

Subprogramas

Gustavo Scaloni Vendramini
Guilherme José Henrique
Sean Carlisto de Alvarenga
Vinícius Fernandes de Jesus

23 de setembro de 2013

<i>SUMÁRIO</i>	2
----------------	---

Sumário

1	Subprogramas Como Parâmetro	3
2	Chamar Subprogramas Indiretamente	4
3	Sobrecarga de Subprogramas	5
4	Subprogramas Genéricos	6
4.1	Template em C++	6
4.2	Métodos Genéricos em Java	7
4.3	Métodos Genéricos em C#	7

1 Subprogramas Como Parâmetro

Em muitas ocasiões temos a necessidade de passar um subprograma através de um parâmetro. A ideia é interessante e simples, mas gera duas complicações em termos de implementação.

Primeiro, temos a complicação que consiste na maneira de realizar a checagem de tipo (*type checking*, `??`) do subprograma passado por parâmetro. Em C e C++, onde a passagem de subprogramas é feita através de ponteiro para função, essa checagem é feita pelo tipo do ponteiro.

A outra complicação ocorre em linguagens de programação que permitem subprogramas aninhados. O problema refere-se a qual ambiente de referência o subprograma passado por parâmetro terá. Nessa situação, há três tipos possíveis:

Shallow Binding: O ambiente é o local onde o subprograma é chamado.

Deep Binding: O ambiente refere-se onde o subprograma foi definido.

Ad Hoc Binding: O ambiente condiz com o local que o subprograma foi passado por parâmetro.

Como exemplo, considere a listagem 1, cuja syntax é de JavaScript. O subprograma `sub2()` apenas imprime o valor da variável `x`, porém, seu valor depende do ambiente de referência utilizado.

```
1 function sub1() {  
2   var x;  
3   function sub2() {  
4     alert(x);  
5   };  
6   function sub3() {  
7     var x;  
8     x = 3;  
9     sub4(sub2);  
10  };  
11  function sub4(subx) {  
12    var x;  
13    x = 4;  
14    subx();  
15  };  
16  x = 1;  
17  sub3();  
18 };
```

Listing 1: Código retirado de [5]

Caso a listagem em questão utilize o ambiente de referência *Shallow Binding*, o valor impresso seria 4. Caso o ambiente *Deep Binding* fosse escolhido, o valor impresso seria 1. Já para *Ad Hoc Binding*, o valor seria 3.

Segundo Robert W. Sebesta, a abordagem *Ad Hoc Binding* nunca foi implementada [5].

2 Chamar Subprogramas Indiretamente

Há momentos, durante a programação de um software, em que se torna necessário chamar subprogramas de forma indireta. Isso ocorre quando o subprograma a ser chamado é somente conhecido em tempo de execução, como eventos disparados por bibliotecas de interface gráfica e funções de *callback*.

Em C e C++, podemos utilizar ponteiro para função para chamar um subprograma conhecido em tempo de execução [1, 5]. Para utilizar essa técnica, primeiro temos que declarar uma função, como por exemplo:

```
int sum(int a, int b)
{
    return a + b;
}
```

A seguir, temos que declarar um ponteiro com a mesma assinatura da função escolhida. Como no nosso exemplo (função *sum*) a função possui dois parâmetros e um retorno do tipo *int*, temos um ponteiro como mostrado:

```
int (*sum_pointer)(int, int);
```

Em seguida, é necessário atribuir a função em questão para o ponteiro declarado. Em nosso exemplo:

```
sum_pointer = &sum;
```

Por fim, basta invocar a função, como da seguinte maneira:

```
(*sum_pointer)(1,2);
```

Em C#, podemos referenciar métodos em forma de objetos, através do uso de *delegate* [3, 5], o que torna muito poderoso e flexível. Para fazer uso do mesmo, precisamos declarar um *delegate* para um determinado protocolo de função, como:

```
public delegate int SumDelegate(int a, int b);
```

Podemos instanciar um *SumDelegate* passando por parâmetro para seu construtor o nome de uma função cuja declaração tenha o mesmo protocolo. Suponha a existência da função chamada *sum* que respeite essa assinatura, podemos então instanciar *SumDelegate* como segue:

```
SumDelegate sumDelegate= new SumDelegate(sum);
```

Para executar o *delegate* em questão, fazemos da seguinte forma:

```
sumDelegate(2,3);
```

3 Sobrecarga de Subprogramas

A maioria das linguagens de programação permitem sobrecarga de subprogramas, que consiste em subprogramas com o mesmo nome, mas com parâmetros diferentes, seja por quantidade, ordem ou tipo.

Essa técnica é utilizada quando temos subprogramas que tem o mesmo objetivo, porém fazem uso de parâmetros diferentes. Em C++, C# e Java a sobrecarga de construtores é a mais utilizada por desenvolvedores [6].

Conforme apontado por Sebesta, as linguagens Ada, Java, C++, C# e F# são exemplo de linguagens que permitem sobrecarga de operadores [5].

4 Subprogramas Genéricos

No desenvolvimento de software, o reuso do código fonte é um fator importante para aumentar a produtividade e diminuir gastos. Uma maneira de conseguir esse reuso seria fazer um único subprograma funcionar para qualquer tipo de dados. Com isso, por exemplo, não precisamos criar um subprograma de ordenação para cada tipo de dado existente.

Um subprograma polimórfico é algo que pode nos oferecer esse tipo de reuso. Segundo Sebesta [5], há três tipos de polimorfismo, sendo eles:

Ad Hoc: Consiste na sobrecarga de subprogramas, conforme mostrado na seção 3.

Subtipo: Específico para linguagens orientadas a objeto, que não está dentro do escopo deste trabalho, mas pode ser encontrado em [8].

Paramétrico: Baseia-se do uso de tipos genéricos, onde esse tipo genérico pode assumir qualquer tipo de dado. Esta seção irá abordar esse tipo de polimorfismo.

Subprogramas com polimorfismo paramétrico são subprogramas que utilizam tipos de dados genéricos, o que possibilita seu reuso para cada tipo de dado. A seguir, iremos exemplificar o uso dessa técnica em C++, Java e C#.

4.1 Template em C++

Em C++, funções genéricas são obtidas através do uso de *templates* [2]. Primeiro, é necessário declarar um template, fazendo-o da seguinte forma:

```
template <template parameters>
```

Um *template parameter* pode ter uma das seguintes formas

```
class identifier
typename identifier
```

Segundo Sebesta [5], a palavra chave *class* é utilizada para especificar tipos e *typename* é necessária para especificar um valor absoluto.

Como exemplo, vamos declarar um função template, que retorna o maior valor entre dois dados. A declaração é feita da seguinte forma

```
template <class myType>
myType GetMax (myType a, myType b) {
    return (a>b?a:b);
}
```

Para chamarmos *GetMax* para dois inteiros, fazemos da seguinte maneira

```
GetMax<int> (1,2);
```

Para trocar o tipo dos parâmetros da função *GetMax*, basta trocar o tipo de dados ao chamar a função em questão (por exemplo, de *int* para *float*).

Templates podem ser utilizado em classes, podem ser aninhados e também podem possuir especializações. Explicar essas funcionalidades não faz parte do escopo deste trabalho, mas isso pode ser obtido em [2].

4.2 Métodos Genéricos em Java

Em Java, podemos fazer o uso de métodos genéricos, sendo similar aos *templates* de C++. Um exemplo de um método genérico em Java é como segue

```
public static <T> T doIt(T[] list) {  
    ...  
}
```

Onde é definido o método *doIt* que recebe como parâmetro uma lista de tipos genéricos. O método *doIt* pode ser chamado passando uma lista do tipo *String* da seguinte forma

```
doIt<String>(myList);
```

Diferentemente de C++, os tipos genéricos em java devem ser uma classe, não podendo ser um valor absoluto ou um tipo primitivo. Além do mais, Java permite que limitamos os tipos genéricos, como informando de qual classe ele deve herdar ou qual interface deve implementar [5].

Tipos genéricos em Java nos fornece muitos outros recursos (como *Wildcards*), que não abordados neste trabalho, mas que podem ser encontrados com detalhes em [7].

4.3 Métodos Genéricos em C#

Métodos genéricos em C# tem as mesmas funcionalidades de Java, porém não suportam *wildcards*. Uma diferença em C# é que o tipo do argumento pode ser omitido caso o compilador poder inferir o tipo [5].

Mais informações e exemplos sobre tipos genéricos em C# podem ser encontrados em [4].

Referências

- [1] Alex Allain. Function pointers in c and c++. <http://www.cprogramming.com/tutorial/function-pointers.html>. Acessado em: 22/09/2013.
- [2] C++ Documentation. Templates. <http://www.cplusplus.com/doc/tutorial/templates/>. Acessado em: 23/09/2013.
- [3] Microsoft Developer Network. Delegates (c# programming guide). [http://msdn.microsoft.com/en-us/library/ms173171\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/ms173171(v=vs.90).aspx). Acessado em: 22/09/2013.
- [4] Microsoft Developer Network. Generics (c# programming guide). <http://msdn.microsoft.com/en-us/library/512aeb7t.aspx>. Acessado em: 23/09/2013.
- [5] Robert W. Sebesta. *Concepts of Programming Languages (10th Edition)*. Addison-Wesley, 2012.
- [6] The Java Tutorials. Defining methods. <http://docs.oracle.com/javase/tutorial/java/java00/methods.html>. Acessado em: 23/09/2013.
- [7] The Java Tutorials. Lesson: Generics. <http://docs.oracle.com/javase/tutorial/java/generics/index.html>. Acessado em: 23/09/2013.
- [8] The Java Tutorials. Polymorphism. <http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>. Acessado em: 23/09/2013.