Linguagens de Imperativa – parte 3: subprogramas

Prof. Arnaldo Candido Junior

UNESP – IBILCE

São José do Rio Preto, SP

Subprogramas

- Consistem basicamente de funções e procedimentos
  - Recebem parâmetros; efetuam alguma operação; podem devolver um resultado
  - Conforme a linguagem, podem ser aninhados (prática não recomendada, exceto para clousures, mais adiante)

Subprogramas (2)

- Existem dois tipos de subprogramas: **funções** (retornam valores) e **procedimentos** (não retornam valores)
- Adotaremos a terminologia da linguagem C: procedimentos são funções que não retornam valores
  - Procedimento tem outro significado em linguagens formais (algoritmo que nunca para)

Subprogramas (3)

- Definição de função: fragmento do código em que o subprograma é inserido
- Chamada de função: fragmento do código em que o subprograma é efetivamente utilizado
  - Uma função pode ser chamada em diferentes pontos
  - E pode chamar a si mesma (funções recursivas)

Parâmetros

- Parâmetros formais: definidos junto com a função
  - double media(double a, double b) { … }
- Parâmetros reais: passados na chamada de função
  - double c = media(10, 20);

Parâmetros Opcionais

- Parâmetros opcionais (keyword parameters)
  - Tem valores default e sua passagem é opcional
  - Um valor diferente do default pode ser definido ao evocar a função
  - Tipicamente o nome do parâmetro é passado durante a chamada da função. Podem ser passados fora de ordem
  - A seguir, exemplo em Python

Parâmetros Opcionais (2)

```
def f(a, b, c="default", d=0):
  print(a, b, c, d)

f("texto", 10)
f("texto", 10, "alterado")
f("texto", 10, d=20)
f("texto", 10, d=20, c="alterado")

# python também permite transformar
# arrays e dicionários em parâmetros
obrigatórios = ["texto", 10]
opcionais = {"c": "alterado", "d" :20}
f(*obrigatórios)
f("texto", 10, **opcionais)
f(*obrigatórios, **opcionais)
```

Parâmetros Variáveis

- Diferentes linguagens permitem que o número de valores de uma função varie
  - As vezes são chamadas de variadic functions
  - Um exemplo é a função printf em C
  - O excesso de parâmetro é tipicamente capturado em uma array
  - Exemplo a seguir em Python

Parâmetros Variáveis (2)

```
def f(fixo, *variavel):
  print(fixo)
  for x in variavel:
    print(x)

# converter parâmetros para arrays é o processo
# inverso da conversão de arrays para parâmetros
# mostrada no exemplo anterior
f("texto")
f("texto", 10)
f("texto", 10, 20)
f("texto", 10, 20, 30)
```

Passagem de Parâmetros

- **in mode**: a função chamada recebe dados da chamadora pelo parâmetro
- **out mode**: a função chamada devolve dados para a chamadora pelo parâmetro
- **inout mode**: a função chamada recebe e devolve dados à chamadora pelo parâmetro
- Diferentes estratégias de passagem pode ser implementadas

Passagem de Parâmetros (2)

- Passagem **por valor**: os valores das variáveis da função chamadora são copiados para a função chamada
  - Mudança nos parâmetros da função chamada não afetam a função original
- Passagem **por referência**: uma referência à função chamadora é passada
  - Mudanças nos parâmetros da função chamada afetam a original

Passagem de Parâmetros (3)

- Por default, geralmente
  - Passados por valor: chars, números e boleanos são passados por valor
  - Passados por referência: todo o resto (strings, arrays, registros, objetos, etc)

Passagem de Parâmetros (3)

- Outras formas de passagem:
  - Passagem **por resultado**: usado para parâmetros out mode
  - Passagem **por nome**: de forma resumida, a função chamada acessa uma variável local da chamadora diretamente (não é uma referência)
    - Conceito teórico e pouco usado na prática

Funções genéricas

- Nas linguagens estaticamente tipadas, uma função genérica pode receber diferentes tipos de parâmetros
  - Conceito relacionado: classes e atributos genéricos
  - Outro conceito relacionado: sobrecarga de função
    - Na função genérica, a sobrecarga é feita pelo compilador em vez do programador

Funções genéricas (2)

```
public class Main {
  public <T> void mostraArray(T[] array) {
    for (T element : array) {
      System.out.println(element);
    }
  }
  public <T extends Number> double somaNumeros(T[] numbers){
    double sum = 0.0;
    for (T number : numbers) {
      sum += number.doubleValue();
    }
    return sum;
  }
  // continua
```

Funções genéricas (3)

```
  public static void main(String[] args) {
    Main main = new Main();

    Integer[] inteiros = {1, 2, 3, 4, 5};
    main.mostraArray(inteiros);
    System.out.println("soma de inteiros: " +
      main.somaNumeros(inteiros));

    Double[] reais = {1.1, 2.2, 3.3, 4.4, 5.5};
    main.mostraArray(reais);
    System.out.println("soma de doubles: " +
      main.somaNumeros(reais));
  }
}
```

Funções puras

- São mais próximas do conceito matemático de função
  - Não mudam estado do programa
  - Ou seja, não tem efeitos colaterais (entrada e saída é aceitável)
- Representam um conceito essencial em linguagens funcionais

Funções anônimas

- Funções anônimas não tem um nome oficial dentro do programa
- Porém, variáveis de outras funções podem referenciar essas funções
- Via de regra, funções anônimas são funções puras

Funções anônimas (2)

- Também são conhecidas como funções lambda em referência ao cálculo-λ
  - Teoria que estuda a correspondência entre funções matemáticas e computacionais
- Exemplo a seguir em Python

Funções anônimas (3)

```
def f(x):
  return x+1

# variável com referência à função nomeada (sem
parenteses)
soma_1 = f
print(soma_1(5))

# função anônima
soma_2 = lambda x: x+2
print(soma_2(10))
```

Clousure

- Uma clousure é uma função aninhada ou anônima que captura o ambiente (variáveis) da função externa
- Podem ser retornadas pela função externa
  - Útil no contexto de programação funcional
- Obs: variável capturada pode ser uma cópia (números, booleanos, ...) ou uma referência (outros casos)

Clousure (2)

```
def soma_10():
  x = 10
  return lambda y: y + 10

def contador():
  x = 1
  def interna():
    nonlocal x
    x += 2
    return x
  return interna # sem parenteses
```

Corrotinas

- De forma intuitiva, uma corrotina é uma função chamada que está no “mesmo nível” da função chamadora
- Avançado: no paradigma concorrente, são usadas para implementar produtores e consumidores
- Veremos uma versão mais simples de corrotina em Python

Corrotinas (2)

- Corrotinas não perdem sua pilha de variáveis entre diferentes evocações
- São capazes de retomar a execução do ponto em que pararam na última execução
- Em Python, retornam valores com o comando “yield”

Corrotinas (3)

```
def corrotina():
  x = 0
  while x < 10:
    yield x
    x+=1
  while x < 100:
    yield x
    x+=10
  while x < 1000:
    yield x
    x+=100
  # um return reseta a corrotina
  # obs: uma corrotina não precisa ser resetada
  # ex.: contador, while True, etc
  return None
```

Corrotinas (4)

```
def f():
  for x in corrotina():
    print(x)
  for x in corrotina(): # recomeçando execução após reset
    print(x)

f()
```