Subprogramas

Paradigmas de Programação – BCC/UFRPE
Lucas Albertins – lucas.albertins@deinfo.ufrpe.br

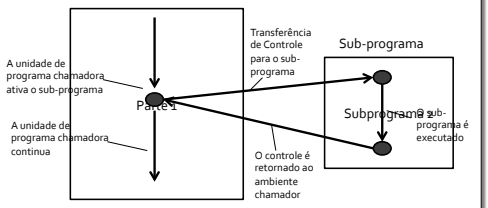
Agenda

- Fundamentos de Subprogramas
- Definições Básicas
- Passagem de Parâmetros
- Subprogramas como parâmetros
- Subprograma Genéricos
- Decisões de projeto para funções
- Implementação de Subprogramas

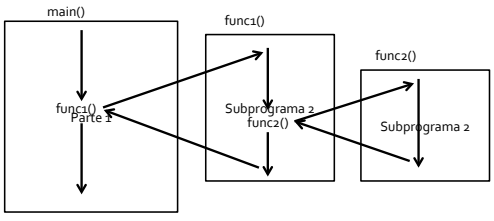
Fundamentos de Subprogramas

- Abstração de Processo
- Características
  - Possuem um único ponto de entrada;
  - O invocador tem a sua execução suspensa durante a execução de um subprograma
  - Quando o subprograma invocado termina a execução, o controle retorna para o invocado

Fundamentos de Subprogramas



Fundamentos de Subprogramas



Definições básicas

- **Subprograma**: descreve a interface e as ações;
- **Chamada**: requisição explicita para que o subprograma seja executado;
- **Cabeçalho**: primeira linha (definição: nome, o tipo do retorno, parâmetros formais);
- **Perfil dos parâmetros**: lista de parâmetros formais,incluindo o número, onde e seus tipos;
- **Protocolo**: perfil de parâmetros e o tipo de retorno (se função);
- **Declaração**: o protocolo mas não o corpo;

Parâmetros

- **Parâmetro formal**: variável fictícia, definida no cabeçalho de um subprograma. O seu escopo é geralmente igual ao do subprograma;
- **Parâmetro real**: representa o valor (ou endereço) das variáveis ou constantes, utilizados no ponto de invocação do subprograma;
- Normalmente a vinculação é efetuada pela ordem dos parâmetros formais aos reais (parâmetros posicionais)
  - Ex: 1º parâmetro real é vinculado ao 1º parâmetro formal, e assim por diante

Parâmetros

- Parâmetros com palavras-chave
  - Ex: Python

```
soma(tamanho= meu_tamanho,
     lista=meu_vetor, soma=minha soma)
```

- Parâmetros Default
  - Ex. Python:

```
def maximo(x,y=0,z)
```

Tipos de Subprogramas

- Procedimentos: coleções de sentenças que definem computações parametrizadas;
- Funções: Semelhantes aos procedimentos, mas semanticamente modeladas como funções matemáticas.
  - Se é um modelo fiel, não produz efeitos colaterais
  - Mas na prática, elas produzem

Variáveis Locais

- Dinâmica na pilha (Java, C++, Python, C# e default em C)
  - Desvantagens: alocação, desalocação, tempo de inicialização; endereçamento indireto; não é sensível a história;
  - Vantagens: Suporte à recursão
- Estática (quando se usa **static** em C)
  - Desvantagem: não permite recursividade;
  - Vantagem: não há endereçamento indireto (mais eficiente); não existe reserva e liberação de memória; sensível à história.

Subprogramas Aninhados

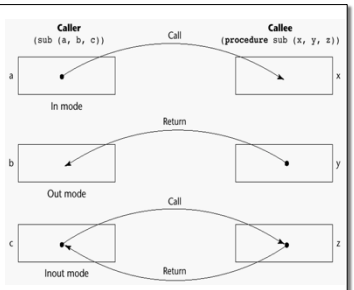
- Permitem: Algol60, Pascal, Ada, Python, Ruby

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

- Não permitem: Linguagem baseadas em C

Passagem de Parâmetros

- Entrada (in)
- Saída (out)
- Entrada e Saída (inout)



Passagem de Parâmetros

- Modelos de conceituais de passagem de parâmetro
  - Passagem por valor (entrada): cópia
  - Passagem por resultado (saída)
  - Passagem por valor/resultado (entrada e saída)
  - Passagem por Referência (entrada e saída)
  - Passagem por nome (modo múltiplo)

Passagem por valor

- O valor do parâmetro real é usado para inicializar o parâmetro formal correspondente
  - Normalmente implementado por cópia
  - Pode ser implementado transmitindo um caminho de acesso, mas não é recomendado (garantir proteção de escrita não é simples)
  - Desvantagens:
    - Se for uma cópia, espaço de armazenamento extra necessário e a cópia pode ser custosa para parâmetros grandes
    - Se for via caminho de acesso, garantir proteção de escrita

Passagem por Resultado

- Nenhum valor é transmitido para o subprograma; o parâmetro formal funciona como uma variável local e seu valor é transmitido para o parâmetro real quando o controle é retornado para o invocador
  - Requer armazenamento local e uma operação de cópia
- Problemas potenciais:
  - `sub(p1, p1);`
  - `sub(list[sub], sub);`

Passagem por Valor/Resultado

- Uma combinação de passagem por valor e por resultado
- Algumas vezes chamada de passagem por cópia
- Parâmetros formais tem armazenamento local
- Desvantagens
  - Os mesmos de passagem por valor
  - Os mesmos de passagem por resultado

Passagem por Referência

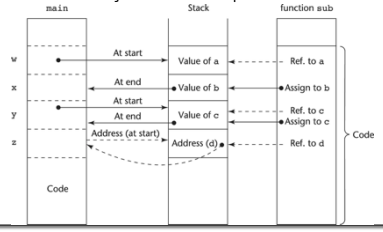
- Passa um caminho de acesso (endereço)
- Também chamado de passagem por compartilhamento (o parâmetro real é compartilhado com o subprograma)
- Vantagem
  - Processo de passagem é eficiente (nenhuma cópia e armazenamento duplicado)
- Desvantagem
  - Acesso mais lento (comparado com valor) para parâmetros formais
  - Potenciais para efeitos colaterais

Passagem por Nome

- Parâmetro formal é substituído textualmente pelo parâmetro real em todas as suas ocorrências no subprograma no momento da chamada
- Mas a vinculação para um valor ou endereço só acontece no momento de uma referência ou atribuição com o parâmetro
- Permite flexibilidade em vinculação tardia
- Implementação requer que o ambiente de referenciamento do invocador seja passado com o parâmetro para que o endereço o parâmetro real possa ser calculado

Implementando passagem de parâmetros

- Na maioria das linguagens usa-se uma pilha
- Passagem por referência é a mais simples de implementar pois só um endereço é colocado na pilha



Passagem de Parâmetros nas linguagens

- C
  - Passagem por valor e por referência (uso de ponteiros)
- C++
  - Existe um tipo de ponteiro especial para passagem por referência
- Java
  - Tipos primitivos usam passagem por valor
  - Objetos usam passagem por referência
- Ada
  - Três modos de semântica: **in, out, in out**; **in** é o padrão
  - Parâmetros formais **out** pode ser atribuídos mas não referenciados; parâmetros **in** podem ser referenciados mas não atribuídos; **in out** podem ser referenciados e atribuídos

Passagem de Parâmetros nas linguagens

- C#
  - Parâmetros podem ser declarados **in, out** ou **in out**
- PHP
  - Similar a C#, exceto que o parâmetro real e formal podem especificar referências
- Perl
  - Todos os parâmetros reais são implicitamente alocados num vetor predefinido (**@_**)
- Python e Ruby
  - Usam atribuição (o real é atribuído para o formal)

Checagem de Tipo de Parâmetros

- Muito importante por questões de confiabilidade
- Primeiras versões de C: nenhuma verificação
- Pascal, FORTRAN 90+, Java e Ada fazem
- Perl, Javascript e PHP não precisam de checagem de tipo
- Em Python e Ruby as variáveis não tem tipos, por isso a checagem não é possível

Considerações de Projeto em Passagem de Parâmetros

- Eficiência
- Transferência de dados de uma ou duas direções
- Boa prática em programação sugere acesso limitado a variáveis, o que significa transferência em uma direção quando possível
- No entanto passagem por referência para estruturas de tamanho significativo é mais eficiente

Subprogramas como parâmetros

- Ex:

```
void imprima(char str[80]) {
      printf("%s\n",str);
}
void alo_mundo(void (func)(char str[80] )) {
      func("Alo mundo");
}
int main() {
      alo_mundo(imprima);
      return0;
}
```

Sobrecarga de Subprogramas

- Tem o mesmo nome que outro subprograma no mesmo ambiente de referenciamento com protocolo distinto
- C++, Java, C# e Ada permitem ao usuário escrever múltiplas versões de subprogramas com o mesmo nome
- Em C++

```
int maximo(int x, int y);

double maximo(double x, double y);

int maximo(int x, int y, int w, int z);
```

Subprogramas Genéricos

- Aquele que recebe parâmetros de tipos diferentes em diferentes invocações
- Uso de polimorfismo
  - Sobrecarga de subprogramas são um tipo de ad hoc polimorfismo
- Polimorfismo de subtipo significa que uma variável T pode acessar qualquer objeto do tipo T ou qualquer tipo derivado de T (linguagens OO)
- No Polimorfismo paramétrico um subprograma toma um parâmetro genérico que é usado em uma expressão que determina os possíveis tipos do parâmetro
  - Uma forma barata em tempo de compilcação para vinculação dinâmica

Subprogramas Genéricos

- C++
  - Uso de templates

```
#include <iostream>
template <class Type>
Type maximo(Type prim, Type seg){
      return prim>seg? prim:seg;
}

int main(){
      std::cout << maximo(3, 7) << std::endl;
      std::cout << maximo(3.0, 7.0) << std::endl;
      std::cout << maximo<double>(3, 7.0) <<
      std::endl;
      return 0;
}
```

Subprogramas Genéricos

- Java
  - Tipos genéricos devem ser classes
  - Restrições podem ser especificada nos tipos das classes que podem ser passadas como parâmetros genéricos
  - Tipos coringas (*Wildcards*)

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

- O parâmetro é um array do tipo T (genérico)
- A chamada seria doIt<String>(myList)

```
public static <T extends Comparable> T
     doIt(T[] list) { … }
```

- O tipo genérico deve ser de uma classe que implementa Comparable

Subprogramas Genéricos

- Java
  - Coringas - Wildcards
  - `Collection<?>` é um coringa para a coleção de classes

```
void printCollection(Collection<?> c) {
      for (Object e: c) {
        System.out.println(e);
      }
    }
```

Funciona para qualquer coleção de classes

Subprogramas Genéricos

- C# 2005
  - Suporta métodos genéricos similar a Java
  - Diferença: o tipo dos parâmetros reais numa chamada podem ser omitidos se o compilador puder inferir o tipo não especificado
  - C# não suporta coringas

Decisões de Projeto para Funções

- Efeitos colaterais são permitidos?
  - Em geral parâmetros devem ser do modo de entrada para evitar efeitos colaterais
- Quais tipos de retorno de valores são permitidos?
  - A maioria das linguagens imperativas restringem os tipos de retorno
  - C permite qualquer tipo exceto vetores e funções
  - C++ é como C
  - Ada podem retornar qualquer tipo, menos subprogramas
  - métodos em Java e C# podem retornar qualquer tipo, mas como um método não é um tipo não pode ser retornado
  - Python e Ruby tratam métodos como objetos de primeira ordem, então eles podem ser retornados, assim como qualquer outra classe

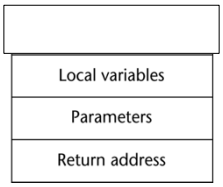
Implementação de Subprogramas

Implementação de Subprogramas

- Supondo subprogramas simples
  - Semântica de chamada
    - Salvar o status de execução do invocador
    - Passar os parâmetros
    - Passar o endereço de retorno para o invocado
    - Transferir controle para o invocado
  - Semântica de retorno
    - Se passagem por valor/resultado ou parâmetros de saída, mover os valores atuais dos parâmetros formais para os parâmetros reais
    - Se for uma função, mover o retorno da função para um local que o invocador possa acessá-lo
    - Restaurar o status de execução para o invocador
    - Transferir o controle para o invocador

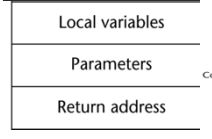
Implementação de Subprogramas

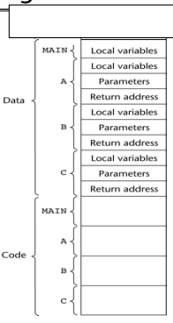
- Partes: o código e a parte que não é código (variáveis locais, e dados que podem ser alterados)
- A parte que não é código é chamada de registro de ativação



Implementação de Subprogramas

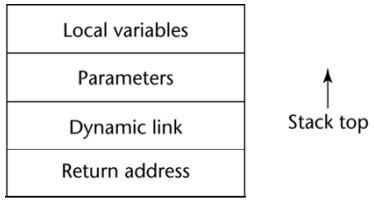
- Partes: o código e a parte que não é códi (variáveis locais, e dados que podem ser alterados)
- A parte que não é código é chamada de registro de a





Implementação de Subprogramas

- Para subprogramas que permitam recursão usa-se uma pilha para registrar os registros de ativação



Implementação de Subprogramas

```
void sub(float total, int part)
{
  int list[5];
  float sum;
  …
}
```

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

Implementação de Subprogramas

```
void sub(float total, int part)
{
  int list[5];
  float sum;
  …
}
```

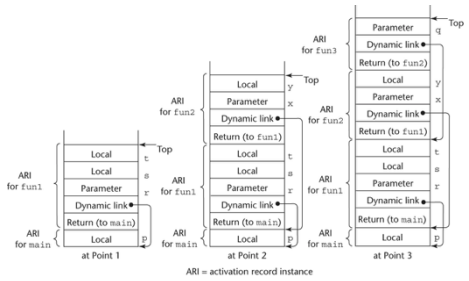
Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

Implementação de Subprogramas

```
void fun1(float r) {
   int s, t;
   ... ------------->2
   fun2(s);
   ...
}
void fun2(int x) {
   int y;
   ... ------------->3
   fun3(y);
   ...
}
void fun3(int q) {
   ...
}
void main() {
   float p;
   ... ------------->1
   fun1(p);
   ...
}
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

Implementação de Subprogramas



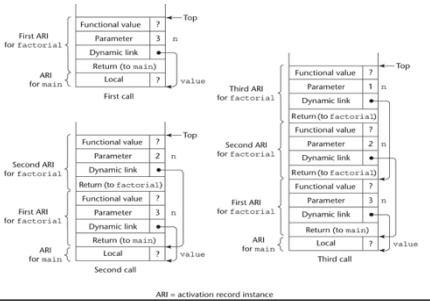
Implementação de Subprogramas

+ O Registro de Ativação usado no exemplo anterior suporta recursão

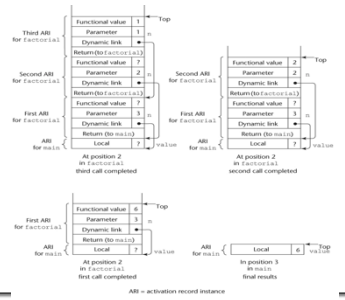
```
int factorial (int n) {
  <---------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
  <---------------2
}
void main() {
  int value;
  value = factorial(3);
  <-----------------3
}
```

Functional value	
Parameter	n
Dynamic link	
Return address	

Implementação de Subprogramas



Implementação de Subprogramas

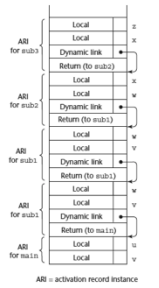


Implementação de Subprogramas

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

- Implementação de escopo dinâmico geralmente se dá de duas formas
  - Acesso em Profundidade (*Deep Access*)
    - Referencias não locais são acessadas através de uma busca pelas instâncias dos registros de ativação que estão ativos
  - Acesso em superfície (*Shallow Access*)
    - As variáveis locais são armazenadas em um local central onde cada variável terá uma pilha

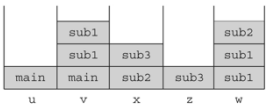
Implementação de Subprogramas



- Acesso em Profundidade
  - Suponha a seguinte ordem
    - main chama sub1
    - sub1 chama sub1
    - sub1 chama sub2
    - sub2 chama sub3
- Tempo de execução mais lento do que linguagens com escopo estático
- Registro de Ativações devem armazenar o nome das variável além do seu valor para cada busca, enquanto que em escopo estático só os valores são necessários

Implementação de Subprogramas

- Acesso em Superfície
  - Uma pilha para cada variável
  - Uma tabela central com uma entrada para cada nome de variável
- Rápida referencia a variáveis
- Manter pilhas nas entradas e saídas a subprogramas é custoso



(The names in the stack cells indicate the program units of the variable declaration.)

Leitura Adicional

- Capítulo 9 e 10 – Subprogramas. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011
- Próxima aula
  - Capítulo 11 – Tipos Abstratos de Dados e Encapsulamento. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011