Subprogramas

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

8 de outubro de 2013

Sumário

1 Ambiente de Referênciamento Local

Variáveis que são definidas dentro de subprogramas são chamadas de variáveis locais, porque o acesso a elas normalmente é restrito ao subprograma na qual são definidas [6]. Essas variáveis podem ser variáveis estáticas ou variáveis dinâmicas na pilha.

1.1 Variáveis Locais

Variáveis estáticas são vinculadas a células de memória antes do inicio da execução do programa e continuam até o término de sua execução.Uma das vantagens de variáveis estáticas é que elas possuem um endereçamento direto na memória como mostra a figura 1, permitindo assim um acesso mais rápido em relação a variáveis com endereçamento indireto. Outra vantagem, é por não causarem sobrecarga em tempo de execução para alocação de desalocação. Uma desvantagem das variáveis estáticas é por não se comportarem bem em subprogramas recursivos. Considere a listagem 1 de uma função em C, onde a saída após a execução do programa será 6, 12 e 18.

```
1  #include <stdio.h>
2
3  int sum ( int arr[], int n )
4  {
5      static int result = 0;
6      if (n == 0)
7          return result ;
8      else {
9          result += arr[n - 1];
10         sum(arr, n - 1);
11      }
12 }
13
14 int main(void) {
15     int array[5] = {1,2,3,4,5};
16     printf("%d\n", sum(array, 3));
17     printf("%d\n", sum(array, 3));
18     printf("%d\n", sum(array, 3));
19     return 0;
20 }
```

Listing 1: Programa recursivo com variável estática

Variáveis dinâmicas na pilha, são vinculadas ao armazenamento quando o subprograma inicia sua execução e desvinculadas do armazenamento quando ele se encerra. Variáveis desse tipo permitem maior flexibilidade para subprogramas (por exemplo, subprogramas recursivos), além de permitir o compartilhamento de parte do armazenamento para variáveis locais de todos subprogramas (não ativos ao mesmo tempo). Porém, variáveis dinâmicas na pilha possuem algumas desvantagens como o enderaçamento indireto 1 e o tempo gasto para inicialização (se necessário), alocação e desalocação.

Linguagens como ALGOL 60 e suas linguagens descendentes, possuem variáveis locais dinâmicas na pilha. Em funções em C as variáveis são dinâmicas na pilha a menos que sejam especificamente declaradas como **static**. Subprogramas Pascal e Ada e métodos em C++, Java, C# têm somente variaveis dinâmicas na pilha.

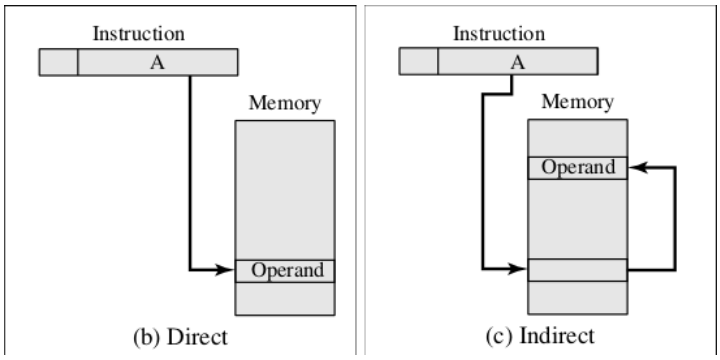

Figura 1: Endereçamento direto e indireto [7]

1.2 Aninhamento de Subprogramas

Linguagens como ALGOL 68, Pascal e Ada permitem aninhamento de subprogramas, assim como linguagens mais recentes como JavaScript, Python e Lua. Linguagens descententes de C não permitem aninhamento. O trecho de código a seguir mostra um exemplo de aninhamento de funções em JavaScript.

```
function hipotenusa(a, b) {
   function quadrado(x) {
      return x * x;
   }
   return Math.sqrt(quadrado(a) + quadrado(b));
}
```

2 Métodos de Passagem de Parâmetros

A passagem de parâmetros é uma das formas do subprograma ter acesso aos dados que irá processar. Os parâmetros podem ser utilizados para apresentar valores ao subprograma e recuperar seu valor após termino do subprograma. Existem dois tipos de parâmetros, os parâmetros no cabeçalho do subprograma são chamados parâmetros formais e os parâmetros apresentados numa chamada ao subprograma são chamados de parâmetros reais.

Exemplo de cabeçalho de subprograma. Exemplo de chamada a um subprograma.

Três modelos semânticos de passagem de parâmetros são descritos na literatura: modo entrada (in mode), modo saída (out mode) e modo entrada/saída (inout mode). Estes modelos definem como os dados são transmitidos entre os parâmetros formais e reais. No modo entrada os parâmetros formais podem receber dados do parâmetro real, no modo saída eles podem transmitir dados para o parâmetro real e no modo entrada/saída podem fazer os dois. Estes três modelos básicos de transmissão de parâmetros são utilizados pelos projetistas de linguagem nas implementações dos métodos de passagem de parâmetros.

2.1 Passagem por valor

A passagem por valor é um modelo de implementação para parâmetros de modo entrada, onde o parâmetro formal recebe dados do parâmetro real. Nesse modo, o valor do parâmetro real é utilizado para inicializar o parâmetro formal que atua como uma variável local no subprograma.

A transferência dos dados pode ser feita por cópia dos valores ou pela transmissão de um caminho de acesso (ponteiro ou referência) para o valor do parâmetro real. Na passagem por valor normalmente é utilizado a transferência por cópia, visto que para a transmissão do caminho de acesso seria necessário que o dado estivesse numa célula protegida contra escrita, o que não é uma tarefa simples.

Exemplo: Imagine se o subprograma passe o parâmetro para outro subprograma.

O método de passagem por valor é rápida na vinculação e no tempo de acesso. Se for utilizado transferência por cópia, tem como desvantagem a necessidade de espaço adicional para armazenamento e as operações de transferência podem ser custosas se o parâmetro for grande.

2.2 Passagem por resultado

A passagem por resultado segue modelo semântico de passagem de parâmetro modo saída. Neste método nenhum valor é transmitido na chamada do subprograma, o parâmetro formal funciona como variável local e antes que o controle retorne para o chamador, o valor do parâmetro formal é copiado para o parâmetro real.

Assim como no método de passagem por valor, a passagem por resultado compartilha as desvantagens da necessidade de armazenamento e operações extras caso for retornado valores em vez do caminho de acesso. Porém devido a dificuldade de implementação da passagem por resultado transmitindo um

caminho de acesso utilizamos a transferência por cópia, dificuldade esta de garantir que o valor inicial do parâmetro real não seja utilizado no subprograma chamado.

Ainda como desvantagem existe o problema colisão de parâmetros reais. Suponha um subprograma com dois parâmetros formais diferentes, na chamada do subprograma atribuicao(a,a) qual será o valor retornado para a? Será retornado o valor que for atribuído por último ao parâmetro real a. Mas essa ordem depende da implementação. Exemplo de colisão de parâmetros reais em C#:

```
void atribuicao(out int x, out int y) {
    x = 29;
    y = 15;
}
...
f.atribuicao(out a, out a);
```

Outro problema é na escolha do tempo para avaliar os endereços dos parâmetros reais (momento da chamada ou momento do retorno).

Por exemplo, no subprograma abaixo o resultado pode depender do tempo de avaliação do endereço. Como a variável index pode ser modificada pelo subprograma, o parâmetro formal x pode assumir valores distintos dependendo da ordem de avaliação caso houver uma chamada deste subprograma passando uma lista como parâmetro real.

```
void subprograma(out int x, int index) {
    x = 23;
    index = 5;
}
...
sub = 21;
f.subprograma(list[sub], sub);
```

Se o endereço é avaliado na entrada do subprograma o valor 23 é atribuído a list[21], se for avaliado na saída o valor 23 é atribuído a list[5].

2.3 Passagem por valor-resultado

A passagem por valor-resultado é a implementação de parâmetro em modo entrada/saída onde é combinado a passagem por valor e a passagem por resultado. O valor do parâmetro real é usado para inicializar o parâmetro formal que atua como variável local. No termino do subprograma o valor do parâmetro formal é transmitido de volta para o parâmetro real.

A passagem por valor-resultado também é chamada de passagem por cópia (porque o parâmetro real é copiado para o parâmetro formal na chamada do subprograma e é copiado de volta para o parâmetro real no fim do subprograma) e partilha dos mesmos problemas da passagem por valor e passagem por resultado.

2.4 Passagem por referência

Mais um modelo de implementação de parâmetros em modo entrada/saída. O parâmetro real é compartilhado com o subprograma chamado pela transmissão

de uma caminho de acesso. A passagem por referência traz como vantagem o custo de tempo e espaço, pois não é necessário espaço duplicado e nem operações de cópia. Como desvantagem o acesso ao parâmetro formal é mais lento do que a passagem por valor devido ao endereçamento indireto, ou seja, é preciso mais de um acesso para chegar ao valor da variável. Além disso, se for necessário uma comunicação unidirecional torna-se um problema, pois pode haver mudanças no parâmetro real.

Presença de problemas como colisões de parâmetros reais e de apelido. A função em C++ abaixo ilustra o problema de apelido que é prejudicial para legibilidade e confiabilidade do código.

```
void função(int &first, int &second)
...
função(total, total)
```

2.5 Passagem por nome

A passagem por nome é mais um método de transmissão de parâmetros em modo entrada/saída. O parâmetro real é textualmente substituído pelo parâmetro formal em todas as suas ocorrências no subprograma. O parâmetro formal é vinculado a valores ou a endereços reais, mas a vinculação real é retardada até o momento que o parâmetro formal seja atribuído ou referenciado.

Esse método é complexo de implementar e ineficiente, porém é usado em tempo de compilação para parâmetros genéricos de subprogramas genéricos em C++, Java 5.0 e C# 2005.

2.6 Implementando métodos de passagem de parâmetros

A comunicação de parâmetros é realizada por uma pilha em tempo de execução que é gerenciada pelo sistema.

Na passagem de parâmetros por valor são copiados valores dos parâmetros reais para posições na pilha que serve como armazenamento para os parâmetros formais correspondentes. Já na passagem por resultado os valores que serão atribuídos ao parâmetro real fica na pilha até que possam ser recuperados pelo chamador após o término da chamada subprograma. Na passagem por valor-resultado uma posição na pilha é inicializada pela chamada e depois é usado como uma variável local no subprograma chamado. Na passagem por referência apenas é copiado o endereço do parâmetro para a pilha.

Figura.

2.7 Métodos de passagem de parâmetros das principais linguagens

O C usa passagem por valor, porém obtêm a semântica da passagem por referência utilizando ponteiros (copiado do ALGOL 68). O valor do ponteiro como parâmetro é copiado para a função chamada e nada é retornado, entretanto a função chamada pode fazer alterações naquele dado uma vez que possui o caminho de acesso. A proteção contra escrita é feita implicitamente na função chamada.

O C++ utiliza também a passagem por valor e garante a passagem por referência com ponteiros, além de um tipo especial de ponteiro chamado tipo de referência que após sua inicialização não pode referenciar outra variável.

Em Java os parâmetros também são passados por valor, porém como os objetos são apenas acessados por variáveis de referência os parâmetros são passados com a semântica de referência. Ada e Fortran 95+ permitem ao programador especificar o modo de cada parâmetro formal (entrada, saída e entrada-saída).

O C# utiliza a passagem por valor como padrão, mas também permite ao programador utilizar passagem por refreferência se o prefixo ***ref*** for utilizado antes dos dois parâmetros (real e formal). Também suporta passagem de parâmetro em modo saída, passado por referência, com o modificador out antes do parâmetro formal.

A passagem de parâmetro do PHP é semelhante à do C#, excepto que tanto o parâmetro real quanto parâmetro formal pode ser passado por referência precedendo um ou ambos os parâmetros com uma comercial (&).

Em Python e Ruby é utilizado a passagem por atribuição, nesse sentido todos os valores de dados são objetos. Assim cada variável armazena uma referencia para o valor. Se uma variável referenciada é acrescida de uma unidade então é criado um novo objeto com o valor da variável mais 1 e a variável referencia o novo objeto. No caso de vetor passado como parâmetro se houver uma atribuição ao parâmetro formal que referencia o vetor então não tem efeito no chamador, porém se houve uma atribuição à um elemento do vetor passado então o correspondente parâmetro real será modificado.

2.8 Verificação de tipos

A verificação dos tipos dos parâmetros reais em relação aos seus correspondentes parâmetros formais faz-se necessária uma vez que erros causados por tipos de parâmetros diferentes podem levar a erros de programa difíceis de detectar. Desta forma em muitas linguagens é feita a verificação de tipos.

Nas versões recentes do C e do C++ é feita verificação de tipos, onde os tipos dos parâmetros formais são incluídos na lista no cabeçalho da função. Não existe verificação de tipos de parâmetros em Python e Ruby porque estas linguagens são dinamicamente tipadas.

3 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*, 2.8) 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 3, cuja sintax 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 2: Código retirado de [6]

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 [6].

4 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, 6]. 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, 6], 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);
```

5 Sobrecarga de Subprogramas

A maioria das linguagens de progamação 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 [8].

Conforme apontado por Sebesta, as linguagens Ada, Java, C++, C# e F# são exemplo de linguagens que permitem sobrecarga de operadores [6].

6 Subprogramas Genéricos

No desenvolvimente 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 [6], há três tipos de polimorfismo, sendo eles:

Ad Hoc: Consiste na sobrecarga de subprogramas, conforme mostrado na seção 5.

Subtipo: Específico para linguagens orientadas a objeto, que não está dentro do escopo deste trabalho, mas pode ser encontrado em [10].

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#.

6.1 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 [6], 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 charmar 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].

6.2 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 [6].

Tipos genéricos em Java nos fornece muitos outros recursos (como *Wildcards*), que não abordados neste trabalho, mas que podem ser encontrados com detalhes em [9].

6.3 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 [6].

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

7 Questões de projetos referente a funções

Algumas questões relevantes ao projeto em relação a funções precisam ser levados em conta. As próximas seções discutem sobre questões de efeitos colaterais, tipos de retorno e quantidade de valores de retorno.

7.1 Efeitos colaterais

Uma questão relevante ao projeto em relação a funções é se efeitos colaterais serão permitidos. Linguagens puramente funcionais como Haskell, não possuem variáveis e portanto suas funções não possuem efeitos colaterais. Já linguagens que permitem funções com parâmetros passados por valor ou por referência, possuem efeitos colaterias, como por exemplo *aliasing* (*alias*, apelido). Considere o exemplo a seguir.

```
int x = 3;
... // se int* y = &x;
*y = 9;
```

O compilador nada pode fazer para otimizar o código acima, uma vez que não é possível saber se y = &x. Então, se o otimizador tentar aplicar, por exemplo, o constant propagation sabendo que x é 5, e y for um *alias*, ponteiro para x, então a propagação será feita de forma errada.

7.2 Tipos de Valores Retornados

C permite qualquer tipo ser retornado por suas funções exceto vetores e funções. Ambos nesse caso podem ser manipulados por ponteiros. C++ é semelhante ao C mas também permite tipos definidos pelo usuário ou classes serem retornados. Fortran 77 e Pascal permitem apenas tipos não estruturados. Ada, Python, Ruby e Lua, retornam valores de qualquer tipo, exceto no caso do Ada, onde funções não são tipos, portanto não podem ser retornadas (mas ponteiros para funções sim). Em Java e C#, qualquer tipo ou classe podem ser retornados por seus métodos, porém, como método não é um tipo, não pode ser retornado.

7.3 Quantidade de valores retornados

A maioria das linguagens atuais permitem apenas 1 valor de retorno. A linguagem Lua, permite o retorno de multiplos valores. Considere um exemplo de função na linguagem Lua que retorna 3 valores [6]. A ordem das variáveis que chamam a função, serão vinculadas na ordem dos valores retornados pela função.

A chamada da função

```
a, b, c = fun()
```

Onde o retorno da função fun() é

```
return 3, sum, index
```

8 Sobrecarga de operadores definidos pelo usuário

Linguagens como Ada, Python, Ruby e C++ suportam sobrecarga de operadores (*Overloaded Operators*). Sobrecarga de operadores permite efetuar operações como soma, subtração, multiplicação e divisão em tipos de dados definidos pelo usuário. Considere a listagem 3, que realiza a sobrecarga do operador + para efetuar a soma de vetores. A saída do programa é 4, 3.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {};
9      CVector (int,int);
10     CVector operator + (CVector);
11 };
12
13 CVector::CVector (int a, int b) {
14   x = a;
15   y = b;
16 }
17
18 CVector CVector::operator+ (CVector param) {
19   CVector temp;
20   temp.x = x + param.x;
21   temp.y = y + param.y;
22   return (temp);
23 }
24
25 int main () {
26   CVector a (3,1);
27   CVector b (1,2);
28   CVector c;
29   c = a + b;
30   cout << c.x << "," << c.y;
31   return 0;
32 }
```

Listing 3: Código retirado de [5]

9 Closure

Closure é uma variável local em uma função que é mantida viva (não é desalocada) após o retorno dessa função. Apenas linguagens que permitem subprogramas aninhados é possivel closure. Linguagens como C# e JavaScript possuem closure. Considere a listagem 4, na linguagem JavaScript, que utiliza closure.

```
1 function foo(x) {
2   var tmp = 3;
3   return function (y) {
4     alert(x + y + (++tmp));
5   }
6 }
7
8 var bar = foo(2);
9 bar(10);
```

Listing 4: Closure em JavaScript

No programa acima na linha 8, bar é um closure. A saída do programa exibe 16 como saída quando executado pela primeira vez, e vai incrementando em 1 a cada execução.

10 Co-rotinas

Co-rotinas são um tipo especial de subprogramas. A linguagem Lua é uma das linguagens que possui co-rotinas. Co-rotinas possuem múltiplos pontos de entrada controlados elas mesmas. A cada execução (chamado de resumo), a co-rotina volta a sua execução não do início, mas sim de algum outro ponto (por isso resumo). Co-rotinas mantém seu estado entre as ativações e portanto são sensíveis ao histórico, ou seja, possuem variáveis estáticas.

Geralmente, corrotinas são criadas pela aplicação por uma unidade chamada de unidade mestre. A unidade mestre não é uma co-rotina, é tem a função de ativar uma das co-rotinas, que após suas inicializações, resumem uma as outras até que a execução de todas termine e o controle volte para a unidade mestre. As figuras 2 e 3 ilustram o possível funcionamento de duas co-rotinas A e B.

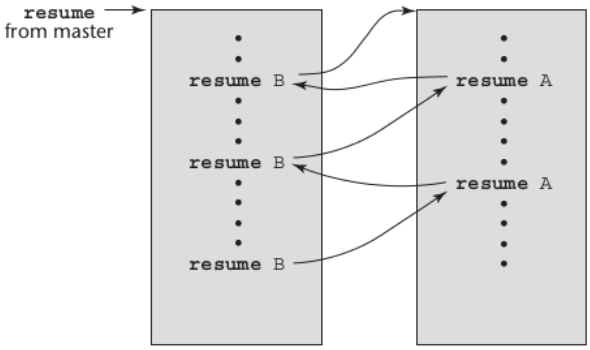

Figura 2: Execução de co-rotinas 1 [6]

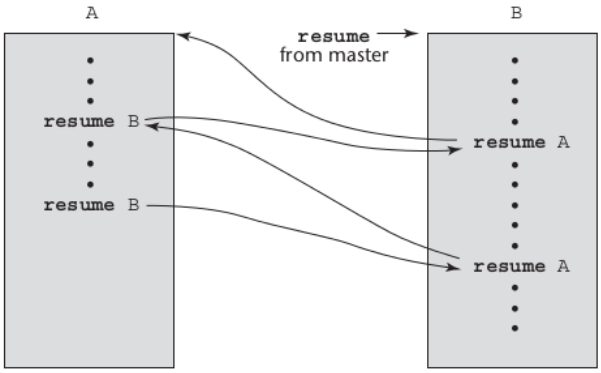

Figura 3: Execução de co-rotinas 2 [6]

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`. 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] C Plus Plus. Overload operator. `http://www.cplusplus.com/doc/tutorial/classes2/`. Acessado em: 23/09/2013.

[6] Robert W. Sebesta. *Concepts of Programming Languages (10th Edition)*. Addison-Wesley, 2012.

[7] William Stallings. *Computer Organization and Architecture: Designing for Performance (8th Edition)*. Prentice Hall, 2009.

[8] The Java Tutorials. Defining methods. `http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html`. Acessado em: 23/09/2013.

[9] The Java Tutorials. Lesson: Generics. `http://docs.oracle.com/javase/tutorial/java/generics/index.html`. Acessado em: 23/09/2013.

[10] The Java Tutorials. Polymorphism. `http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html`. Acessado em: 23/09/2013.