

Paradigmas de Linguagens de Programação

Subprogramas

Cristiano Lehrer, M.Sc.

Características Gerais dos Subprogramas

- Cada subprograma tem um único ponto de entrada.
- Toda unidade de programa chamadora é suspensa durante a execução do programa chamado, o que implica na existência de somente um subprograma em qualquer momento dado.
- O controle sempre retorna ao chamador quando a execução do subprograma encerra-se.

```
int main() {  
    int x = soma(10, 15);  
    printf("%d", x);  
    return 0;  
}
```

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

Definições Básicas (1/3)

- Uma **definição de subprograma** descreve a interface e as ações da abstração de subprograma.
- Uma **chamada a subprograma** é a solicitação explícita para executar o programa:
 - Um subprograma está **ativo** se, depois de ter sido chamado, ele iniciou a execução, mas ainda não a concluiu.
 - Existem dois tipos fundamentais de subprogramas:
 - Funções.
 - Procedimentos.

Definições Básicas (2/3)

- Um **cabeçalho de subprograma** é a primeira linha da definição, incluindo o nome, o tipo de subprograma e os parâmetros formais.
- O **perfil de parâmetro** de um subprograma é o número, a ordem e os tipos de seus parâmetros formais.
- O **protocolo** de um subprograma é seu perfil de parâmetros mais, se for uma função, seu tipo de retorno.

Definições Básicas (3/3)

- Uma **declaração de subprograma** providencia o protocolo, mas não o corpo do subprograma.
- Um **parâmetro formal** é uma variável vazia listada no cabeçalho do subprograma e utilizada no subprograma.
- Um **parâmetro real** representa um valor ou endereço utilizado no subprograma pela declaração que o chamou.

Parâmetros (1/3)

- Parâmetros posicionais

- Em quase todas as linguagens de programação, a correspondência entre parâmetros reais e formais – ou a vinculação de ambos – é feita simplesmente pela posição:
 - O primeiro parâmetro real é vinculado ao primeiro parâmetro formal e assim por diante.

- Exemplo em **Java**:

```
public void somar(int a, int b) {...}
```

```
...
```

```
s = somar(10, 20);
```

Parâmetros (2/3)

- Parâmetros Nomeados

- O nome do parâmetro formal a que um parâmetro real deve ser vinculado é especificado com esse último.
- Vantagem:
 - A ordem é irrelevante.
- Desvantagem:
 - Usuário precisa saber os nomes dos parâmetros formais.
- Exemplo em **ADA**:

```
SOMADOR (COMPRIMENTO => MEU_COMPRIMENTO,  
         LISTA => MEU_ARRAY,  
         SOMA => MINHA_SOMA) ;
```

Parâmetros (3/3)

- Valores padrão:
 - Valores utilizados se nenhum parâmetro real for passado ao parâmetro formal no cabeçalho do subprograma.
 - Exemplo em C++:

```
float calculo_pagamento(float renda,  
                        float tarifa_imposto,  
                        int isencoes = 1);  
  
...  
pagamento = calculo_pagamento(20000.0, 0.15);
```

Procedimentos e Funções

- Procedimentos:
 - Declarações definidas pelo usuário.
- Funções:
 - Operadores definidos pelo usuário.

Questões de Projeto Referentes aos Subprogramas

- Qual método (ou métodos) de passagem de parâmetro é usado.
- Os tipos dos parâmetros reais são verificados em relação aos tipos de parâmetros formais.
- As variáveis locais são estaticamente ou dinamicamente alocadas.
- Se subprogramas puderem ser passados como parâmetros, qual é o ambiente de referência desse subprograma.
- Se subprogramas puderem ser passados como parâmetros, os tipos de parâmetros são verificados quanto aos parâmetros em chamadas aos subprogramas passados.
- Definições de subprograma podem aparecer em outras definições de subprograma.
- Subprogramas podem ser sobrecarregados.
- Subprogramas podem ser genéricos.
- A compilação separada ou independente é possível.

Ambientes de Referência Locais (1/2)

- Variáveis que são definidas dentro de subprogramas são chamadas **variáveis locais**:
 - Se as variáveis locais são stack-dinâmicas:
 - Vantagem:
 - Suporte a recursão.
 - Armazenamento para variáveis locais de todos os subprogramas podem ser compartilhado.
 - Desvantagem:
 - Tempo de alocação e desalocação.
 - Endereçamento indireto.
 - Subprogramas não podem ser sensíveis à história.
 - Para as variáveis locais estáticas é o oposto.

Ambientes de Referência Locais (2/2)

- Exemplos:

- FORTRAN 77 e 90 – a maioria é estática, mas pode ser stack-dinâmica.
- C – ambos (variáveis declaradas com `static` são estáticas, mas o padrão é stack-dinâmica).
- Pascal, Modula-2 e ADA – somente stack-dinâmica.
- Código em C:

```
int somador (int list[], int listlen){
    static int soma = 0;
    int cont;
    for (cont = 0; cont < listlen; cont++)
        soma += list[cont];
    return soma;
}
```

Método de Passagem de Parâmetros (1/2)

- Modelos Semânticos de Passagem de Parâmetros:
 - Modo entrada (*in mode*).
 - Modo saída (*out mode*).
 - Modo de entrada/saída (*inout mode*).
- Modelos Conceituais de Transferência:
 - Um valor real é transferido fisicamente.
 - Um caminho de acesso é transmitido.

Método de Passagem de Parâmetros (2/2)

- Modelos de Implementação de Passagem de Parâmetros:
 - Passagem por Valor
 - *Pass-by-Value*
 - Passagem por Resultado
 - *Pass-by-Result*
 - Passagem por Valor-Resultado
 - *Pass-by-Value-Result*
 - Passagem por Referência
 - *Pass-by-Reference*
 - Passagem por Nome
 - *Pass-by-Name*

Passagem por Valor

- Passagem por Valor (*in mode*):
 - O valor do parâmetro real é usado para inicializar o parâmetro formal correspondente, que, então age como uma variável local no subprograma.
 - Normalmente implementada pela transferência de dados real, mas pode também utilizar o caminho de acesso.
 - Desvantagens do acesso pelo método do caminho:
 - O valor deve estar numa célula protegida contra a escrita.
 - Acesso mais complicado.
 - Desvantagens do método de cópia física:
 - Requer mais armazenamento.
 - Custo para transferir fisicamente o parâmetro.

Passagem por Resultado

- Passagem por resultado (*out mode*):
 - Valores locais são passados de volta ao chamador.
 - Transferência física é usualmente utilizada.
 - Desvantagens:
 - Se o valor é passado, tempo e espaço.
 - Em ambos os casos, a dependência da ordem pode ser problema:
 - Exemplo em [Ada](#):

```
procedure sub1 (y : out Integer, z : out Integer)
...
sub1 (x, x);
```
- O valor de x no chamador dependerá da ordem dos assinalamentos no retorno.

Passagem por Valor-Resultado

- Passagem por valor-resultado (*inout mode*):
 - Transferência física, em ambos os modos.
 - Também chamado de passagem por cópia.
 - Desvantagens:
 - Os referentes ao passagem por resultado.
 - Os referentes ao passagem por valor.

Passagem por Referência (1/2)

- Passagem por referência (*inout mode*):
 - Passagem pelo método de caminho de acesso.
 - Também conhecido como passagem por compartilhamento.
 - Vantagens:
 - Processo de passagem de parâmetros é eficiente.

Passagem por Referência (2/2)

- Desvantagens:

- Acesso lento.

- Pode permitir *aliasing*:

- Colisão de parâmetros – exemplo em C++:

```
void sub1(int& a, int& b)
```

```
...
```

```
sub1(x, x);
```

- Colisão de elementos de array – exemplo em C++:

```
sub1(a[i], a[j]); /* se i = j */
```

```
sub2(a, a[i]);
```

- Colisão entre parâmetros formais e globais.

Passagem por Nome (1/3)

- Passagem por Nome (múltiplos modos):
 - Parâmetros passados por nome, o parâmetro real é, com efeito, textualmente substituído para o parâmetro formal correspondente em todas as suas ocorrências no subprograma.
 - Parâmetros formais são vinculados a valores ou a endereços reais no momento da chamada ao subprograma.
 - Propósito:
 - Flexibilidade na vinculação tardia.
 - Resultados semânticos:
 - Se o parâmetro real for uma variável escalar, será passado por referência.
 - Se o parâmetro real for uma expressão constante, será passado por valor.
 - Se o parâmetro real for um elemento de *array*, será passado diferente de qualquer em dos métodos estudados.

Passagem por Nome (2/3)

- Exemplo, em ALGOL:

```
procedure BIGSUB;  
  integer GLOBAL;  
  integer array LIST[1 : 2];  
  procedure SUB (PARAM) ;  
    integer PARAM;  
    begin  
      PARAM := 3;  
      GLOBAL := GLOBAL + 1;  
      PARAM := 5;  
    end;  
end;
```

```
begin  
  LIST[1] := 2;  
  LIST[2] := 2;  
  GLOBAL := 1;  
  SUB (LIST [GLOBAL] ) ;  
end.
```

- Após a execução, o array LIST tem os valores 3 e 5, ambos definidos em SUB.
 - O acesso a LIST[2] é oferecido depois que GLOBAL se incrementa para o valor 2 em SUB.

Passagem por Nome (3/3)

- Desvantagens da passagem por nome:
 - Referência muito ineficientes.
 - São difíceis de implementar e podem confundir tanto leitores como os escritores de programas.

Exemplos em Linguagens de Programação (1/2)

- **FORTRAN :**
 - Antes do **FORTRAN 77**, passagem por referência.
 - **FORTRAN 77** – variáveis escalares são passados como valor-resultado.
- **ALGOL 60:**
 - Passagem por nome é o padrão.
 - Passagem por valor é opcional.
- **C:**
 - Passagem por valor.
 - Passagem por referência através da utilização de ponteiros.
- **Pascal e Modula-2:**
 - Passagem por valor é o padrão.
 - Passagem por referência é opcional.

Exemplos em Linguagens de Programação (2/2)

- **C++:**

- Como o **C**, mas também permite tipo referência, o qual providência a eficiência da passagem por referência com a semântica da passagem por valor.

- Exemplo:

```
void fun (const int &p1, int p2, int &p3) { .... }
```

- `p1` é passado por referência, mas não pode ser mudado na função `fun`.
- o parâmetro `p2` é passado por valor.
- o parâmetro `p3` é passado por referência.

- **ADA:**

- Todos os três modos semânticos estão disponíveis:

- Em modo *out* podem ser atribuídos, mas não referenciados.
- Em modo *in* podem ser referenciados, mas não atribuídos.
- Em modo *inout* podem ser tanto atribuídos como referenciados.

- **Java:**

- Sempre por valor, inclusive para objetos.

Parâmetros de Verificação de Tipo

- Agora considerado muito importante para a confiabilidade;
- **FORTRAN 77** e **C** original:
 - Nenhum.
- **Pascal**, **Modula-2**, **FORTRAN 90**, **Java** e **ADA**:
 - Ela é sempre requerida.
- **ANSI C** e **C++**:
 - A escolha é feita pelo usuário.
 - Exemplo:

```
double sin(x)
    double x;
{...}
```

Evita a verificação de tipos

```
double sin(double x)
{...}
```

Método protótipo, variáveis são coagidas caso não sejam do mesmo tipo.

Considerações de Projeto para Passagem de Parâmetros

- Eficiência.
- Uma direção ou duas direções.
- Estas duas estão em conflito com uma outra:
 - Boa programação:
 - Limitar o acesso as variáveis, o qual significa que uma direção é sempre possível.
 - Eficiência:
 - Passar por referência é o modo mais rápido para passar estruturas de tamanho significativo.
- Também, funções poderão não permitir parâmetros de referência.

Parâmetros que são Nomes de Subprogramas (1/5)

- Os parâmetros são checados quanto ao tipo:
 - **Pascal** inicial e **FORTRAN 77** não faziam.
 - Versões posteriores do **Pascal**, **Modula-2** e **FORTRAN 90** fazem.
 - **ADA** não permite subprogramas como parâmetros.
 - **C** e **C++** passam ponteiros para funções, sendo que os parâmetros podem ser verificados.

Parâmetros que são Nomes de Subprogramas (2/5)

Exemplo em C:

```
int soma(int a, int b) { return a + b; }  
  
int multiplica(int a, int b) { return a * b; }  
  
void aplicarOperacao(int (*operacao)(int, int), int x, int y) {  
    printf("Resultado: %d\n", resultado = operacao(x, y));  
}  
  
int main() {  
    int a = 5, b = 3;  
  
    aplicarOperacao(soma, a, b);           // Resultado: 8  
    aplicarOperacao(multiplica, a, b);    // Resultado: 15  
  
    return 0;  
}
```

Parâmetros que são Nomes de Subprogramas (3/5)

- Qual é o ambiente de referenciamento correto para executar o subprograma passado:
 - Possibilidades:
 - O ambiente da instrução de chamada que ordena o subprograma passado (vinculação rasa).
 - O ambiente da definição do subprograma passado (vinculação profunda).
 - O ambiente da instrução de chamada que passou o subprograma como um parâmetro real (vinculação *ad hoc*).
 - Para linguagens de escopo estático, a vinculação profunda é mais natural.
 - Para linguagens de escopo dinâmico, a vinculação rasa é mais natural.

Parâmetros que são Nomes de Subprogramas (4/5)

- Exemplo na sintaxe do **Pascal**:

```
procedure SUB1;
var x : integer;
  procedure SUB2;
  begin
    write('x = ', x);
  end;
  procedure SUB3;
  var x : integer;
  begin
    x := 3;
    SUB4 (SUB2);
  end;
```

```
procedure SUB4 (SUBX);
var x : integer;
begin
  x := 4;
  SUBX;
end;
begin
  x := 1;
  SUB3;
end.
```

Parâmetros que são Nomes de Subprogramas (5/5)

- Vinculação rasa:
 - O ambiente de referenciamento dessa execução é o de SUB4, de modo que a referência a x em SUB2 é vinculada à local x em SUB, e o resultado do programa é $x = 4$.
- Vinculação profunda:
 - O ambiente de referenciamento da execução de SUB2 é o de SUB1, de modo que a referência a x em SUB2 é vinculada à local x em SUB1, e o resultado é $x = 1$.
- Vinculação *ad hoc*:
 - O ambiente de referenciamento da execução de SUB2 é o de SUB3, de modo que a referência a x em SUB2 é vinculada à local x em SUB3, e o resultado é $x = 3$.

Subprogramas Sobrecarregados (1/3)

- Um subprograma sobrecarregado é um que tem o mesmo nome que outro subprograma no mesmo ambiente de referenciamento.
- **C++** e **Java** possuem subprogramas sobrecarregados predefinidos, e usuários podem escrever seus próprios subprogramas sobrecarregados.

Subprogramas Sobrecarregados (2/3)

- Exemplo em **ADA**:

```
procedure MAIN is
  type VETOR_FLOAT is array (INTEGER range <>) of FLOAT;
  type VETOR_INT is array (INTEGER range <>) of INTEGER;
  ...
  procedure SORT (LISTA_FLOAT : in out VETOR_FLOAT;
                  LIM_INF : in INTEGER;
                  LIM_SUP : in INTEGER) is
    ...
  end SORT;
  ...
  procedure SORT (LISTA_INT : in out VETOR_INT;
                  LIM_INF : in INTEGER;
                  LIM_SUP : in INTEGER) is
    ...
  end SORT;
  ...
end MAIN;
```

Subprogramas Sobrecarregados (3/3)

- Exemplo em C:

```
void fun(float b) {...}
```

```
void fun() {...}
```

...

```
fun();
```

- Problemas:

```
void fun(float b = 0.0) {...}
```

```
void fun() {...}
```

...

```
fun(); /* chamada ambígua - erro de compilação*/
```

Subprogramas Genéricos

- Um **subprograma genérico** ou **polimórfico** é um subprograma que leva parâmetros de diferentes tipos em várias ativações.
- Subprogramas sobrecarregados apresentam uma espécie particular de polimorfismo chamado de polimorfismo *ad hoc*.
- Polimorfismo paramétrico é apresentado por um subprograma que leva um parâmetro genérico usado em uma expressão de tipos que descreve os tipos dos parâmetros do subprograma.
- Exemplo de polimorfismo paramétrico em C++:

```
template <class Tipo>
    Tipo max (Tipo primeiro, Tipo segundo) {
        return primeiro > segundo ? primeiro : segundo;
    }
```

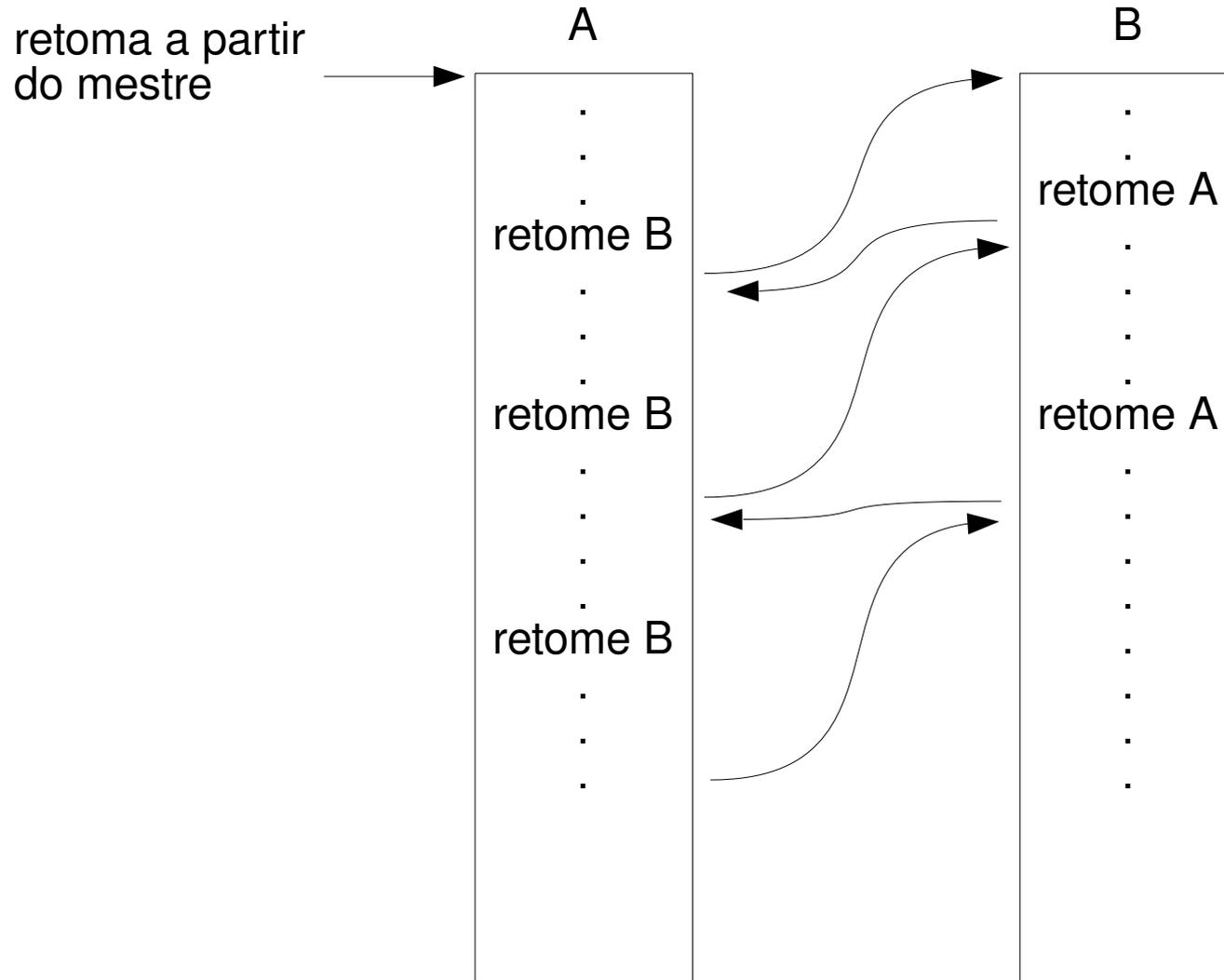
Compilação Separada e Independente

- **Compilação Independente** é a compilação de algumas unidades do programa realizadas separadamente do resto do programa, sem o benefício da informação de cabeçalho.
- **Compilação Separada** é a compilação de algumas unidades do programa realizadas separadamente do resto do programa, utilizando a informação do cabeçalho para verificar a validade da interface entre as duas partes.
- Exemplos de Linguagens de Programação:
 - **FORTRAN II** para **FORTRAN 77** – independente.
 - **FORTRAN 90**, **ADA**, **Modula-2**, **C++** – separada.
 - **Pascal** – nenhum delas (completa).

Corrotinas (1/4)

- Uma corrotina é um subprograma que pode possuir múltiplas entradas e controles por si só.
- Também conhecido como modelo de controle unitário simétrico.
- A invocação de uma corrotina é chamada de retomada em vez de chamada.
- A primeira retomada é para o início, mas chamadas subsequentes começam no ponto onde a corrotina parou de executar na última vez.
- Tipicamente, corrotinas repetidamente são retomadas por outras, possivelmente para sempre.
- Corrotinas providenciam execução quase concorrente de unidades de programas.
- Sua execução é intercalada, mas não sobreposta.

Corrotinas (2/4)



Corrotinas (3/4)

Exemplo em Python:

```
def contador():
    print("Iniciando...")
    yield 1
    print("Depois do primeiro yield")
    yield 2
    print("Quase lá...")
    yield 3
    print("Fim!")

# Criando o gerador
gen = contador()

# Consumindo o gerador passo a passo
print(next(gen)) # Saída: Iniciando... \n 1
print(next(gen)) # Saída: Depois do primeiro yield \n 2
print(next(gen)) # Saída: Quase lá... \n 3
# next(gen) agora levantaria StopIteration
```

Corrotinas (4/4)

- Python
 - Suporte a co-rrotinas com `yield` desde o Python 2.5
- Lua
 - Suporte a co-rrotinas está no core da linguagem, com `coroutine.create`, `coroutine.resume` e `coroutine.yield`.
- PHP
 - PHP 5.5 introduziu generators (`yield`), mas foi com extensões como `amphp` ou `ReactPHP` que se simulam co-rrotinas.
- Unity / C#
 - C# tem `IEnumerable` e `yield return`, que permitem corrotinas estilo Unity.
- Perl
 - Suporte a corrotinas via módulos como `Coro`.