Departamento Estudos e Investigação em Tecnologias de Informação e Sociedade

kriativ.tech

Edição Nº 6 – 29 de Maio de 2018
ISSN Print: 1646-9976 | ISSN Online: 2184-223X |
DOI: https://doi.org/10.31112/kriativ-tech-2018-01-16
http://www.kriativ-tech.com http://www.kriativ-tech.pt

Diferentes Aproximações em Linguagens de Programação

José Câmara
Professor Adjunto do ISTEC
ISTEC – Departamento de Estudos e Investigação em Tecnologias de Informação e Sociedade
josecamara@netcabo.pt

Resumo: faremos uma breve resenha das diferentes técnicas de subprogramação e acesso aos dados utilizados nos subprogramas em diferentes linguagens de programação de referência, dado que há linguagens que implementam apenas uma aproximação, mas há outras que requerem duas aproximações. Com este artigo, tentaremos fazer um esclarecimento sobre estes aspetos.

Palavras-chave: *Subprograma, Função, Sub-rotina, Procedimento, Passagem de Parâmetros por Valor, Passagem de Parâmetros por Referência.*

Abstract: *we will briefly review the different subprogramming techniques and data access used in subprograms in different reference programming languages, since there are languages that implement only one approach, but there are others that require two approaches. With this article, we will try to clarify these aspects.*

Keywords: *Subprogram, Function, Subroutine, Procedure, Passing Parameters by Value, Passing Parameters by Reference.*

I. Introdução

Em programação computacional, existem duas formas de implementação de subprogramas [3] [6], no desenvolvimento de programas, as quais são funções (*functions*) [1] e procedimentos (*procedures*) [6], que constituem técnicas bem distintas em natureza e aplicação. No sentido restrito, os procedimentos também são conhecidos por sub-rotinas (*subroutines*), já que no sentido lato todos os suprogramas são procedimentos.
Estas duas técnicas podem ou não estar ambas disponíveis no mesmo programa, já que dependem da conceção e arquitetura da linguagem de programação subjacente, ou seja, se esta optou ou não, aquando da sua criação, em ter apenas uma ou as duas capacidades.
Há linguagens de programação que aplicam ambas as técnicas, tais como o Visual Basic.net [5] [6] e o Delphi, entre outras. Também a linguagem SQL (*Structured Query Language*), que é uma linguagem especificamente orientada para acessos e consultas a Bases de Dados, aplica ambas as técnicas [7] [8].
Mas há linguagens de programação, muito populares e relevantes, tais como C/C++ [2], Java [11] [12], C#, JavaScript e PHP [9] [10], entre outras, que somente aplicam uma técnica, a técnica das funções (ou métodos, como também são conhecidas).
Tanto as funções como os procedimentos (*procedures*) servem para desenvolver programas bem organizados e estruturados,

subdivididos em tarefas específicas, as quais podem ser invocadas em qualquer parte do programa, evitando assim repetições e redundâncias de código, conducentes à construção de programas com maior clareza, desempenho, capacidade de depuração e manutenção.

Para poder esclarecer, da melhor forma, a razão porque existem linguagens de programação que disponibilizam somente uma técnica, funções, ou as duas técnicas, funções e procedimentos (*procedures*) na implementação de subprogramas, no processo de desenvolvimento de programas, é necessária a introdução ou a revisão abaixo de alguns princípios acerca das funções e procedimentos (*procedures*) bem como das suas envolvências.

Ao acabar de me referir em "introdução ou revisão de alguns princípios" era mais no caso do leitor conhecer a natureza da programação sem ser programador, ou estar simplesmente interessado no tema, sendo ou não da área da programação, ou ser mesmo programador, mas que nunca teve oportunidade ou necessidade de utilizar procedimentos (*procedures*) em programação.

II. Arquitetura e Objetivo das Funções e das P*rocedures*

Sintaxe Tipo dos Procedimentos (*Procedures*)

<nome do procedimento> (<lista de parâmetros formais>)
início
    <declaração de variáveis locais>
    corpo do procedimento;
fim;

Sintaxe Tipo das Funções

<tipo básico> <nome da função> (<lista de parâmetros formais>)
início
    <declaração de variáveis locais>
    corpo da função
    retorno <valor de retorno>;
fim;

Tanto as funções como os procedimentos são blocos de código, devidamente nomeados, destinados à execução de determinadas tarefas, como, por exemplo, cálculos, leitura ou impressão de dados. São invocados ou chamados, sempre que necessário, a partir de uma parte de código do programa que os contém. Normalmente, recebem dados da parte "chamante" sob a forma de parâmetros ou argumentos. Os parâmetros da parte "chamante" têm o nome "parâmetros ou argumentos atuais" e os parâmetros da parte chamada têm o nome "parâmetros ou argumentos formais". Os parâmetros atuais podem ser constantes, valores e nomes de variáveis ou objetos. Os parâmetros formais são nomes de variáveis que ficam associadas correspondentemente aos parâmetros atuais, mas que requerem uma declaração formal das mesmas, incluindo os seus tipos de dados e uma indicação implícita ou explícita de como os dados são acedidos, com ou sem afetação dos seus valores originais [2] [3] [6] [12].

Apesar destas semelhanças, acima descritas, entre as funções e procedimentos (*procedures*), tratam-se, na realidade, de dois tipos de subprogramas completamente distintos em natureza e aplicação.

As funções retornam ao código "chamante" um valor e têm um tipo de dados associado à sua declaração formal, que deve estar de acordo com o tipo de dados a devolver. Se o valor da devolução for, por exemplo, do tipo inteiro, então a função é do tipo inteiro. Nem sempre as funções obrigatoriamente retornam valores, como, por exemplo, uma função destinada à mera impressão de cálculos, mas têm sempre essa capacidade de devolução. Quando uma função não retorna qualquer valor é do tipo *void* (têm que ter sempre um tipo associado à sua declaração).

Quanto ao acesso e à afetação dos seus dados passados como parâmetros podem ou não ser alterados os seus valores originais de acordo com uma determinada indicação a ser analisada um pouco mais à frente neste artigo. Essa indicação reside no facto dos seus parâmetros poderem ser variáveis normais e variáveis com

um endereço de memória associado. Se tiverem um endereço de memória associado, então os seus valores originais podem ser alterados [1] [2].

Por sua vez, os procedimentos (*procedures*/*subroutines*) não retornam quaisquer valores nem têm um tipo de dados associado à sua declaração, já que são subprogramas especificamente desenhados com a capacidade de poderem obter vários cálculos, os quais podem ser de diferentes tipos de dados, e armazená-los diretamente na memória, nas localizações das variáveis que passaram os seus endereços para esse efeito. Os seus parâmetros são assim variáveis normais e variáveis com um endereço de memória associado [3] [4] [6]. Neste aspeto, a diferença relativamente às funções é que estas podem retornar um valor e as *procedures* não.

III. Definições e Princípios

Variáveis Globais

São as variáveis declaradas fora de qualquer subprograma, sendo acessíveis e partilháveis em qualquer parte do programa. O seu escopo é assim global. Se um subprograma alterar o valor de uma destas variáveis, o seu novo valor será assim visível e alterado para todo o programa. Dada a sua natureza, estas variáveis não são passadas como parâmetros aos subprogramas.

Variáveis Locais

São as variáveis declaradas dentro dos subprogramas, sendo somente acessíveis dentro do subprograma onde foram declaradas. O seu escopo é assim local. sistemas, aplicações ou utilizadores finais interagem com esses referidos recursos [11].

Passagem de Argumentos por Valor

Quando é passada somente uma cópia do valor da variável para o subprograma.
Qualquer alteração feita no parâmetro não é refletida exteriormente ao subprograma [1].

Passagem de Argumentos por Referência

Quando o parâmetro passado ao subprograma contém o endereço da memória da variável.
A variável comporta-se assim como uma variável global. Qualquer alteração feita ao valor de uma destas variáveis fica assim afetado e acessível a todo o programa [1].
Os subprogramas do tipo procedimento (*procedures*) servem-se desta capacidade da passagem de argumentos por Referência para poderem obter diversos cálculos e armazená-los diretamente na memória ao dispor. Como não retornam valores e podem colocar nas memórias das variáveis associadas aos seus endereços que receberam nos parâmetros por Referência não podem ter um tipo de dados na sua declaração formal.
Por conseguinte, nas linguagens que adotam as duas técnicas na construção dos seus subprogramas, os procedimentos (*procedures*), para além da passagem de argumentos por Valor, são especialmente dedicados à passagem de argumentos por Referência, podendo assim obterem resultados de vários cálculos, que podem ser de diferentes tipos de dados e colocá-los na memória para acesso global.
Nas linguagens que adotam apenas uma técnica, a das funções, na construção dos seus subprogramas, utilizam passagens de argumentos por Valor e por Referência.
A forma de indicar como os argumentos são passados por Valor ou por Referência, por vezes difere de uma linguagem para outra, conforme será analisado seguidamente.

IV. Subprogramação em algumas Linguagens Populares

Visual Basic.NET

Esta Linguagem dispõe de funções e procedimentos (*procedures*) [4] [5] [6].
Na Linguagem de Programação Visual Basic.NET existem as cláusulas específicas ByVal (ex.: ByVal x As Integer) e ByRef (ex.: ByRef y As Double) para indicar se os seus argumentos estão a ser passados por Valor ou por Referência.

Um exemplo tipo de uma *procedure/subroutine* é o seguinte:

Sub calcMedMaxMinVetor(**ByRef** media As Single, **ByRef** max As Integer, **ByRef** min As Integer, **ByVal** ParamArray valores() As Integer)

> Aqui é o local do código para calcular os valores do máximo, mínimo e média do vetor valores.

media = resultado obtido acima no cálculo do valor da média do Array
max = resultado obtido acima no cálculo do valor máximo do Array
min = resultado obtido acima no cálculo do valor mínimo do Array

End Sub

No exemplo acima, o procedimento recebe por Valor a cópia dos valores de um vetor de inteiros, calcula os valores da média, máximo e mínimo do vetor, atribuindo-os às respetivas variáveis media, max e min, as quais ficam assim acessíveis globalmente (passagem por Referência). O procedimento obteve assim vários cálculos no mesmo subprograma.

Um exemplo tipo de uma ***function*** é o seguinte:

Function *calcMed (**ByVal** ParamArray valores() As Integer)* ***As Single***

> ***Aqui é o local do código para calcular o valor da média do vetor valores.***
>
> ***Return (Valor_da_media)***

End Function

No exemplo acima, a função recebe por Valor a cópia dos valores de um vetor de inteiros, calcula o valor da média do vetor e devolve-o, através do comando *Return* ao programa "chamante". A Função obteve assim um cálculo do tipo *Single* (vírgula flutuante) e a função é assim do tipo *Single*.

Linguagem SQL (PL/SQL, MySQL)

Esta Linguagem dispõe de funções e procedimentos (*procedures*) [7] [8].
Na construção dos seus subprogramas, conhecidos por *Stored Procedures*, usa as cláusulas IN, OUT e INOUT [7] [8] na declaração formal dos seus parâmetros. Um parâmetro IN é do tipo *Read Only* (Passagem por Valor). Um parâmetro OUT permite o conteúdo de uma variável ser conhecido fora do subprograma (Passagem por Referência), após a execução deste. Um parâmetro *INOUT* permite o conteúdo de uma variável de entrada ser alterado dentro do subprograma e, depois, ser conhecido fora do subprograma.

Linguagem Java

A linguagem Java dispõe somente de funções [11] [12].
A linguagem Java, que dispõe somente de funções, tem as seguintes regras:
Passagem de Argumentos Por Valor:
Variáveis do tipo *int, float, boolean, char* são passadas por Valor.

Passagem de Argumentos Por Referência:
Os objetos e *arrays* são passados por Referência.

Linguagem PHP

Esta Linguagem dispõe somente de funções [8] [9].
Na construção dos seus subprogramas usa o carácter &, antes de uma variável, que significa o endereço dessa variável, na declaração formal dos seus parâmetros sempre que seja pretendido a passagem de um parâmetro por Referência.

Linguagem C/C++

Esta linguagem dispõe somente de funções [1] [2] [3].
A Linguagem C/C++, que também dispõe somente de funções, precursora de muitas linguagens modernas e relevantes, usa uma

técnica diferente, conhecida por ponteiros ou apontadores (*pointers*).

Dada a relevância desta Linguagem e a natureza do conceito de apontadores e também devido ao facto do leitor poder não o conhecer na prática, segue-se uma introdução e uma aplicação.

V. A Particularidade da Linguagem C/C++ na Subprogramação

Ponteiros/Apontadores na Linguagem C/C++ [1] [2] [3]:
Um ponteiro é uma variável que guarda o endereço de outra variável, permitindo assim o acesso a esta, à sua localização de memória, de uma forma indireta.

Declaração de um ponteiro:
tipo_de_dados_da_variável_que_aponta_para *nome_ponteiro;

Exemplo:

```
float *p1;
```

No exemplo acima, significa que p1 é um ponteiro que aponta para uma variável do tipo *float*. Ou seja, p1 guarda o endereço de uma variável do tipo *float* e pode assim acedê-la indiretamente.

Particularidades dos ponteiros:

&
Antes de uma variável (préfixo) refere-se ao próprio endereço dessa variável.

*
Antes de ponteiro, mas fora da sua declaração, refere-se a um acesso indireto (o ponteiro acede ao conteúdo da variável que está a apontar para).

Exemplo:

```
int x;          ---- declara x como variável do tipo inteiro
x = 10;         ---- atribuição do valor 10 à variável x

int *p;         ---- p é um ponteiro que aponta para uma variável inteira
p = &x;         ---- p guarda o endereço de x (p aponta para x)

printf("\n %d", *p);    ---- mostra no ecrã o valor de x (10)
```

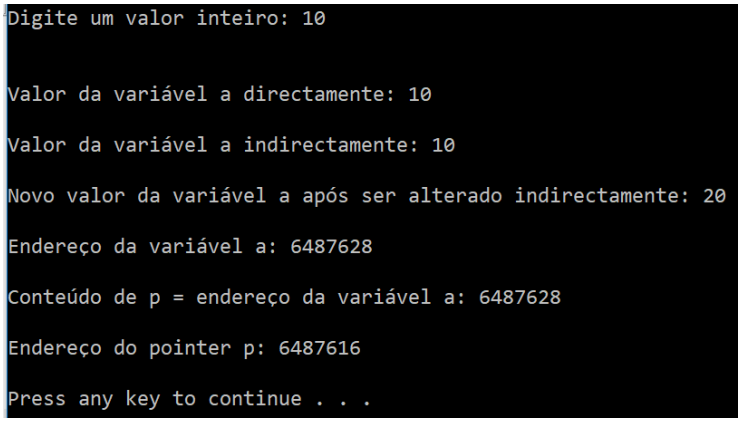
Caso Prático 1:

O seguinte programa mostra como um ponteiro pode aceder à localização da memória de uma dada variável e ler ou alterar o seu valor.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main()
{
setlocale(LC_ALL, "Portuguese");
int a; // declara variável a do tipo inteira

int *p; // declara um ponteiro para inteiros
p = &a;          // p guarda o endereço da variável a

printf("\n\nDigite um valor inteiro: ");
fflush(stdin); // limpa a buffer do teclado
scanf("%d", &a); // leitura do valor
printf("\n\nValor da variável a acedida diretamente: %d", a);
printf("\n\nValor da variável a acedida indiretamente: %d", *p);

*p = 20; // altera indiretamente o valor de a

printf("\n\nNovo valor da var a após ser alterado indiretamente: %d", *p);

// ver os endereços reais onde este programa está a ser executado

printf("\n\nEndereço da variável a: %ld", (unsigned long) &a);
printf("\n\nConteúdo de p = endereço da variável a: %ld", (unsigned long) p);
```

```
printf("\n\nEndereço do pointer p: %ld", (unsigned long) &p);
printf("\n\n");
system("pause");
return 0;
}
```

O resultado produzido abaixo é ilustrativo:



Caso Prático 2:

O seguinte programa demonstra como um ponteiro numa função altera o conteúdo de uma variável, por Referência. O argumento actual passa à função **alteraValor()** o endereço da variável **x**, incluindo a indicação do endereço, através de **&**, antes do nome da variável, conforme abaixo representado:

alteraValor(&x); // chamada à função

Por sua vez, o argumento formal declara um ponteiro para um valor inteiro, recebendo o endereço do seu valor e a função efetivamente altera o valor de x para 33. A função é do tipo *void*, pelo que não retorna qualquer valor, não precisando já que o novo valor de x se encontra acessível globalmente na memória, conforme abaixo ilustrado:

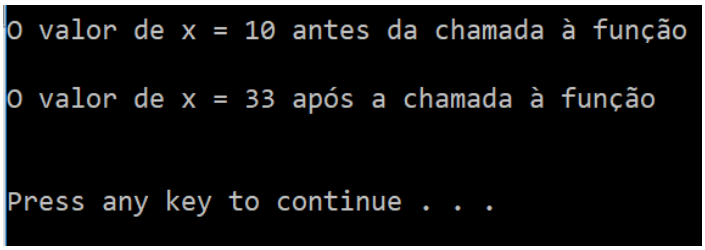
```
void alteraValor(int *p)
{
    *p = 33; // o conteúdo da localização apontado pelo ponteiro p é alterado de 10 para 33
}
```

Finalmente, a parte de código "chamante" imprime o valor de x após a chamada à função, comprovando efetivamente a sua alteração de 10 para 33.

Vejamos o código:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>


void alteraValor(int *p)
{
*p = 33; // o conteúdo da localização apontado pelo ponteiro p é alterado
}

int main()
{
 setlocale(LC_ALL, "Portuguese");
 int x = 10;
 printf("\n\nO valor de x = %d antes da chamada à função", x);
 alteraValor(&x); // chama a função passando o endereço da variável x
 printf("\n\nO valor de x = %d após a chamada à função", x);
 printf("\n\n\n");
 system("pause");

 return 0;
}
```

O resultado produzido abaixo é ilustrativo:



Desta forma, a Linguagem C/C++, através de ponteiros, pode perfeitamente aceder e alterar efetivamente os conteúdos das variáveis passadas como parâmetros às funções, permitindo dispensar os procedimentos (*procedures*).

VI. Conclusão

Conclui-se o presente artigo, em esclarecimento final, que, no que respeita à implementação de subprogramas, reafirma-se que há linguagens de programação que disponibilizam somente funções e linguagens de programação que disponibilizam funções e procedimentos (*procedures*).
No entanto, as linguagens de programação que somente disponibilizam funções têm princípios, capacidades e técnicas, de poderem desempenhar ambas as funcionalidades, ou seja, de funções e *procedures*.

XIV. Referências

[1] B. Kernighan, D. Ritchie, "The C Programming Language-Second Edition", AT&T Bell Laboratories, USA, 1988.
[2] L. Damas, "Linguagem C", Lisboa, FCA, 1999.
[3] H. Deitel, P. Deitel, "C: How to Program", Prentice Hall, USA, 1994.
[4] N. Nina, "Visual Basic.NET", Lisboa, FCA, 2003.
[5] H. Loureiro, "Visual Basic 2010", Lisboa, FCA, 2010.
[6] A. Carriço, J. Carriço, "Programação em Visual Basic.NET", Lisboa, ISTEC-Academia Software, 2002.
[7] L. Damas, "SQL", Lisboa, FCA, 2005.
[8] F. Tavares, "MySQL", Lisboa, FCA, 2015.
[9] C. Serrão, J. Marques, "Programação com PHP 5.3", Lisboa, FCA, 2009.
[10] F. Tavares, "Desenvolvimento de Aplicações em PHP", Lisboa, FCA, 2015.
[11] H. Deitel, P. Deitel, "Java: How to Program", Prentice Hall, USA, 1999.
[12] F. Martins, "JAVA 6: Programação Orientada pelos Objectos", Lisboa, FCA, 2009.