Ficha 3

Introdução ao Perl

9 de Março de 2015

1 Subrotinas

Em Perl quando se quer criar uma função usa-se o comando `sub`. Ao contrário de outras linguagens os parâmetros da subrotina são passados no *array* `@_`. Para além disso, o valor da última instrução executada pela subrotina é devolvido pela mesma. O próximo exemplo mostra algumas implementações da mesma subrotina, que calcula o máximo de 2 valores.

```
1  sub max_ilegivel {
2      ($_[0] > $_[1])?$_[0] : $_[1];
3  }
4  sub max1 {
5      my $x = shift;
6      my $y = shift;
7      ($x > $y) ? $x : $y;
8  }
9  sub max2 {
10     my($x, $y) = @_;
11     ($x > $y) ? $x : $y;
12 }
```

1.1 Retorno de valores e argumentos de tamanho variável

Contudo, também existe o comando `return` para retornar o valor da subrotina tal como nas outras linguagens. O comando `return` devolve o valor acabando imediatamente a subrotina. Também é importante frisar que como os argumentos da subrotina são passados no array `@_` o número de argumentos pode ser variável. O próximo exemplo evidencia isto mesmo ao procurar um elemento num array [1]. Repare que a primeira subrotina usa um truque, a avaliação do array `@_` como um valor de verdade o que equivale a perguntar se a lista não é vazia.

```
1  sub procurar1 {
2      my $valor = shift;
3      $encontrou = ($valor == shift) while(!$encontrou && @_);
4      $encontrou;
5  }
```

[1] A subrotina devolve verdadeiro ou falso caso o elemento, que é o primeiro argumento da subrotina, existe no resto dos argumentos da função

```
sub procurar2 {
    my $valor = shift;
    for $x (@_) {
        return 1 if $x == $valor;
    }
    return 0;
}
sub procurar3 {
    my ($valor, @lista) = @_;;

    for $x (@lista) {
        return 1 if $x == $valor;
    }
    return 0;
}

@lista = (1..10);
print "encontrou\n" if procurar1(5, @lista);
print "encontrou\n" if procurar2(5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

1.2 Variáveis locais e argumentos

É preciso cuidado ao manipular o array `@_` dentro da função porque se alteram os valores que foram usados na invocação como se pode verificar no exemplo a seguir. Repare que a subrotina `trocar_sem_var_local` não declara a variável `$t` como uma variável local e por isso o valor global da variável muda quando a subrotina é invocada.

```
sub trocar {
    my $t = $_[0];
    $_[0] = $_[1];
    $_[1] = $t;
}
$x = 2; $y = 3; $t = 4;
trocar $x, $y;
print "x = $x\ty = $y\tt = $t\n";
sub trocar_sem_var_local {
    $t = $_[0];
    $_[0] = $_[1];
    $_[1] = $t;
}
$x = 2; $y = 3; $t = 4;
trocar_sem_var_local $x, $y;
print "x = $x\ty = $y\tt = $t\n";
```

1.3 Argumentos passados por nomes e valores por omissão

Tal como já tínhamos visto antes, é possível atribuir um *array* (lista) a um *hash* (dicionário) e vice-versa. Usando esse conceito é possível implementar subrotinas que tenham argumentos

passados por nome que possam existir ou não e até ter valores atribuídos por omissão caso estes não sejam passados. O próximo exemplo chama-se `potencia` que deve ter dois valores: a base e o expoente. Vamos assumir que a base tem sempre que existir mas o expoente, caso não exista, deverá ter por omissão o valor 1.

Há um caso especial, a invocação que tem o caracter `&` antes. Neste caso, a invocação utiliza o array `@_` do nível actual.

```
sub potencia {
    my %args = @_;
    return "Falta a base" unless exists $args{base};
    $args{expoente} = 1 unless exists $args{expoente};
    $args{base} ** $args{expoente};
}

@_ = (base => 2, expoente => 3);
$x = potencia;
$y = potencia(); # Esta invocacao e a anterior sao equivalentes
$z = &potencia; # Neste caso, a invocacao esta a usar o @_ deste nivel
$w = potencia(base => 7);
print "$x\n$y\n$z\n$w\n";
```

Também é possível ter os primeiros argumentos fixos e só após esses ter os restantes usando um hash. O próximo exemplo demonstra isto mesmo. Neste caso, a subrotina `potencia` tem o primeiro argumento fixo e o segundo com um valor por omissão. Repare que neste caso simples era possível implementar o sistema de outra maneira.

```
sub potencia {
    my ($base, %args) = @_;
    $args{expoente} = 1 unless exists $args{expoente};
    $base ** $args{expoente};
}

$x = potencia 7;
$y = potencia(7, expoente => 2);
print "$x\n$y\n";
```

2 Ficheiros

Existe várias formas de abrir um ficheiro em Perl mas quase todas elas passam pelo comando `open`. Mesmo essa função tem várias formas de utilização (consulte o manual `http://perldoc.perl.org/functions/open.html` para ver todas as formas) mas nós só vamos falar de uma delas, a mais simples.

```
open(FILEHANDLE, MODO, EXPRESSAO);
```

Onde a expressão indica qual o ficheiro a abrir, modo qual é o modo de abertura do mesmo e filehandle onde fica guardado o apontador do ficheiro. Seguem-se alguns exemplos ilustrativos.

```
# Abrir para escrita, se o ficheiro existia perde todo o conteudo
```

```
open $fh, ">", "dados.txt" or die "Nao consegui abrir o ficheiro para escrita: $!";
print $fh "uma linha para o ficheiro\n";
close $fh;

# Abrir para escrita no fim do ficheiro (i.e., conserva o ficheiro actual)
open $fh, ">>", "dados.txt" or die "Nao consegui abrir o ficheiro para escrita: $!";
print $fh "mais uma linha para o ficheiro\n";
close $fh;


# Abrir o ficheiro para leitura; acabar o programa com uma mensagem de erro caso nao se
    consiga abrir o ficheiro
open( $fh, "<", "dados.txt") or die "Nao consegui abrir o ficheiro para leitura: $!";

# Abrir o ficheiro para leitura assumindo o encoding UTF8
open( $fh, "<:utf8", "dados.txt") or die "Nao consegui abrir o ficheiro para leitura: $!";

# Abrir o ficheiro para leitura assumindo que encoding latin1
open( $fh, "<:encoding(Latin1)", "dados.txt") or die "Nao consegui abrir o ficheiro para
    leitura: $!";

while(<$fh>) {
    print;
}
close $fh; # Fechar o ficheiro
```

Repare que a variável `$!` serve para armazenar a mensagem de erro do sistema operativo. É também possível abrir um ficheiro em vários modos (i.e., leitura e escrita ao mesmo tempo). Segue-se uma lista dos modos.

Modo	Descrição
<	Leitura; dá erro se o ficheiro não existe
>	Escrita; não dá erro se o ficheiro não existe e trunca o ficheiro caso ele exista
>>	Concatenação; modo especial de abertura para escrita que não dá erro se o ficheiro não existe e só permite escrever no fim do ficheiro
+<	Abrir para leitura e escrita
+>	Abrir para leitura e escrita; trunca o ficheiro caso ele exista
+>>	Abrir para leitura e concatenação

Eis um exemplo em que se abre um ficheiro em modo latin1 e se imprime em modo utf8:

```
open( $fh, "<:encoding(Latin1)", "/home/rui/CETEMPublicoAnot.001.txt") or die "boo: $!";
binmode(STDOUT, ":utf8");
# pode ir buscar este modulo: utf8::all;

while(<$fh>) {
    print unless $. > 100;
}
```

Vou dar dois exemplos interessantes que poderão ser úteis da utilização do `open` com 3 parâmetros. O primeiro exemplo ilustra a utilização de uma variável para o *filehandle* e a abertura de um ficheiro

temporário[2]. Repare que `undef` é uma palavra reservada que representa o valor indefinido e que neste caso está a ser utilizada no nome do ficheiro.

```
1  open(my $tmp, "+>", undef) or die ...
```

O segundo exemplo mostra a utilização de uma variável como se fosse um ficheiro. Isto usa uma referência (neste caso para a variável `$var` que é algo de que vamos falar depois.

```
1  open($fh, ">", \$var) || die "nao consegui escrever para a variavel $!";
2
3  print $fh "ola mundo\n";
4  for $x (1..10) {
5         print $fh "$x\n";
6  }
7
8  close $fh;
9
10 print "Vamos ver o que tem a variavel: '$var'";
```

[2]Isto é, que só existe enquanto o ficheiro está aberto.

3 Referências

3.1 Definição

Uma referência é um apontador[3]armazenado num escalar. As referências podem ser criadas explícita ou implicitamente.

```
1  sub foo { my ($x, $y) = @_; ($x < $y) ? $x : $y; }
2  $x = 2;
3  @a = qw {programar ler jogar_xadrez jogar_go};
4  %h = (nome => 'rui', gostos => 'como posso colocar aqui uma lista?');
5
6  $ref_a = \@a;
7  $ref_h = \%h;
8  $ref_foo = \&foo;
9
10 # Formas pouco limpas
11 print "@$ref_a $$ref_a[1] $$ref_h{gostos}\n";
12 print &$ref_foo(2,3);
13
14 # Formas recomendadas
15 print "@$ref_a $ref_a->[1] $ref_h->{gostos}\n";
16 print $ref_foo->(2,3);
```

Para a criação implícita podemos usar a seguinte forma.

```
1  $ref_array = [1, 2, a, b, 3];
2  $ref_hash = {a => 1, b => 2};
3
4  $bd = {100 => {nome => 'rui', gostos => [ler, programar, puzzles, jogos]}, 101 => {nome =>
       'alberto', gostos => [ler, programar,dormir, comer]}};
5  print "$bd->{100}{gostos}[1]\n";
6  print "$ref_array $ref_hash $bd\n";
```

Repare que ao usar os parentesis rectos estamos a criar uma referência para um array enquanto que ao usar chavetas criamos uma referência para um hash. Repare como a linha 5 imprime algo do género:

```
ARRAY(0x9015818) HASH(0x9035680) HASH(0x90357f0)
```

Utilizando uma sintaxe semelhante no valor que retorna pode-se usar a função `ref` para saber de que tipo de referência estamos a tratar.

```
1  $ref_escalar = \("ola"); #Uma referencia para um escalar
2  $ref_array = [1, 2, a, b, 3];
3  $ref_hash = {a => 1, b => 2};
4
5  print ref($ref_escalar) . "\t";
6  print ref($ref_array) . "\t";
7  print ref($ref_hash) . "\n";
```

[3]Isto é, um endereço para uma zona de memória que armazena um valor de um tipo de dados qualquer

Que deve imprimir

```
SCALAR ARRAY HASH
```

Existe um módulo muito útil, o `Data::Dumper` que permite imprimir o valor de uma referência recursivamente.

```
use Data::Dumper;
$bd = {100 => {nome => 'rui', gostos => [ler, programar, puzzles, jogos]}, 101 => {nome =>
    'alberto', gostos => [ler, programar,dormir, comer]}};
print Dumper($bd);
```

Que deverá imprimir algo com este aspecto:

```
$VAR1 = {
          '101' => {
                     'nome' => 'alberto',
                     'gostos' => [
                                   'ler',
                                   'programar',
                                   'dormir',
                                   'comer'
                                 ]
                   },
          '100' => {
                     'nome' => 'rui',
                     'gostos' => [
                                   'ler',
                                   'programar',
                                   'puzzles',
                                   'jogos'
                                 ]
                   }
        };
```

Pode-se criar directamente uma referência para uma subrotina:

```
$coderef = sub { $_[0] ** 2 };
```

Eis um exemplo de uma forma de usar referências para fazer uma programação de mais alto nível:

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y!\n"; };
}
$h = newprint("Howdy");
$g = newprint("Greetings");

&$h("world");
&$g("earthlings");
```

Que imprime

```
Howdy, world!
Greetings, earthlings!
```

3.2 HC SVNT DRACONES

Algo que podera *eventualmente* ser útil, mas que é bastante **perigoso**, é a utilização de referências simbólicas (que não é permitido quando se usa `use strict;` ou mais específicamente `use strict 'refs';`.

```
$name = "foo";
$$name = 1; # Atribui a $foo
${$name} = 2; # Atribui a $foo
${$name x 2} = 3; # Atribui a $foofoo
$name->[0] = 4; # Atribui a $foo[0]
@$name = (); # Atribui a @foo
```

3.3 Forma Recomendada de Manipular referências

Visto que as referências se tornam facilmente difíceis de perceber, o melhor é adotar um estilo de programação. Para isso, podem-se escrever as referências que vimos anteriormente de uma forma mais legível.

```
$bd = {100 => {nome => 'rui', gostos => [ler, programar, puzzles, jogos]}, 101 => {nome =>
    'alberto', gostos => [ler, programar,dormir, comer]}};

print "@{$bd->{100}{gostos}}\n"; # A lista com todos os gostos do rui
print "$bd->{100}{gostos}[1]\n"; # O segundo gosto do rui
```

De salientar que esta é a forma como sugerimos que programem referências, deixando as formas anteriores para código mais *sujo*.

Vamos voltar ao exemplo da construção do índice. Neste caso queremos construir uma lista das linhas em que cada palavra aparece. Para isso poderíamos fazer algo assim:

```
use strict;
use warnings;
use Data::Dumper;
$Data::Dumper::Terse = 1;    # don't output names where feasible
$Data::Dumper::Indent = 1;   # level of pretty print
$Data::Dumper::Sortkeys = 1; # sort hash keys

my $linhas;
while(<>) {
       for my $palavra (split) {
              # $.: o n. da linha atual
              push @{$linhas->{$palavra}}, $.;
       }
}

```

```
print Dumper($linhas);
print "\n";
```

Ou então assim:

```
use strict;
use warnings;
use Data::Dumper;
$Data::Dumper::Terse = 1; # don't output names where feasible
$Data::Dumper::Indent = 1; # level of pretty print
$Data::Dumper::Sortkeys = 1; # sort hash keys

my $linhas;
while(<>) {
    for my $palavra (split) {
        # $.: o n. da linha atual
        $linhas->{$palavra}{$.} = ();
    }
}

print Dumper($linhas);
print "\n";
```

Entre as duas alternativas, qual é a melhor?

4 Exercícios

1. Faça os últimos exercícios da ficha anterior utilizando referências onde elas possam ser úteis;
2. Faça um programa que leia uma matriz e imprima a sua transposta (assume-se que os valores de cada linha estão separados por espaços);
3. Faça uma função que receba como parâmetro o nome de dois ficheiros (cada um contendo uma matriz) e imprima a matriz que resulta da multiplicação de ambas as matrizes (ou um erro caso não seja possível;
4. Faça um programa que leia um ficheiro com linhas do género (isto é, os valores dos vários campos são separados por tabs)

   `<nome da pessoa>\t<nome da UC>\t<classificação>`

   e que armazene essa informação. Repare que cada pessoa pode ter mais do que uma classificação em cada UC. Depois de ler o ficheiro o programa deverá ter uma estrutura de dados com todas as classificações que cada pessoa teve a cada UC. Isto deverá ser armazenado numa única referência.

   ```
   joana li2 17
   carlos pl 16
   joana li2 12
   carlos pi 13
   luisa tc 16
   ```

5. Faça uma função chamada `dump` que recebe uma referência e imprime o seu conteúdo de uma forma semelhante ao módulo `Data::Dumper`; alternativamente, a sua função pode gerar `html` (ou LaTeX).

6. Faça um conjunto de funções que lhe permitam manipular *multi sets*. Um multi set deverá representar várias ocorrências de um mesmo elemento. Assim,

   ```
   ms_union({a=> 3, b=>4},  {a=>4, c=>2})
   ```

   deveria devolver a referência equivalente a:

   ```
   {a=> 7, b=>4 ,c=>2}
   ```

   Pense em criar as funções que façam sentido incluindo imprimir, calcular reuniões, interseções, etc.