da opção "-w" ao interpretador na primeira linha do programa. Assim, como a opção "-w", a primeira linha passa a ser:

```
#!/usr/bin/perl -w
```

Desta forma, o ficheiro poderá ser executado através da invocação do nome do ficheiro na linha de comando, desde que o utilizador que requisite a execução **possua permissão de execução sobre o referido ficheiro**. (as permissões podem ser alteradas através do comando **chmod**).

Exemplo de execução de um *script* perl:

```
user@ubuntu:~$./script.pl
```

Outra forma de executar um script em PERL consiste no lançar do interpretador perl seguido do script a executar,sendo que nesse caso, não é necessário o ficheiro perl ter:

```
user@ubuntu:~$perl ./script.pl
```

1.2. Comentários

Em PERL, os comentários são iniciados pelo caráter # (cardinal) abrangendo o resto da linha.

```
(...)
# comentário
print("Ola!\n"); # Outro comentário
```

Listagem 2: Comentários em PERL

1.3. Expressões

Todas as expressões (*statements*) em PERL acabam com ";" (ponto e vírgula), tal como acontece na linguagem de programação C.

```
#!/usr/bin/perl –w
use strict;  # Força a declaração de variáveis
use warnings;# Ativa os avisos do interpretador de PERL

my $var1;
print("Olá mundo\n");
(…)
```

Listagem 3: Expressões

2. Opções da linha de comandos e *output* da *script*

2.1. Opções da linha de comandos

O interpretador **PERL** pode ser executado de diversos modos. Para se definir o modo de execução utilizam-se os carateres opcionais, a seguir à palavra **PERL**. Para saber as várias opções que podem ser utilizadas execute o comando: perl **–h**

Algumas opções:

-w → apresenta as mensagens de *warnings*. Equivalente ao uso da linha "use warnings;"

-c → verifica a sintaxe sem executar as instruções.

2.2. *Output* da *script*

Sempre que se pretender mostrar ou imprimir alguma informação através do canal padrão de *output*, que por omissão é o ecrã, deve ser utilizada a função *print* ou *printf*.

```
#!/usr/bin/perl –w
print("Olá mundo\n");
printf("Olá mundo\n");
```

Nos casos em que se pretende tratar os erros gerados pela *script*, os mesmos devem ser direcionados para o canal padrão de *output* dos erros, o STDERR. As mensagens direcionadas para esse canal, através da função *print,* serão mostradas no ecrã.

```
#!/usr/bin/perl –w
print STDERR ("Erro: Teste ao output de erros\n");
```

Quando um *script* perl é executado com sucesso, o mesmo devolve por omissão o valor 0 (zero), através da inclusão automática da linha "**`exit(0);`**" no final do programa, o que indica que o mesmo **foi executado com sucesso**. Esta abordagem é igualmente seguida por outros programas. Assim, caso um programa devolva um valor **diferente de zero**, indica ao utilizador que o mesmo **foi executado com insucesso**, sendo o valor devolvido correspondente a um código de erro que foi especificado pelo programador. Assim, para os casos em que um programa termine sem sucesso, o programador deve utilizar a função **`exit`**, seguida do respetivo código de erro definido anteriormente, para que se possa informar o utilizador acerca desse erro.

```
#!/usr/bin/perl -w
use strict;
use warnings;
if ($var == 0){
  print STDERR ("ERRO!");
  exit(1);
}
exit(0);
```

3. Variáveis

3.1. Declaração

Em PERL, todas as variáveis escalares são declaradas com a função *"my"* e o nome destas começa sempre por "$". O caráter a seguir ao "$" deve ser uma letra ou "underscore", os números e outros carateres estão reservados para funções especiais. Tal como acontece em C, os nomes das variáveis são sensíveis a maiúsculas e minúsculas (*case-sensitive*).

Exceto nas variáveis com estruturas (não escalares), em PERL ao declararmos uma variável não indicamos o seu tipo. Consoante as atribuições que fazemos a essa variável o PERL determina automaticamente o tipo da variável. Por isso, uma determinada variável pode conter um número e de seguida uma linha de texto, ou vice-versa. A atribuição de valores é feita com o operador "=", tal como em C.

```
#!/usr/bin/perl
use strict;
use warnings;

my $qualquer_coisa = "linha de texto";
print("$qualquer_coisa \n");

my ($var1, $var2, $var3); # declaração de várias variáveis com ()


my $c = 10 + length($qualquer_coisa);
print("$c \n");
```

Listagem 4: Declaração de variáveis

3.2. Strings

As *string's*, ou cadeias de carateres podem ser *single-quoted* ou *double-quoted*.

```
'single-quoted string'; # distinguem-se pelo
"double-quoted string"; # uso de ' ou de "
```

Listagem 5: *Single* e *double-quoted* strings

3.2.1. Plicas (*single-quoted*)

As strings *single-quoted* garantem que quase todos os carateres são escritos como aparecem ("what you see is what you get"). No entanto, existem 2 carateres especiais: \ e **'.** No exemplo em baixo, podemos ver o uso destes carateres:

```
print('xxx\'xxx');     # aparece no ecran: xxx'xxx
print('xxx\xxx');      # aparece no ecran: xxx\xxx
print('xxx\\xxx');     # aparece no ecran: xxx\\xxx
print('xxx\\\'xxx');   # aparece no ecran: xxx\'xxx
```

Listagem 6: Carateres especiais

Nas strings *single-quoted* também podemos inserir linhas diretamente, como no exemplo, a seguir:

```
print('Uma linha
outra linha');  # aparece no ecrã com mudança de linha
```

Listagem 7: Duas linhas na mesma string

3.2.2. Aspas (*double-quoted*)

As strings *double-quoted* são semelhantes às single-quoted, mas permitem mais funcionalidades. Este tipo de strings funciona da mesma forma que as strings em C, ou seja, existem sequências de carateres com um significado especial:

String	Valor interpolado
\\	O caráter \
\$	O caráter $
\@	O caráter @
\t	Tab
\n	Newline
\r	Hard return
\f	Form feed
\b	Backspace
\a	Alarm (bell)
\e	Escape
\033	Caráter representado pelo valor octal 033
\x1b	Caráter representado pelo valor hexadecimal 1b

Tabela 1: Operadores para números e strings

Seguem-se alguns exemplos:

```
#!/usr/bin/perl
use strict;
use warnings;

print("a backslash: \\ \n");
print("tab follows:\tover here\n");
print("Ring!\a\n");
print("bkm\@ebb.org\n");
print("200\$00");
```

Listagem 8: Exemplos de interpolação

3.3. Números

O PERL trata os números de uma forma bastante simples, não sendo necessário indicar se é um número de vírgula flutuante ou inteiro. Mais, nem sequer é necessário indicar se é um número ou uma string (o PERL interpreta consoante o contexto). Por exemplo, podemos somar a string "14" com 4 e obter 18 como resultado. A única exceção é nas strings com formas parecidas com os números octais "0x234" ou hexadecimais "0123".

```
#!/usr/bin/perl
use strict;
use warnings;

print (2E-4, " ", 9.77e-5, " ", 100.00, " ", 2_000_300, "\n");
my $num='0x432';
print("$num\n");      # escreve 0x432
my $hex=oct $num;     # converte para hexadecimais ou octais
print("$hex\n");      # escreve o equivalente em decimal
```

Listagem 9: Números em PERL

Internamente, todos os números em PERL são armazenados de forma equivalente aos "doubles" do C.

4. Leitura da entrada padrão

Em PERL é possível aceder aos dados da entrada padrão (usualmente teclado, se não tiver sido especificado redireccionamento de entrada) através do descritor especial "<STDIN>".

O exemplo seguinte, realiza a leitura de um número (na realidade é lida uma string, não existindo garantias que seja mesmo um número) a partir da entrada padrão, exibindo-o depois. É prática corrente passar-se o valor lido pela função chomp( ) por forma a que o **\n** lido do STDIN seja removido.

```
#!/usr/bin/perl
use strict;
use warnings;

print("Introduza um número: ");

my $Numero = <STDIN>;   # Lê do teclado para a variável

print("Numero lido é '$Numero'\n");

chomp($Numero);   # Elimina o caráter "\n"

print("Numero lido é '$Numero'\n");
```

Listagem 10: Leitura através da entrada padrão

5. Operadores

5.1. Operadores Aritméticos

Operador	Função	Exemplo	Resultado
+	Soma	3 + 4	7
-	Subtração ou	4 – 2	2
	Negação	-5	-5
*	Multiplicação	5 * 5	25
/	Divisão	15 / 4	3.75
**	Expoente	4**5	1024
%	Resto da divisão inteira	15 % 4	3

Tabela 2: Operadores para números e strings

Todas as operações em PERL resultam em números em vírgula flutuante (i.e. números reais), se desejarmos um resultado num número inteiro podemos empregar o operador **int**, conforme ilustrado de seguida:

```
#!/usr/bin/perl
use strict;
use warnings;

my $resultado= int 15/4;  # int converte para um número inteiro
print("$resultado \n");

my $valor=10/3;
print("$valor \n");          # para formatar a forma como o
printf("%.2f\n",$valor);     # nº é escrito, da mesma maneira
                             # que em C
$valor=sprintf("%.4f\n",12/7);
print($valor);
```

Listagem 11: Vírgula flutuante versus números inteiros

Exercício 1

Fazer o programa "converte.pl", em PERL, para converter graus Fahrenheit para graus Celsius. O programa deve pedir os dados necessários ao utilizador. A fórmula de conversão é a seguinte:

$$^{o}C = \frac{5 \times (^{o}F - 32)}{9}$$

5.2. Operadores de teste e comparação

Os operadores de comparação são usados para testar a relação entre dois números ou duas strings. Existem ainda os operadores lógicos para efetuarmos comparações lógicas (booleanas), que resultam em verdadeiro ou falso.

Em PERL, todas as variáveis escalares (números, strings e referências) são verdadeiras, exceto nas seguintes três situações:

- Uma string vazia, “”
- O número zero (0)
- E o valor “undef”, que resulta quando declaramos uma variável e não lhe atribuímos qualquer valor.

5.2.1. Operadores relacionais e de igualdade

Segue-se uma tabela com os operadores relacionais e de igualdade. Chama-se a atenção para o facto dos operadores para strings **serem diferentes** dos operadores numéricos. Isto deve-se ao facto do PERL converter automaticamente as strings em números e vice-versa. Deste modo, é necessário indicar-lhe em que formato queremos comparar – números ou strings – através do operador apropriado.

Teste	Operador numérico	Operador para strings
Igualdade	==	eq
Diferente	!=	ne
Menor que	<	lt
Maior que	>	gt
Menor ou igual	<=	le
Maior ou igual	>=	ge

Tabela 3: Operadores para comparação de números e strings

```
4<5           # verdadeiro
4<=4          # verdadeiro
4<4           # falso
5<4+5         # verdadeiro
6<10>15       # syntax error

'5' < 10        # verdadeiro
'5' lt 10       # falso, porque em ASCII o ‘1’ aparece antes do ‘5’

'add'<'adder'  # erro, se usarmos a opção –w
'this'=='that' # erro, se usarmos a opção –w, senão é verdadeiro,
               # porque converte as 2 strings para 0 (zero)
'add'lt'adder' # verdadeiro
'add'lt'Add'   # falso, maiúsculas e minúsculas são diferentes
'add'eq'add '  # falso, os espaços também contam
```

Listagem 12: Comparações e os seus resultados

5.2.2. Operadores Lógicos

Os operadores lógicos permitem-nos efetuar operações com base nas regras da álgebra Booleana. Por exemplo, "x AND y" é verdadeiro se "x" e "y" forem verdadeiros. Segue-se uma tabela com os operadores lógicos. Como se pode observar existem dois conjuntos de operadores, um "emprestado" do C e o outro à "moda" do PERL:

À moda do C	À moda do PERL	Significado
&&	and	"E" lógico
||	or	"OU" lógico
!	not	Negação

Tabela 4: Operadores lógicos C versus PERL

A única diferença entre os dois estilos é que os operadores "à moda" do PERL têm uma precedência mais baixa que os operadores "à moda" do C. Isto pode traduzir-se no facto de nos operadores em PERL se evitar uns parêntesis. Contudo, recomenda-se que se use sempre parêntesis de modo a explicitar as prioridades, melhorando também a legibilidade do código.

<=> → retorna -1, 0 ou 1 conforme o operando esquerdo é numericamente menor, igual ou maior que o operando direito
cmp → é equivalente a <=> mas faz a comparação dos operandos em modo de string
gt, ge, lt, le → correspondem aos operadores <, <=, >, >= respetivamente mas tratando os operandos como strings
eq, ne → correspondem aos operadores == e != mas tratando os operandos como strings
or, and, xor → correspondem na operação efectuada aos operadores ||, && e ^ respetivamente mas com menor precedência
. → concatenação de strings
.. → devolve uma lista cujos elementos estão compreendidos entre os operadores (por exemplo, @A=1..3; é equivalente a @A = (1,2,3);)

Tabela 5: Quadro-resumo de operadores

5.3. Alguns Operadores Específicos do PERL

5.3.1. Operador <=>

$a <=> $b: compara dois números devolvendo:

-1 se $a < $b
0 se $a == $b
1 se $a > $b

```
#!/usr/bin/perl
use strict;
use warnings;

my($a);
my($b);
$a = 12;
$b = 23;
print ($a <=> $b);    # devolve -1 ($a < $b)
```

Listagem 13: exemplo de uso do operador <=>

5.3.2. Operador cmp

O operador "cmp" atua de forma análoga ao operador <=>, com exceção que os operandos são strings.

Exercício 2

Elabore o **programa le_e_compara.pl** que deve numa primeira fase, pedir um número ao utilizador e indicar se esse número é menor ou superior a 20. Numa segunda fase, o programa lê uma string, comparando-a com a string "SistemasOperativos".

6. Funções numéricas e de strings

O PERL disponibiliza um elevado número de funções. Apresentam-se de seguida algumas dessas funções. Uma lista mais completa de funções PERL está em http://perldoc.perl.org/index-functions.html.

Função	**o que faz…**
abs	valor absoluto
chr	caráter cujo código ASCII é x
int	parte inteira de um real
lc	torna minúsculas todas as letras de uma string
lcfirst	torna minúscula o primeiro caráter de uma string
length	número de bytes
ord	ordem de um caráter na tabela ASCII
rand	número aleatório entre 0..1, ou entre 0 e o argumento passado exclusive.
uc	torna maiúsculas todas as letras de uma string
ucfirst	torna maiúscula o primeiro caráter de uma string

Tabela 6: Algumas funções do PERL

No PERL, o acesso à documentação respeitante a uma função pode ser feito através do utilitário `perldoc`, na seguinte forma:

```
perldoc -f NOME_DA_FUNCAO
```

Poderá ser necessário instalar a aplicação, o que se consegue da seguinte forma:

```
sudo apt-get install perl-doc
```

7. Estruturas de decisão

7.1. if, if…else, if…elsif

Umas das estruturas de decisão mais usadas, é o "if" e suas variantes. De seguida, ver-se-á a sintaxe destas, que é muito similar ao C. Uma diferença importante em relação ao C, é que no PERL é sempre obrigatória a criação de bloco através de chavetas, mesmo que o bloco apenas tenha uma instrução. Esta regra é válida para todas as estruturas do PERL que sejam orientadas ao bloco (if, while, etc.).

```
# if

if (teste)
{ # início do bloco código

} # fim do bloco


# if…else
if (teste)
{
  # código
}
else
{
  # código
}
```

```
#if…elsif

if (teste)
{
  # código
}
elsif (teste)
{
  # código
}
elsif (teste){
  # código
}
else
{
  # código
}
```

Exercício 3

Elabore o programa `positivo_ou_negativo.pl`, que deve pedir um número ao utilizador e indicar se o número inserido é positivo ou negativo.

8. Estruturas de repetição

8.1. Ciclos while

Os ciclos são outra forma de controlar a execução do código. Nos ciclos `while` a sintaxe também é igual à do C.

```
while (teste)
{
  #código
}

do
{
  #código
} while (teste);

# Exemplo
do
{
  print ("Senha:");
  $res=<STDIN>;
  chomp $res;
} while ($res ne "pass");
```

Listagem 14: Sintaxe(s) do ciclo while

Exercício 4

Elabore o programa `ate_sete.pl` que deve contar o número de tentativas necessárias para que saia, de forma aleatória, o número sete, , com uma gama de números de 0 a 10.
Nota: use as funções *srand* e *rand* juntamente com ciclo while.

8.2. until

A estrutura de repetição ***until*** atua de forma inversa ao while, isto é, mantém-se o ciclo se a expressão for falsa.

```
until( expressão )
{
   # expressão
}

do
{
   # expressão
} until( expressão );
```

Exercício 5

Rescrever o exemplo seguinte, recorrendo à estrutura `untile` depois com a estrutura "`do while`"; Quais são as situações em que deverá ser empregue a estrutura "`do while | do until` em detrimento da `while | until`?

```
#!/usr/bin/perl –w
$pal = "";
while ( uc($pal) ne "FIM" )
{
  if($pal){ print("a palavra escrita foi: $pal \n"); }
  print("Escreva uma palavra: ");
  chomp($pal=<STDIN>);
}
```

Listagem 15: Exemplo do uso do ciclo *while*

8.3. for

Estrutura de repetição muito semelhante à existente na linguagem C.

```
for( condição inicial; condição teste; condição de mudança)
{
  …;
}
```

Exercício 6

Implemente um programa que apresente a tabuada de um número X que está compreendido entre 1 e 10. O número X é pedido ao utilizador.

Exercício 7

Num sistema informático, a tabela ASCII contém o mapeamento entre código numérico e o respetivo carater. Por exemplo, na tabela ASCII, a letra 'A' (maiúscula) tem o código 65, ao passo que a letra 'a' (minúscula) tem o código 97. Na linguagem PERL, dado um determinado código numérico ASCII, é possível obter o respetivo caráter através da função **chr(**...). Por exemplo, para ser devolvido o 'A', bastará chamar **chr(65),** sendo que caso se pretenda escrever na saída padrão se poderá usar o seguinte código:

```
printf("%s\n", chr(65));
```

Elabore, recorrendo à linguagem PERL, o programa mostraASCII.pl que deve mostrar os carateres da tabela ASCII cujo código está compreendido entre 32 (inclusive) e 127 (inclusive). Por forma a possibilitar uma melhor visualização da tabela ASCII, devem ser apresentados cinco resultados por linha, sendo para cada elemento mostrado o código numérico, dois pontos (:) e o caráter correspondente ao código (entre plicas). A listagem abaixo ilustra a saída pretendida.

```
32:' ' 33:'!' 34:'"' 35:'#' 36:'$'
37:'%' 38:'&' 39:''' 40:'(' 41:')'
(...)
122:'z' 123:'{' 124:'|' 125:'}' 126:'~'
127:'^?'
```

8.4. foreach

A estrutura *foreach* itera uma lista de valores, processando sequencialmente os elementos da lista (um elemento em cada iteração).

```
Foreach $iterador ( lista de elementos )
{ …; }
```

```
my $I = 1;
foreach my $Num ( "um", "dois", "três", "quatro" )
{
  print ("Iteração $I  : $Num\n");
  $I++;
}
```

Listagem 15: Estrutura *foreach* para lista de *strings*

```
my $I = 1;
foreach my $Num (1..100)
{
  print ("Iteração $I  : $Num\n");
  $I++;
}
```

Listagem 16: Estrutura *foreach* para lista de números

8.5. Controlo de Estruturas de repetição

8.5.1. last

A instrução força a paragem da execução do ciclo (semelhante ao `break` do C), sendo que a execução *salta* para depois do fim da estrutura do ciclo.

8.5.2. next

Termina a execução da iteração corrente do ciclo, regressando ao topo do ciclo para iniciar a próxima iteração (semelhante ao `continue` do C).

8.5.3. redo

Semelhante à instrução next, com exceção que a condição do ciclo não é reavaliada, isto é, a iteração corrente é novamente executada.

Exercício 8

Qual é a saída do seguinte programa em PERL ?

```
#!/usr/bin/perl -w

for ($i=0; $i<10; $i++)
{
   if ( ($i % 2) == 0)
   {
      next;
   }
   if ( ($i % 5) == 0 )
   {
      last;
   }
   print("$i\n");
}
```

Listagem 17: Código do exercício proposto

8.6. Leitura de dados em ciclos de repetição

Uma técnica comum no PERL é a leitura de dados de entrada a partir da entrada padrão <STDIN> (que está associada, por omissão, ao teclado do terminal). O exemplo a seguir (`lestdin.pl`) tira partido da leitura do STDIN.

```
#!/usr/bin/perl -w
use strict ;

# Programa "lestdin.pl"
my $j=0;
my $linha="";
$j = 1;
while ($linha = <STDIN>)
{
   print("linha $j - $linha\n");
   $j++;
}
```

Listagem 18: Leitura de dados em ciclo

Exercício 9

Execute, num diretório onde exista o ficheiro de texto '**ficheiro.txt**', o programa `lestdin.pl` da seguinte forma:

```
$ lestdin.pl < ficheiro.txt
```

9. Vetores (*arrays*)

Até agora lidamos com elementos **escalares**, isto é, com um só valor. O PERL suporta também o conceito de ***array***, que representa um vetor de elementos escalares. Isto é,. um *array* é formado por vários escalares. O símbolo "@" identifica uma variável como sendo um array.

```
#!/usr/bin/perl

$a = "120";                    # "variável a" na forma de escalar
@a = ('1', '2', '3', '4');     # o array a
@num = (1 .. 5);   # Array especificado através de uma gama de números

# Array de strings
@strings = ('um','dois','três','quatro','cinco');

# Alternativa no array de strings: qw (quoted word)
@strings = qw( um dois três quatro cinco);

@misto = (3.1415, 'zero', 20);              # Array misto
@misto2 = (@num, @strings);                # outro array misto

$misto[2] = 5;       #altera o 3° elemento de @misto

@num[2,3,4] = (23, 33, 43);  #altera os valores do 3°,4° e 5° elemento de
@num

print('@strings =',"@strings \n");
print('$strings[3]=',"$strings[3] \n");
print('@misto =',"@misto \n");
print('$#misto=',"$#misto \n"); # $#misto contém o último índice da lista @misto
```

Listagem 19: Arrays em PERL

9.1. Acesso aos elementos de um array

O acesso a um elemento de um array processa-se através da seguinte sintaxe:

`$Nome_do_array[índice]`, em que **índice** representa a posição do elemento pretendido (tal como no C, o primeiro elemento tem índice 0). O PERL, como linguagem flexível que é, admite índices negativos. Por exemplo, o índice `-1` refere-se ao último elemento do array (i.e. o mesmo que `$array[$#array]`).

```
@Pares = (2, 4, 6, 8, 10);
print("$Pares[0]\n");  # imprime primeiro elemento do array
print("$Pares[10]\n"); # acesso a um valor indefinido do array
```

Listagem 20: Exemplo que usa arrays

9.1.1. Acesso ao último elemento de um array

O número de elementos de um array consegue-se através de $#Nome_do_array acrescido em uma unidade

```
@vector = (1, 3, 5, 7, 9);
$j = 0;
while ($j <= $#vector ) #índice do último elemento
{
  print("Vector: $vector[$j]\n");
  $j++;
}
```

Listagem 21: Obter o número de elementos do array

A estrutura de iteração "**foreach**" é uma alternativa mais expedita de conseguir a iteração de um *array*, conforme ilustrado no seguinte exemplo:

```
foreach $elm (@vector){
  print("$elm\n");
}
# Nota – outra forma ainda mais simples para mostrar o array, seria:
print("@vector\n"); # imprime todos os elementos separados por espaço
```

Listagem 22: Percorrer array sem obter o número de elementos

9.1.2. Contexto – escalar e vetorial

No PERL, consoante o contexto em que se insere, uma variável do tipo array, pode assumir um valor escalar, ou um valor vetorial.

```
@vetor_1 = (0, 2, 4, 5, 8);
@vetor_2 = @vetor_1;      # cópia do vetor_1 (contexto vectorial)
$NumElementos = @vetor_1;# N° de elementos do array (contexto escalar)
```

Listagem 23: Cópia de arrays

O PERL disponibiliza a função *scalar* para a criação de um contexto escalar em situações em que o mesmo não é necessariamente explícito.

Exemplo:

```
# mostra o número de elementos de um vector (contexto escalar)
@vogais = qw( a e i o u);
print ("Número de vogais: scalar(@vogais)\n");
```

Listagem 24: O contexto escalar de um array

9.1.3. Ordenação de arrays

A ordenação dos elementos de um *array* obtém-se através da função `sort`. Por omissão, a função `sort` ordena de acordo com os códigos da tabela ASCII.

```
# Vector de strings
@Strings = (”Paris”, ”Lisboa”, ”Madrid”, ”Barcelona”, ”Leiria”, ”Porto”);
@Strings_ordenadas = sort @Strings;
print(”@Strings_ordenadas”);
```

Listagem 25: Ordenação de vetores (*arrays*)

A ordenação acima apresentada funciona corretamente com strings. Contudo, para a ordenação de números torna-se necessário acrescentar uma função (anónima) de ordenação, função que deve comparar dois elementos individuais (qualquer algoritmo de ordenação se reduz, na sua essência, a comparar dois elementos). No PERL, a função é indicada da seguinte forma, com $a a representar o operando da esquerda e $b o operando da direita.

```
  sort { $a <=> $b }
```

A listagem seguinte exemplifica a ordenação de um vector de números, recorrendo-se ao `sort` do PERL.

Exemplo:

```
# Mostra o vector numérico @Nums_L ordenado
my @Nums_L = (1974, 1, 2, 27.12, 26.0, 567, 23);
my @Nums_Sorted_L = sort{ $a <=> $b } @Nums_L;
print("\@Nums_L=@Nums_L\n");
print("\@Nums_Sorted_L=@Nums_Sorted_L\n");
```

Listagem 26: Ordenação de vectores (*arrays*) numéricos

9.1.4. Iterando os elementos de um vetor através do foreach

No PERL é possível iterar os elementos de um vetor, através da estrutura de iteração `foreach`. Por exemplo, considere o seguinte código:

```
#!/usr/bin/perl
use strict;
use warnings;
# Código que itera pelos 10 elementos do vector ”@Numeros_L”
my @Numeros_L = (1..10);
foreach my $Num (@Numeros_L){
  print(”$Num=’$Num’\n”);
}
```

Listagem 27: Iteração do conteúdo de um vector através do foreach

9.1.5. Outras funções úteis para utilização de listas

Apresentam-se na tabela seguinte algumas das funções disponibilizadas pelo PERL para a manipulação de listas.

Nome da função	Funcionalidade
`push LIST, VALUES`	Copia os valores VALUES para o fim da lista LIST
`pop LIST`	Remove da lista LIST o último elemento
`shift LIST`	Retira o 1º elemento da LIST; Em caso de omissão da LIST usa o@ARGV ou o @_
`unshift LIST, VALUES`	Copia os valores VALUES para o início da LIST; Em casode omissão da lista LIST usa o @ARGV ou o @_
`join EXPR, LIST`	Efetua a junção dos elementos da LIST numa string usando a EXPR para unir os elementos
`split PATTERN, EXPR`	Separa o valor de EXPR de acordo com o PATTERN
`reverse LIST`	Inverte a lista LIST

```
#!/usr/bin/perl
# Exemplo de uso das funções push, unshift e join
@list = (1,5,3,9,7);
@l = ('a','bbbbbbbbbbbbb');
push @list,@l;       #insere os elementos de @l no final da lista @list
push @list,33;

#insere o elemento 44, seguido do 55 no final da lista
push(@list,44,55);
#remove o elemento 1 da lista e imprime-o
print("2:\t",shift @list,"\n");
unshift(@list,100);     #insere o elemento 100 no inicio da lista
$i = 200;
unshift(@list,$i);

#imprime cada elemento da lista numa nova linha
print("----\n",join("\n",@list),"\n");

#$res é uma string composta pelos elementos de @list separados por '+'
$res = join("+",@list);
print($res,"\n");
```

```
#!/usr/bin/perl
# Exemplo de aplicação da função split
$s ="Isto é uma cadeia de carateres separados por espaços";
# separa a string $s pelas suas palavras@r = split(/ /,$s);
  print("1:\t",$r[0],"\n");
print("2:\t",$r[$#r],"\n");
$s ="Isto é uma frase que, por motivos de demo, está separada por
vírgulas";
@r = split(/,/,$s); #$s é separada pelas vírgulas
print("3:\t",$r[0],"\n"; print "4:\t",$r[$#r],"\n");

# Exemplo de aplicação da função join e reverse
@list = (1,5,3,9,7);
print(join(";",@list),"\n");
@tsil = reverse(@list);
print(join(":",@tsil),"\n");
```

Exercício 10

Escreva o programa **le_ordena_5_pals.pl** que peça cinco palavras ao utilizador e que as mostre de forma ordenada na saída padrão (ecrã).

Nota: Use a função *push(array,elemento(s))* para adicionar elementos ao array.

10. Execução de um programa externo num *script*

Em PERL, para executar um programa externo (por exemplo, um comando da BASH) pode recorrer-se à função `system`.

A função `system` aceita argumentos onde, para além do nome do programa externo a executar, também podem ser indicados argumentos de linha de comando para o programa externo que se pretende executar.

```
#!/usr/bin/perl
use strict;
use warnings;
system("comando","argumento1","argumento2","argumento3");
```

O valor de retorno do comando executado poderá ser verificado através da variável especial **$?**. O valor 0 na variável $? significa que a execução do programa externo decorreu com sucesso. Pelo contrátio, caso o valor seja **-1**, a execução do programa externo falhou. Neste caso, pode-se utilizar a variável especial **$!** para aceder à mensagem de erro devolvida.

```
#!/usr/bin/perl
use strict;
use warnings;

system("comando", "argumento1");
if($? != 0){
  print STDERR ("O comando falhou: $!\n");
}
else
{
  printf("O comando terminou com o valor de saída: %d", $? >> 8);
}
```

Exercício 11

Tendo em conta o exemplo anterior, recorra à função `system` e elabore o *script* `listaDirs.pl` que deve mostrar a listagem da diretoria atual e a listagem da diretoria `/bin`. Deve verificar os valores de retorno da execução da função `system`, por forma a personalizar o *output* do programa.

10.1. Captura da saída produzida pela execução de programas externos

Em PERL, uma das formas de se executar um programa externo e capturar a saída produzido por esse programa é recorrendo-se ao operador de "plicas invertidas", à semelhança da substituição de comandos da *shell*. Para tal, indica-se o programa externo (muitas vezes um comando da shell, que pode conter inclusivamente operadores especiais da

shell, como *pipes* "|" e redireccionamentos ">" e ">>") com as eventuais opções da linha de comando, tudo isso, delimitado pelo operador plica invertida. O operador plica invertida corresponde ao acento grave (`). Atenda-se ao seguinte exemplo, que executa o comando "ls" com as opções "-l -a" (produzindo portanto uma listagem em formato longo de todos os ficheiros do diretório corrente).

```
#!/usr/bin/perl
@SaidaProg_L = `ls -l -a`;
  print  ("\@SaidaProg_L = @SaidaProg_L\n");
```

Listagem 28: captura da saída produzida pela execução de um programa externo

Considere o exemplo seguinte, no qual é empregue o operador plicas invertidas para executar um comando composto pelos utilitários `file` e `sort`:

```
#!/usr/bin/perl
use strict;
use warnings;
# Filename: 'execFile.pl'
# Author: Patricio R. Domingues
# Created: April 2013
#
# Execute the 'file' utility over all files of the
# /bin/directory resorting to the `...` PERL functionality.
# The ouput is MIME (option -i of file) and it's sorted by
# the MIME type (sort -k 2)
#
# file -i >> generate MIME output
# sort -k 2 >> sort using the 2nd field as the sorting key
my $Exec_S = 'file -i /bin/* | sort -k 2';
my @Output_L = `$Exec_S`; # We kept the output in an array
chomp(@Output_L); # chomp the whole array

# Iterate over the ouput (one file per row)
my $RowNum=0;
foreach my $Output_S (@Output_L){
        $RowNum++;
        printf("[%03d]:'$Output_S'\n", $RowNum);
}

exit(0);
```

Listagem 29: Operador "plicas invertidas"

Exercício 12

Recorrendo ao operador "*plica invertida*" elabore a script `utilizadores.pl` que deve mostrar por ordem alfabética, os *logins* de todos os utilizadores do sistema linux onde está a ser executada. O script deve ainda mostrar no final, a versão do kernel do sistema, bem como o nome da máquina em maiúscula

Sugestão: (1) para cada *login* existe uma linha no ficheiro /etc/passwd. (2) o utilitário `uname` disponibiliza informações sobre o kernel de um sistema linux.

11. Passar valores pela linha de comandos

11.1. Argumentos da linha de comandos

A linguagem PERL permite a utilização da linha de comando para passar argumentos. Os argumentos são guardados numa variável especial do tipo array (@ARGV). Por exemplo, se executarmos o seguinte programa: **prog.pl fich1.txt fich2.txt fich3.txt**, a variável **@ARGV** terá os seguintes elementos ("**fich1.txt", "fich2.txt","fich3.txt**")

Para acedermos individualmente aos elementos da variável @ARGV, utilizam-se as técnicas de acesso a elementos de arrays, ilustradas no exemplo seguinte.

NOTA: Tenha em atenção, que em PERL, o vetor de argumentos "@ARGV" não contém o nome da script, sendo que $ARGV[0] corresponde ao primeiro argumento passado pela linha de comando. O nome da script está acessível através da variável especial **$0**.

Identificador	Funcionalidade
`@ARGV`	Contém a lista de argumentos passados para o script
`$ARGV[0]`	é o primeiro argumento.
`$0`	O nome do script que está a ser executado
`$\`	Separador de linha na saída. Por omissão, é nulo, logo entre dois print (ou outra função de saída) é necessário definir "\n" quando: esta variável não está 'ativada' e se pretende terminar a linha
`$/`	Separador de linha na entrada. Por omissão é o newline (\n) $, Separador de campo na saída. Por omissão, é nulo, logo entre dois argumentos de um print (ou outra função de saída) é necessário definir "um separador" quando esta variável não está 'ativada' e se pretende separar os campos
`$_`	A variável de entrada, por omissão e argumento por omissão para diversas funções buitin. O man perlfunc descreve para cada função, o comportamento, em caso de omissão do argumento, se este for opcional.

```
#!/usr/bin/perl

#aviso de erro se não receber argumentos
if(@ARGV < 1){
  print STDERR ("erro:   $0 fich1, fich2 .... fichn\n");
  exit -1;
}

$\ ="\n";
$, =' <-> '; #uma string qualquer
print ("\n\t++++\n");
for (@ARGV){
        print;          #assume como argumento do print o valor $_
}
print ("\n\t----\n");
$, =' :-> '; #uma string qualquer
foreach (@ARGV){        #por omissão $_ assume o valor iterado
        print ("Tamanho do nome do fx $_",length);
}
print ("\n\t****E agora o cat de todos os ficheiros recebidos como
argmentos***\n");
$/ = ord(0); #o separador de linha na entrada é o EOF (codigo ascii 0)

while (<>){     #lê o fx em cada iteração
        #$ARGV tem o nome do fx do qual se está a ler
        print ("\n----------Ficheiro $ARGV --------------\n");
        #$_ por omissão contém o fx lido.
        # O print por omissão imprime $_
        print;
}
```

Listagem 30: Passagem de valores pela linha de comandos

```
#!/usr/bin/perl
use strict;
use warnings;
# Declaração de variáveis
my ($Argc);
my ($i);

#-----------------------------------------------
# Exemplo com os argumentos da linha de comando
# $ARGV[0]: 1° argumento da linha de comando
# Atenção $ARGV[0] não é o nome do script!
# O nome do script é obtido através de $0
# Executa o programa da seguinte forma
# nome.pl argumento_1 argumento_2 teste
#-----------------------------------------------

# Mostra nome do ficheiro a ser executado (i.e. script)
print("Nome do script: $0\n");

# Número de argumentos
$Argc = $#ARGV+1;
if ( $Argc > 0 ) {
  print("Existem $Argc argumentos da linha de comando: @ARGV\n");
}
else {
  print("Não existem argumentos da linha de comando\n");
}

$i = 0;
while ( $i < $Argc ) {
  print ("ARGV\[$i\]: $ARGV[$i]\n");  # Nota: \ antes de [ e ]
  $i++;
}
```

Listagem 31: Passagem de valores pela linha de comandos

Exercício 13

Elabore, com recurso à linguagem PERL, o script **.template2perl.pl**, cujo propósito é o de gerar um ficheiro cujo nome é indicado como único parâmetro da linha de comando. O ficheiro a gerar destina-se a ser empregue como base para a escrita de um programa em PERL, sendo que conteúdo do ficheiro deve respeitar o formato da listagem abaixo mostrado que corresponde à execução do script da seguinte forma:

./**template2perl.pl a.pl**

Tenha em atenção que os campos sublinhados (**Hostname, Author, Created**) devem ser gerados dinamicamente pelo script, e correspondem, respectivamente, ao nome da máquina, login do utilizador que executa o script e à data de criação.

```
#!/usr/bin/perl -w
use strict;

# Filename: 'a.pl'
# Hostname: 'xlinux'
# Author: 'root'
# Created: 'Wednesday 20150509 12h39'
#=====================================================
# Config
#=====================================================
# Inserir a configuracao aqui
my (variaveis a declarar);
#=
# Code
#=
```

12. Listas Associativas (*hashes*)

Para além dos arrays, O PERL providencia uma segunda forma de agrupar elementos: as listas associativas (*hashes* na literatura Anglo-Saxónica).

Uma lista associativa é uma lista não ordenada de chaves e elementos, servindo a chave (que é normalmente do tipo string) para aceder a um elemento. Assim, ao contrário dos vetores, no qual o acesso se processa através de um índice, nas listas associativas o acesso processa-se através da chave que está associada ao elemento com a seguinte sintaxe **$Nome_da_Lista{chave}**.

Em PERL, as listas associativas são identificadas pelo caráter %, logo no início do identificador.

```
# hash com os meses (chaves) e o número de dias (elementos)
# Por uma questão de clareza as chaves encontram-se sublinhadas
%NumDiasMeses = ('Janeiro', 31, 'Fevereiro', 28, 'Março', 31);

# Acesso ao n° de dias do mês de Janeiro
print $NumDiasMeses{'Janeiro'};
```

Listagem 32: Listas associativas para meses

12.1. Adicionar elemento à lista

```
# Código postais
%CodigoPostal = ('2400','Leiria','3000','Coimbra','1000','Lisboa');
print ("O código postal 2400 é de : $CodigoPostal{'2400'}\n");


# Acréscimo de mais um elemento ao HASH
$CodigoPostal{'2000'} = 'Porto';
print("O código postal 2000 é de : $CodigoPostal{'2000'}\n");
```

Listagem 33: Listas associativas para códigos postais

```
#É possivel usar '=>' em vez da vírgula para separar os elementos de uma
lista
%natural = ('Ana Cristina' => 'Braga', 'Jose Manuel' => 'Viana', 'Antonio
Joaquim' => 'Aveiro');

print "A Naturalidade de Ana Cristina e: $natural{'Ana Cristina'}\n";

@a = keys %natural;
print "@a \n";

#Para alterar o valor de duas ou mais chaves temos de trabalhar em
contexto de lista
$natural {'Ana Cristina'} = 'Aveiro';


#Para uma lista com os valores de uma lista Associativa poderemos fazer
@y = @natural{'Ana Cristina','Jose Manuel'}; #@y é uma lista composta pela
naturalidade da 'Ana Cristina' e do 'Jose Manuel'
print "Y: @y\n";
```

```
#outra forma
@a = keys %natural; #Lista de todas as chaves
print "@a \n";
@a = values %natural; #Lista de todos os valores
print "@a \n";
```

12.2. Apagar um elemento de uma Lista Associativa

A eliminação de um elemento de uma lista associativa é conseguida através do operador *delete*.

Exemplo:

```
delete $ListaAssociativa{'chave'};
```

```
#!/usr/bin/perl
%A = ( 7 => 'aaaaa', 'a' => 23, 12 => 123, 'b' => 'sssssssss');
@A = (1, 5, 3, 6, 4, 2, 1);    #NOTA: @A é uma variável distinta de %A

@b = keys %A; print "@b \n";

delete $A{'a'};
#os pares (7, 'aaaaa') e (12,123) de %A são removidos
delete @A{'7', '12'};

@b = keys %A;
print "@b \n";


delete @A[1..4];              #os elementos 5,3,6,4 de @A são removidos
print "@A \n";
```

12.3. Iterar uma Lista Associativa

A iteração de uma lista associativa (i.e., percorrer todos os seus elementos) requer o uso da função `keys` que devolve um array com as chaves da lista referenciada.

```
%CodigoPostal = ('2400', 'Leiria', '3000','Coimbra', '1000','Lisboa');
foreach $Codigo (sort keys %CodigoPostal)
{
  print("$Codigo - $CodigoPostal{$Codigo}\n");
}
```

Listagem 34: Iterar uma lista associativa

Por sua vez, a função `values`, quando aplicada a uma lista associativa, devolve os elementos da lista (i.e. tudo com excepção das chaves).

```
%CodigoPostal = ('2400','Leiria', '3000','Coimbra', '1000','Lisboa');
# Constrói array apenas com os elementos da lista associativa
@Elementos = values( %CodigoPostal );
```

Listagem 35: elementos da lista associativa

12.4. Extração de dados para Arrays ou Listas Associativas

Quando se procede à leitura de dados a partir de ficheiros de texto, os vários elementos de entrada encontram-se possivelmente numa linha, quando normalmente o requerido é o acesso individual a cada palavra. Para tal, o PERL apresenta a função `split`.

```
# Simulamos uma linha de um ficheiro com uma variável escalar string
$StringdeNumeros = '12 24 34 56 78 23 21';
# Especificamos o espaço (' ') como separador
@Palavras = split(' ', $ StringdeNumeros);
```

Listagem 36: Uso da função split

```
Função split
- Como aceder a uma palavra no PERL ($Linha contém várias palavras)?
# o array @palavras é constituído pelas palavras de $Linha
@palavras = split (' ', $Linha); # palavras recebe as palavras da $linha
- Como aceder aos carateres de uma palavra no PERL ($Palavra contém uma
palavra)?
# o array @carateres é constituído pelos carateres de $Palavra
@carateres = split (//, $Palavra);
```

Listagem 37: Mais exemplos de uso da função split

12.5. Outras funções úteis para a manipulação de arrays ou listas associativas

each %HASH → Retorna o próximo par chave=>valor ou a próxima chave, dependendo do contexto
exists $HASH{$key} # Testa se uma determinada chave existe na HashTable

```
#!/usr/bin/perl
%A = ( 7 => 'aaaaa', 'a' => 23, 12 => 123, 'b' => 'sssssssss');
@p = each %A;                  #@p fica com o par (7,'aaaaa')
@k = each %A;                  #@k fica com o par ('a',23)
print("1:\t$p[0] $p[1]\n");
print("2:\t$k\n");
@p = each %A;                  #@p agora assume o valor (12,123)
$k = each %A;
print("3:\t$p[0] $p[1]\n");
print("4:\t$k\n");
```

```
#!/usr/bin/perl

%A = ( 7 => 'aaaaa', 'a' => 23, 12 => 123, 'b' => 'sssssssss');
print "12 é uma chave do array e o seu valor é: $A{12}\n" if exists

$A{12};
print "'12' é uma chave do array e o seu valor é: $A{'12'}\n" if exists
$A{'12'};

$i = 12;
print "$i é uma chave do array e o seu valor é: $A{$i}\n" if exists
$A{$i};

$i = 'aa';
print "$i não é uma chave do array\n" unless exists $A{$i};

$i = 'a';
print "$i não é uma chave do array\n" unless exists $A{$i};
```

Exercício 14

Escreva o programa `lista_e_conta.pl` que pede um ficheiro ao utilizador e devolva a lista de palavras encontradas, e quantas vezes se repete cada uma delas.

Exercício 15

Escreva o programa `traduz.pl` que procede à tradução do conteúdo de um ficheiro com palavras em Português para Inglês. O nome dos ficheiros deve ser indicado através dos argumentos da linha de comando, sendo que oprimeiro argumento corresponde ao nome do ficheiro contendo o dicionário e o segundo argumento deve conter o nome do ficheiro a ser traduzido. O programa deve mostrar na saída padrão o ficheiro traduzido, sendo que nos casos em que não exista no dicionário uma palavra do ficheiro a traduzir, deve-se manter a palavra não traduzida.

O ficheiro dicionário deverá ter o seguinte formato:

```
rua-street
casa-home
sol-sun
frio-cold
chuva-rain
```

13. Variáveis de Ambiente

Em PERL, existem ainda as variáveis de sistema, entre elas as mais utilizadas são a variável de ambiente, que é uma variável do tipo lista associativa (**%ENV**) onde são guardados diversos valores, como por exemplo o nome do utilizador que está a correr o programa (**$ENV{'USER'}**);

```
#!/usr/bin/perl –w

while(($chave, $valor) = each(%ENV))
{
  print("A chave $chave contem o valor de $valor\n");
}
```

Listagem 38: Listagem das variáveis de ambiente

14. Ficheiros

À semelhança de outras linguagens de programação, no PERL, a utilização de um ficheiro para leitura ou escrita requer a abertura do ficheiro.

14.1. Abrir um ficheiro (Função open)

Para abrir um ficheiro a biblioteca de funções do PERL disponibiliza a função **open**.

Sintaxe: **open (FileVar, OpenMode, FileName);**

FileVar: nome que a utilizar no programa **PERL** para identificar o ficheiro (*file handle)*. Para esta variável é aconselhada a utilização de nomes escritos em maiúsculas, pretendendo-se com este procedimento tornar mais fácil a distinção das variáveis tipo de ficheiro das restantes variáveis do programa.

OpenMode: carater que permite indicar o modo de acesso ao ficheiro.

FileName: Nome do ficheiro que se pretende abrir. Se pretendermos abrir um ficheiro que não esteja na diretoria do programa, é necessário indicar o caminho completo do ficheiro.

Exemplos:

```
open(FICHEIRO, '<', "ficheiro.txt");
open(FICHEIRO, '>', "/local/home/ficheiro.txt");
```

14.2. Modo de acesso a ficheiros

Quando se abre um ficheiro é necessário indicar o modo de acesso ao ficheiro. Existem três modos de acesso a ficheiros:

Modo	Descrição
Leitura	Apenas permite a leitura do conteúdo do ficheiro
Escrita	Elimina o conteúdo do ficheiro e escreve por cima
Acrescentar	Acrescenta ao conteúdo do ficheiro

Tabela 7: Modos de abertura de ficheiros

Utilização da função "open" no modo:

leitura: open(FICHEIRO, '<', "ficheiro.txt");
Escrita: open(FICHEIRO, '>', "ficheiro.txt");
Acrescentar: open(FICHEIRO, '>>', "ficheiro.txt");

Nota: Não deve esquecer que ao abrir um ficheiro:

- O conteúdo do ficheiro é destruído se for aberto no modo escrita;
- Não é possível ler e escrever para o mesmo ficheiro em simultâneo;

14.3. Verificação do sucesso da operação "open"

Antes de se utilizar o descritor iniciado pela função *open*, deve-se verificar se o ficheiro foi corretamente aberto. Para esse efeito, a função *open* devolve um valor diferente de zero ("verdadeiro" na interpretação do PERL) se a operação tiver sido bem sucedida, e zero ("falso"), caso não tenha sido possível abrir o ficheiro (por exemplo, o ficheiro não existe, ou o utilizador não tem permissões para o mesmo, etc.).

O exemplo seguinte ilustra uma forma de testar o valor de retorno da função *open*, terminando a execução caso a operação de abertura falhe. Concretamente, a estrutura de controlo *unless* é utilizada para verificar se o ficheiro foi aberto com sucesso.
A função *die* permite enviar uma mensagem ao utilizador e termina o programa, sendo que a variável do sistema **$!** devolve a mensagem do sistemas descritiva do erro. Finalmente, a função *close* fecha o ficheiro que estava aberto.

```
#!/usr/bin/perl -w

unless(open(FICHEIRO, '<', "dados.txt"))
{
   die("Ocorreu um erro a abrir o ficheiro: $! \n");
}
print("O ficheiro foi aberto com sucesso!\n");
close(FICHEIRO);
```

Listagem 39: *open* e *close*

O exemplo seguinte é equivalente ao anterior, sendo se empregou o operador lógico "||" em lugar do operado *unless*.

Exercício 16

Explique como funciona a solução apresentada com "||".

```
#!/usr/bin/perl -w

open(FICHEIRO,'<',"dados.txt")||die("Erro ao abrir o ficheiro:$!\n");
print("O ficheiro foi aberto com sucesso!\n");
close(FICHEIRO);
```

14.4. Ler o conteúdo de um ficheiro linha a linha

A forma de ler os dados de um ficheiro é idêntica à forma de leitura de dados do STDIN (não é de admirar, dado que o "STDIN" é ele próprio um descritor de ficheiro, neste caso, associado à entrada padrão).

```
$linha = <FICHEIRO>;
```

```
#!/usr/bin/perl -w
open(FICHEIRO, '<', "dados.txt") || die ("Ocorreu um erro a abrir o
ficheiro: $!\n");

print ("O ficheiro foi aberto com sucesso!\n");

$linha = <FICHEIRO>;
while($linha ne "")
{
  print $linha;
  $linha = <FICHEIRO>;
}
close(FICHEIRO);
```

Listagem 40: Ler linhas de um ficheiro

```
#!/usr/bin/perl -w
#--------------------------------------------------
# Programa que exibe o ficheiro "dados.txt"
# linha a linha, acrescido de um numero de linhas
# Patricio, 14-04-2000
#--------------------------------------------------
use strict;
my $i;
my $Linha;

open(FICHEIRO, '<', "dados.txt") ||
    die ("Ocorreu um erro a abrir o ficheiro: $!\n");

print("O ficheiro foi aberto com sucesso!\n");

$i = 1;
foreach $Linha ( <FICHEIRO> ){
  print( "linha $i - $Linha");
  $i++;
}
close(FICHEIRO);
```

Listagem 41: Programa que lê o conteúdo de um ficheiro linha a linha e as apresenta no ecrã

14.5. Ler todo o conteúdo de um ficheiro para uma variável array (@array = <FICHEIRO>;)

```
#!/usr/bin/perl -w
#-----------------------------------------------------------------------
# Programa que lê todo o conteúdo de um ficheiro para um array, sendo
# que cada elemento do array representa uma linha
#-----------------------------------------------------------------------
open(FICHEIRO, '<', "dados.txt") || die ("Ocorreu um erro a abrir o
ficheiro!\n";
print ("O ficheiro foi aberto com sucesso!\n");
@array = <FICHEIRO>;
foreach $linha (@array)
{
   print $linha;
}
close(FICHEIRO);
```

Listagem 42: Leitura completa de um ficheiro para um array

14.6. Escrita para ficheiro

A escrita para um ficheiro requer a abertura do ficheiro no modo escrita ou de acréscimo. Para se escrever no ficheiro pode-se utilizar a função **print** da forma indicada na listagem seguinte.

```
open (FICHEIRO, '>', ">saida.txt");
# Atenção ao parêntesis
print FICHEIRO ("Mensagem enviada para o ficheiro.\n");
```

Listagem 43: Função *print* aplicada a ficheiros

```
#!/usr/bin/perl -w

open(F_DADOS, '<', "dados.txt") || die ("Ocorreu um erro a abrir o
ficheiro!\n";
open(F_COPIA,'>', "copiaDados.txt") || die ("Ocorreu um erro a abrir o
ficheiro!\n";

$linha = <F_DADOS>;
while ( $linha ne "")
{
   print F_COPIA ($linha);
   $linha = <F_DADOS>;
}
close(F_DADOS);
close(F_COPIA);
```

Listagem 44: Programa que copia o conteúdo de um ficheiro para o outro

14.7. Redirecionar os ficheiros de entrada e saída

Quando se executa um programa num sistema UNIX pode-se redirecionar o STDIN (entrada padrão) e STDOUT (saída padrão). Supondo que temos o programa **listar.pl** e pretendemos que os dados de entrada sejam obtidos do ficheiro **bdados.txt** e os dados de saída sejam

direcionados para o ficheiro **sdados.txt**. Neste exemplo podíamos executar o programa da seguinte forma: **listar.pl < bdados.txt > sdados.txt**

Dentro do programa, sempre que aparece-se a instrução **$line = <STDIN>;** os dados eram lidos do ficheiro **bdados.txt**, e a instrução **print "$line";** os dados eram enviados para o ficheiro **sdados.txt**.

14.8. Determinar o estado de um ficheiro

Considere o exemplo:

```
#!/usr/bin/perl -w

unless (open(FICHEIRO, '<', "prog.txt")) #o mesmo que if(! open(...) )
{
   if (-e "prog.txt"){
       die("O ficheiro existe mas não pode ser aberto.\n");
   }
   else{
       die("O ficheiro não existe.\n");
   }
}
```

Listagem 45 – Estado de um ficheiro

Neste programa utilizamos um operador de teste de ficheiros **(-e "prog.txt"),** que nos permite saber se o ficheiro existe. Se existir o valor devolvido é um valor diferente de zero, se o ficheiro não existir devolve zero.

A tabela seguinte apresenta os operadores de teste de ficheiros.

Operador	Descrição
-b	Dispositivo tipo bloco
-c	Dispositivo tipo caráter
-d	É um diretório
-e	O nome existe
-f	É um ficheiro
-g	Tem o bit *setgid* marcado
-k	Tem o bit *sticky* marcado
-l	É uma ligação
-o	O utilizador é o dono
-p	É um pipe
-r	Tem permissões de leitura
-s	Ficheiro não vazio
-t	É um terminal
-u	Tem o bit stuid marcado
-w	Tem permissões de escrita
-x	Tem permissões de execução
-z	Ficheiro vazio
-B	É um ficheiro binário
-S	É um *socket*
-T	É um ficheiro de texto

Tabela 8: Operadores para ficheiros

14.9. Ler uma sequência de ficheiros

Considere o seguinte exemplo:

```
#!/usr/bin/perl -w

while ($linha = <>)
{
   print("$linha\n");
}
```

Listagem 46: Leitura de vários ficheiros

Supondo que o programa é executado com parâmetros, da seguinte forma: **prog.pl fich1.txt fich2.txt fich3.txt**. A instrução **($linha = <>)** lê a primeira linha do ficheiro fich1.txt, depois lê a segunda, até á última, quando terminar de ler todas as linhas do primeiro ficheiro começa a ler a informação dos restantes ficheiros. Pelo formato, o operador "<>" é apelidado de operador diamante.

15. Bibliografia

Para elaboração da ficha, foi empregue material, dos seguintes autores/locais:

- Documentação de Isidro Vila Verde - FEUP / Serprest http://paginas.fe.up.pt/~jvv/Assuntos/PERL/extradocs/perl-intro.html (Abril 2008)
- "Beginning PERL" http://www.perl.org/books/beginning-perl/ (Abril 2008)
- "Perl 5 Tutorial" http://www.cbkihong.com/download/perltut.pdf (Abril 2008)

16. Exercícios

1. Caça ao BUG, descubra os 4 erros no código que se segue:

```
# o meu código
#!/usr/bin/perl –w
use strict;

print ‘Escreve a palavra OPS: ’;
chomp($input=<STDIN>)
if($input=’OPS’){
  print “Obrigado!\n”;
} else {
  print “Não era essa a palavra!”;
}
```

2. Modificar o programa converte.pl, para converter graus Celsius em graus Fahrenheit.

3. Construa um programa em PERL que conte os dígitos de um número introduzido pelo utilizador.

4. Recorrendo à linguagem PERL, escreva o script *avg_e_std.pl*, que recebendo números através da entrada padrão (um número por linha), deverá devolver na saída padrão a contagem dos números, a indicação do menor e do maior número, bem como a média aritmética e o desvio padrão dos números. A saída deve apresentar o formato abaixo mostrado, quando a script é executada da seguinte forma:

```
seq 10000 | avgstd.pl

== SAIDA ==
#
# CONTAGEM=10000
# MENOR=1
# MAIOR=10000
# SOMA=50005000
# MEDIA=5000.5
# DESVIO_PADRAO=2886.75133151437
# CONTAGEM:MENOR:MAIOR:SOMA:MEDIA:DESVIO_PADRAO
10000:1:10000:50005000:5000.5:2886.75133151437
```

Nota: o desvio-padrão de um conjunto de números X é dado pela seguinte expressão:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

5. Elabore o programa EPrimo.pl, que verifica se o número indicado pelo utilizador através da entrada padrão é primo ou não. (sugestão: pode recorrer ao utilitário "fator" – man fator)

6. Aproveitando o programa EPrimo.pl, construa o programa ContaPrimos.pl que indica o número de primos existentes, entre 1 e 100000. Procure otimizar o seu código.

7. Elabore o programa Factor.pl, que dado um número, indica os seus fatores primos.

Sugestão: man fator

8. Escreva um *script,* que peça ao utilizador um número, adicione 3, depois multiplica-o por 2, subtrai 4, subtrai duas vezes o número introduzido pelo utilizador, adicione 3, e imprime o valor final.

9. Recorrendo à linguagem PERL, elabore a script "**plotprimos.pl**" que recebendo um, dois número na linha de comando ($numMin e $numMax) os deve interpretar, respetivamente, como o valor menor e o valor maior de uma gama de números inteiros. A script deve então calcular, para cada inteiro compreendido na gama, o seu número de fator primos, produzindo um ficheiro de nome "primos_$numMin_a_$numMax.dat" e

que contém em cada linha o número seguido do respetivo número de fatores primos (estando os valores separados por "|"). A linha do ficheiro deve estar ordenada, por ordem crescente.

```
plotprimos.pl 10 20

ficheiro primos_10_20.dat
10|2
11|1
12|3
13|1
14|2
15|2
16|4
17|1
18|3
19|1
20|3
```

10. Recorrendo à linguagem PERL

a) Escreva o script *multiplica_V1.pl*, que deverá mostrar o resultado da multiplicação dos números, fornecidos através de um ficheiro de texto (um número por cada linha).

b) Escreva o script *multiplica_V2.pl*, que deverá mostrar o resultado da multiplicação dos números, fornecidos através de um ficheiro de texto (um número por cada linha). Por exemplo, o script deverá poder ser executado da seguinte forma:

```
seq 1000 | ./multiplica_V2.pl
```

```
## Solução – "multiplica_V2.pl"
```

11. Recorrendo à linguagem PERL

a) Construa o *script* Txt2Html.pl, que a partir de um ficheiro de texto, escreve o código HTML, capaz de exibir o ficheiro de texto.

b) Repita a alínea anterior, de modo a nome do ficheiro de texto a ser convertido passe a ser especificado através da linha de comando. O conteúdo HTML do *script* deve ser escrito num ficheiro que mantém o nome do ficheiro original acrescido da extensão html.

Sugestão – esqueleto da página HTML

<HTML> <HEAD> <TITLE> ficheiro </TITLE> </HEAD>
<BODY>
Texto
</BODY>
</HTML>

12. Recorrendo a um *vetor associativo* (vulgo *hash*), construa uma script que efetua a contagem das palavras de um ficheiro de texto, cujo nome é indicado através do 1º argumento da linha de comando.

    a) Indique quantas vezes, existe a palavras "*xpto*"

    b) Mostre a frequência, de cada palavra, do ficheiro

    c) Repita a alínea anterior, mas exibindo a lista ordenada da(s) palavra(s) mais frequente(s) para as menos frequente(s).

    Nota:

```
@palavras = split ' ', $Linha; # palavras recebe as palavras da $linha
```

13. Recorrendo à linguagem PERL, construa o *script* Users2Html.pl, que permita saber os utilizadores com conta na máquina, e que construa um ficheiro, com o formato HTML, onde a informação dos utilizadores, deve ser colocada.
    O nome do ficheiro deve ser index.html, e deve ser passado no momento em que se executa o programa. **Users2Html > index.html**

    Sugestão – esqueleto da página HTML

    <HTML> <HEAD> <TITLE> ficheiro </TITLE> </HEAD> <BODY> <H1> cabeçalho </H1> </BODY> </HTML>

14. Recorrendo à linguagem PERL, construa o *script* mail2Html.pl, que permita saber o endereço de correio electrónico dos utilizadores, com conta na máquina e que construa um ficheiro com o formato HTML, onde essa informação deve ser colocada. O nome do ficheiro deve ser index.html, e o programa deve ser executado sem parâmetros, da seguinte forma: mail2Html.pl.

17. Soluções

De seguida são mostradas as so*luções* para alguns dos exercícios propostos.

17.1. Resolução do exercício 1

```
# o meu código -> não pode estar aqui!
#!/usr/bin/perl –w
use strict;
my $input;
print ‘Escreve a palavra OPS: ’;
chomp($input=<STDIN>);
if($input eq ’OPS’){
  print ("Obrigado!\n");
} else {
  print ("Não era essa a palavra!\n");
}
```

17.2. Resolução do exercício 2

```
#!/usr/bin/perl -w
use strict;

my $fahr=0;
my $cel=0;

print "Introduza a temperatura em Celsius: ";
chomp ($cel=<STDIN>);
$fahr=32+$cel*9/5;
printf("%.2f° Celsius = %.2f° Fahrenheit\n", $cel, $fahr);
```

17.3. Resolução do exercício 3

```
#!/usr/bin/perl –w
use strict;

my $palavra="";

while($palavra ne "fim")
{
  print "Dá-me uma frase ou palavra: ";
  chomp($palavra=<STDIN>);
  print "Escreveste: $palavra \n\n";
};
print "O programa chegou ao fim\n";
```

17.4. Resolução do exercício 4

```
#!/usr/bin/perl -w
use strict;
my $num=0;
my $n=0;
my $conta=0;
print 'Escreve um número inteiro: ';
chomp($num=<STDIN>);
$n=$num;
while($num>0)
{
  $conta+=1;
  $num=int $num/10;
}
if($n==0)
{
  $conta=1;
}
printf("%d tem %d digitos\n", $n, $conta);
```

17.5. Resolução do exercício 5

```
#!/usr/bin/perl -w
use strict;
my $n;
do
{
  print 'Escreve um número entre 0 e 100: ';
  chomp($n=<STDIN>);
}
while($n<0 || $n>100);
print"Escreveu o número $n\n";
```

17.6. Resolução do exercício 13

```
#!/usr/bin/perl -w

$title = "Identificacao";
$h1 = Utilizadores;
my $cmd = "cut -f1 -d: /etc/passwd";
open (IN, "$cmd |") || die ("Erro a abrir o ficheiro: $!\n");

@lista = <IN>;

print "<HTML>\n";
print "<HEAD>\n";
print "<TITLE>",$title,"</TITLE>\n";
print "</HEAD>\n";
print "<BODY>\n";
print "<H1><center>",$h1,"</center></H1>";

foreach $user (@lista)
{
   print "<br>$user";
}

print "</BODY>\n";
print "</HTML>";
```

17.7. Resolução do exercício 14

```
#!/usr/bin/perl -w

$elm = “”;
$title = "Identificacao";
$h1 = Utilizadores;
$Maquina = `hostname –f`  # Nome da máquina para forma endereço de mail
my $cmd = "cut -f1 -d: /etc/passwd";
open (IN, "$cmd |") || die ("Erro a abrir o ficheiro.\n");
open (OUT,”>index.html”) || ("Erro a abrir o ficheiro index.html.\n");

@lista = <IN>;

print OUT "<HTML>\n";
print OUT "<HEAD>\n";
print OUT "<TITLE>",$title,"</TITLE>\n";
print OUT "</HEAD>\n";
print OUT "<BODY>\n";
print OUT "<H1><center>",$h1,"</center></H1>";

foreach $user (@lista)
{
   $elm = “$user\@$Maquina”;
   print OUT "<br>$elm";
}

print OUT "</BODY>\n";
print OUT "</HTML>";
```