Linguagens de Programação

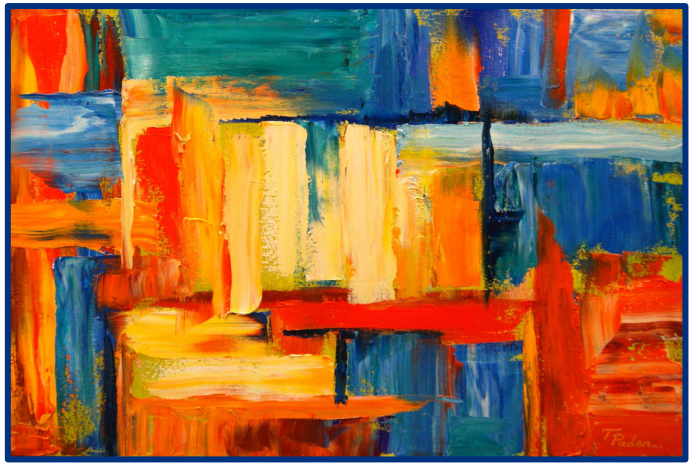
Subprogramas

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Abstrações
- Subprogramas
- Parâmetros
- Passagem de parâmetros
  - Direção da passagem
  - Mecanismos de passagem
  - Momento da passagem

ABSTRAÇÕES



Linguagens de Programação

Abstrações

- Fundamental para resolução de problemas
  - Deve-se focar nos aspectos essenciais, ignorando os aspectos irrelevantes

- Exemplos
  - Comandos SO => abstrai instruções de hardware
  - Assembly => abstrai instruções binárias
  - LP => abstrai instruções em *assembly*

Abstrações

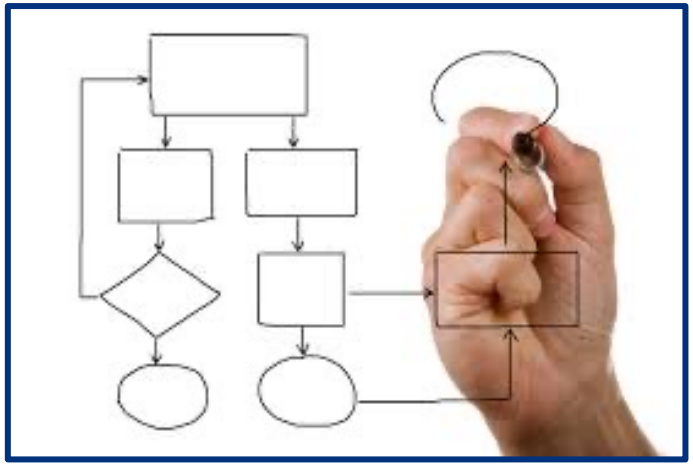
- Especificamente para LPs
  - Abstração sobre o hardware
  - Programadores podem criar suas próprias abstrações

- Visão do programador
  - O que uma parte do programa faz?
    - (abstração de uso)
  - Como é implementado?
    - (abstração de implementação)

Tipos de Abstrações

- Abstrações de **Processos**
  - Abstrações sobre o fluxo de controle do programa
  - Subprogramas
    - **Ex.:** sqrt, printf, …
- Abstrações de **Dados**
  - Abstrações sobre as estruturas de dados do programa
  - Tipos de Dados
    - **Ex:** Pilha, tabelas hash, …

SUBPROGRAMAS



Linguagens de Programação

Subprogramas

- **Função** (Abstração de uma expressão)
  - Exemplo: fatorial
    - Usuário: mapeamento **n** para **n!**
    - Implementador: algoritmo recursivo

- **Procedimento** (Abstração de um comando)
  - Exemplo: ordenação
    - Usuário: ordenação de vetor de inteiros
    - Implementador: método bolha

Funções não deveriam ter efeito colateral, mas na prática podem ter

Subprogramas

- Permitem segmentar o programa em vários blocos logicamente relacionados
- Servem para reusar trechos de código que operam sobre dados diferenciados
- Modularizações efetuadas com base no tamanho do código possuem baixa qualidade
- Vários propósitos: facilitar a legibilidade, depuração, manutenção e reutilização

Subprogramas

- Possui um único ponto de entrada
  - Exceção: subprogramas em FORTRAN podem ter múltiplas entradas
- O chamador tem sua execução suspensa durante a execução do subprograma chamado
- O controle sempre retorna ao chamador quando a execução do subprograma termina

Nomenclaturas

- **Definição de subprograma:** descreve interface e ações que o subprograma implementa

1) **Cabeçalho de subprograma:** primeira linha de sua definição (nome, retorno, parâmetros)
2) **Perfil dos parâmetros:** lista de parâmetros formais, incluindo a quantidade, ordem e seus tipos
3) **Protocolo de um subprograma:** perfil dos parâmetros em conjunto com o tipo de retorno para funções
4) **Declaração de subprograma (protótipo):** fornece o protocolo mas não o corpo do subprograma
5) **Chamada (invocação) de subprograma:** requisição explícita para executar subprograma

```
float double(float);

float double(float v) {
    return 2 * v;
}

void main(float v) {
    float d = double(5);
}
```

PARÂMETROS

Definições

- **Argumento** => valor passado para a função
- **Parâmetro de chamada** ou **parâmetro real** => expressão que produz o argumento
- **Parâmetro formal** => identificador que denota o argumento no interior da função

```
// função
float area(float raio) {
     return pi * raio * raio;
}

// ...

    // chamadas
    area(5);
    area(a+b);
```

Tipo	Exemplo
Argumento	5 e resultado de a + b
Parâmetro real	5 e expressão a + b
Parâmetro formal	raio

Parâmetros

- A ausência de parâmetros reduz
  - **Redigibilidade** => Necessário incluir operações para atribuir os valores desejados às variáveis globais
  - **Legibilidade** => Na chamada não há menção aos nomes usados nos parâmetros
  - **Confiabilidade** => Não exige que sejam atribuídos valores a todas as variáveis globais

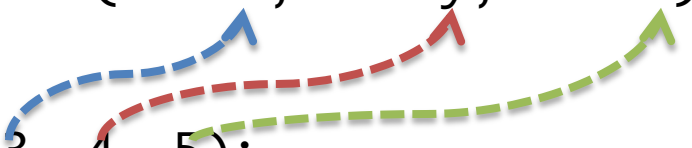
Correspondências

- Correspondência entre **parâmetros formais** e **parâmetros reais** (posicional)
  - A vinculação é feita pela ordem dos parâmetros formais
  - Primeiro parâmetro formal ligado ao primeiro parâmetro real e assim por diante
  - Vantagens: seguro e efetivo

- Exemplo

```
int multiplica(int x, int y, int z);

multiplica(3, 4, 5);
```



Correspondências

- Correspondência entre **parâmetros atuais** e **parâmetros formais** (palavra-chave)
  - A vinculação é feita na chamada, especificando o nome do parâmetro formal
  - Vantagem: ordem dos parâmetros irrelevante
  - Desvantagem: o nome do parâmetro deve ser conhecido na chamada

- Exemplo

```
function multiplica(x, y, z: integer) return integer

multiplica(z=>3, x=>4, y=>5);
```

Valores Padrões em Parâmetros Formais

- Valores definidos nos parâmetros formais podem ser utilizados na falta destes valores nos parâmetros reais
- Existem em: ADA, C++, FORTRAN, Visual Basic
- Devem aparecer no final em LPs com parâmetros posicionais
- Exemplo

```
int max(int x, int y = 0, int z = 0)

max(3, 4, 5); // Retorna 5
max(3, 4);    // Retorna 4
max(3);       // Retorna 3
```

Lista Variável em Parâmetros Formais

- Oferece maior flexibilidade à LP
- Reduz a confiabilidade, pois não é possível verificar os tipos dos parâmetros em tempo de compilação
- Exemplo

```
int printf(char* formatador, ...);

printf("Nome: %s", nome);
printf("%d * %d = %d", x, y, x*y);
```

PASSAGEM DE PARÂMETROS

Linguagens de Programação

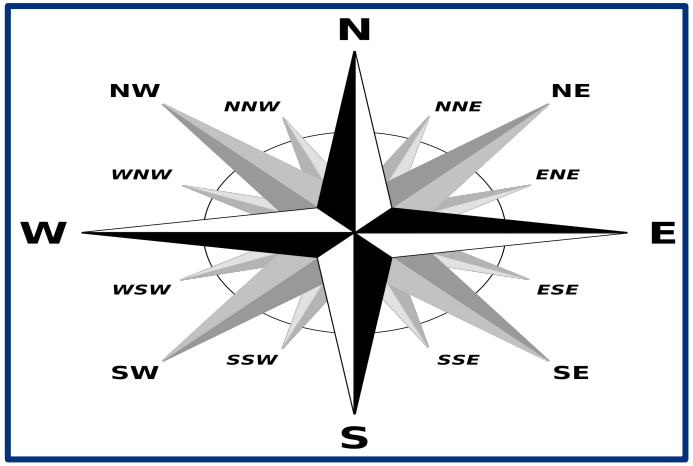
Passagem de Parâmetros

- Formas de transmitir parâmetros de e/ou para subprogramas chamados

- Faz parte do processo de passagem de parâmetros a eventual atualização de valores dos parâmetros reais durante a execução do subprograma

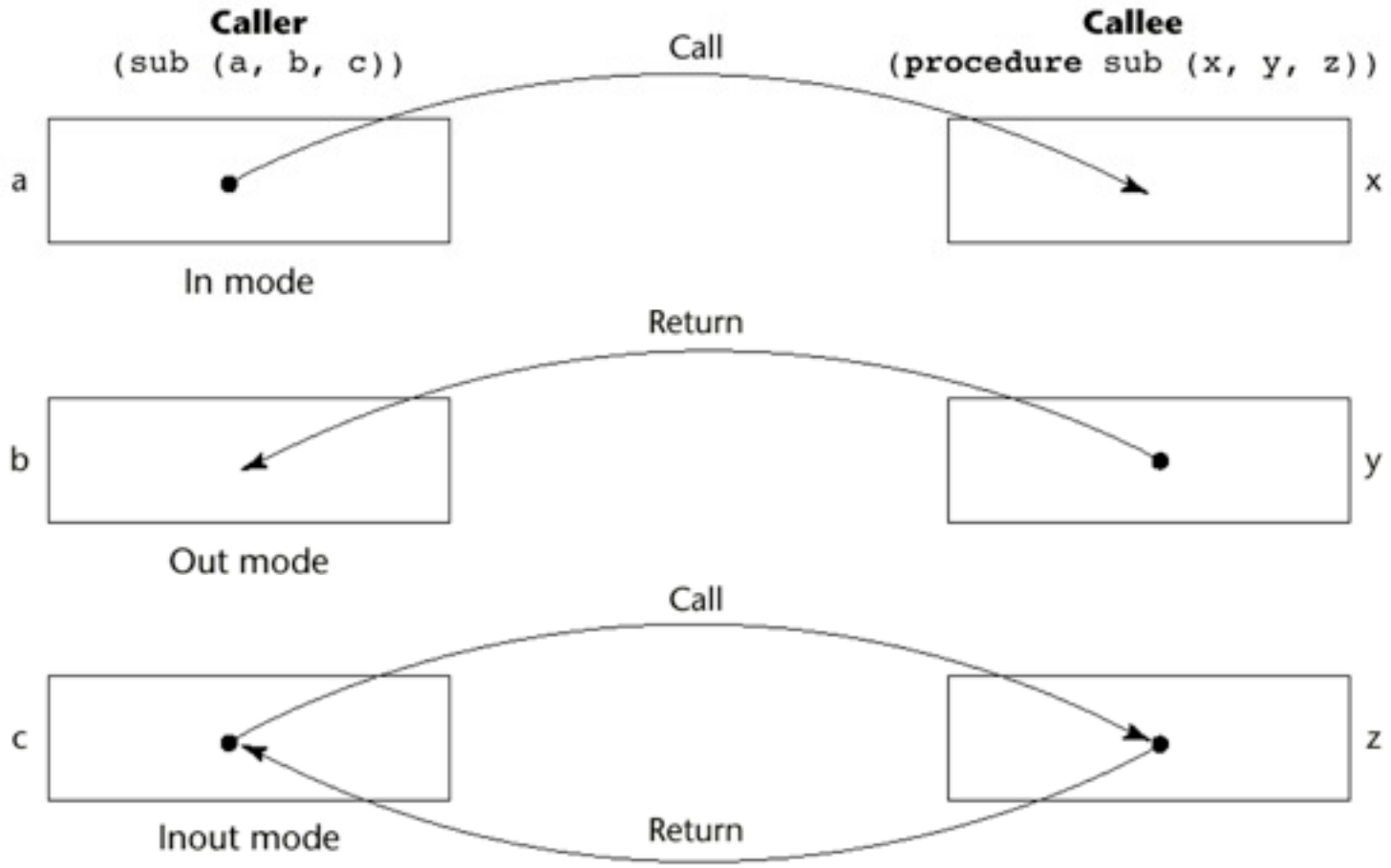
- Três aspectos importantes
  - Direção da passagem
  - Mecanismo de implementação
  - Momento no qual a passagem é realizada

PASSAGEM DE PARÂMETROS > DIREÇÃO



Direção da Passagem

- As possíveis direções de passagem de parâmetros



Direção da Passagem

- As possíveis direções de passagem de parâmetros

Direção da passagem	Forma do parâmetro real (R)	Atribuição do parâmetro formal (F)	Fluxo
Entrada Variável	Variável, Constante ou Expressão	Sim	R → F
Entrada Constante	Variável, Constante ou Expressão	Não	R → F
Saída	Variável	Sim	R ← F
Entrada e Saída	Variável	Sim	R ↔ F

Direção da Passagem

- Passagem unidirecional
  - Entrada variável (em C)

```
void funct(int param1, char param2);
void funct(int* param1, char* param2);
```

  - Entrada constante (C++)

```
void funct(const int param1);
```

- Passagem bidirecional
  - Referência (C++)

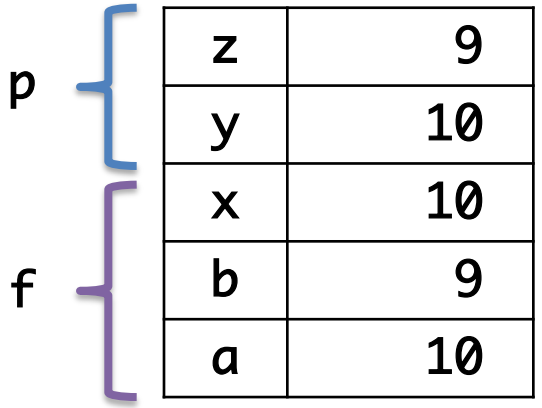
```
void funct(int& param1, char& param2);
```

PASSAGEM DE PARÂMETROS > MECANISMOS

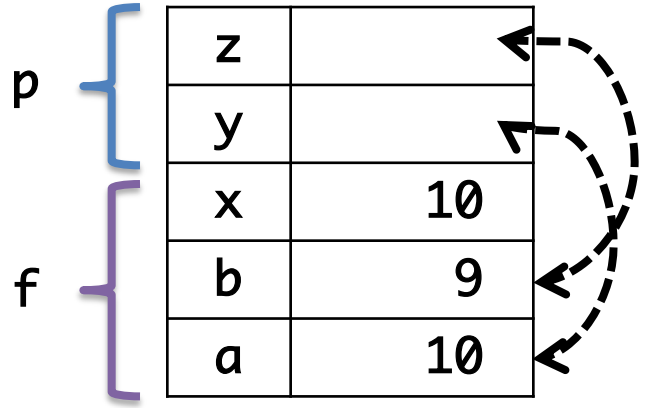
Linguagens de Programação

Mecanismos de Passagem

- Por **cópia**
  - Viabiliza a passagem unidirecional de entrada variável
  - Facilita a recuperação do estado do programa em interrupções inesperadas



- Por **caminho de acesso** (**referência**)
  - Proporciona semântica uniforme e simples na passagem de todos os tipos
  - Mais eficiente por não envolver cópia de dados
  - Pode ser ineficiente em implementação distribuída



Mecanismos de Passagem

- Mecanismos de passagem de parâmetros
  - Passagem **por valor** (*in mode*)
  - Passagem **por resultado** (*out mode*)
  - Passagem **por valor-resultado** (*in/out mode*)

  } Normalmente por cópia

  - Passagem **por referência** } Caminho de acesso
  - Passagem **por nome**

Passagem por Valor (*in mode*)

- O valor do parâmetro real é utilizado para inicializar o parâmetro formal correspondente
- Pode ser implementado como caminho de acesso, mas reforçar proteções de escrita não é fácil
- Desperdício de memória
  - Ex: duplicação de um *array*
- Custo da transferência
  - Ex: tempo de cópia do *array*

Passagem por Valor (*in mode*)

- Exemplo

```
void naoTroca(int x, int y) {
    int aux;
    aux = x;
    x = y;
    y = aux;
}

void troca(int* x, int *y) {
    int aux;
    aux = *x;
    *x = *y;
    *y = aux;
}
```

```
int a = 3, b = 5;
naoTroca(a, b);
printf("%d %d", a, b);
troca(&a, &b);
printf("%d %d", a, b);
```

Passagem por Resultado (*out mode*)

- O parâmetro formal age como uma variável local, mas nenhum valor é passado para o subprograma
- Quando o subprograma retorna, o valor é copiado para o parâmetro real do chamador
- Desvantagens
  - Demanda espaço extra e operação de cópia
  - Colisão de parâmetros reais
- Exemplo

```
function test(out erro: integer) return integer
```

Passagem por Valor-Resultado (*in/out mode*)

- Combinação de passagem por valor e por resultado
- Desvantagens
  - As mesmas da passagem por valor
  - As mesmas da passagem por resultado
- Exemplo

```
function multiplica(in out a: integer, in b: integer)
```

Mecanismo de Passagem (por cópia)

- Resumo dos três tipos de passagens

  Declaração:
```
procedure p(in x: integer, out y: real,
            in out z: real) return integer
```

  Chamada:
```
p(10, a, b)
```

Modo de passagem	Argumento	Efeito na entrada	Efeito na saída
Valor (*in*)	valor	`x := 10`	-
Resultado (*out*)	variável	-	`a := y`
Valor-resultado (in out)	variável	`z := b`	`b := z`

Passagem por Referência

- Transmite o caminho de acesso (**endereço**)
- Parâmetro real é compartilhado com o subprograma invocado/chamado
- Vantagem
  - A transmissão de parâmetros é eficiente
- Desvantagens
  - Acesso menos eficiente aos parâmetros (através de endereçamento indireto)
  - Pode permitir sinônimos (*aliasing*)

Passagem por Referência

- Exemplo

```
void troca(int& x, int& y) {
    int aux;
    aux = x;
    x = y;
    y = aux;
}
```

```
int a = 10, b = 20;
troca(a, b);
cout << a << b << endl;
```

Problemas da Passagem por Referência

- Colisão de parâmetros reais

```
void calc(int& x, int& y) {
    x += 5;
    y *= 3;
}
```

```
int a = 10;
calc(a, a);
cout << a << endl;
```

- Colisão de elementos de *array*

```
void calc(int& x, int& y) {
    x += 5;
    y *= 3;
}
```

```
int a[] = { 5, 10 };
calc(a[0], a[0]);
cout << a[0] << " "
     << a[1] << endl;
```

Passagem por Nome

- Parâmetro formal é substituído textualmente pelo parâmetro real
- Parâmetros formais são vinculados a um método de acesso no momento de chamada, mas a vinculação real a um valor ou endereço é retardada até que uma referência ou atribuição ao parâmetro formal seja realizada
- Propósito: flexibilidade de vinculação tardia (só quando é realmente necessário)

Problemas por Nome

- Exemplo

```
procedure first;
    var integer array LIST[1:2];
        integer GLOBAL;
    procedure sub(PARAM);
      var integer PARAM;
      begin
          PARAM := 3; // LIST[GLOBAL] := 3;
      end;
begin
    LIST[1] = 40;
    GLOBAL := 2;
    sub(LIST[GLOBAL]);
end;
```

PASSAGEM DE PARÂMETROS > MOMENTO

Linguagens de Programação

Momento da Passagem

- Em que momento o parâmetro de chamada/real deve ser avaliado?
  - **Normal** (*eager*): avaliação na chamada do subprograma
  - **Por nome** (*by name*): avaliação quando parâmetro formal é usado
  - **Preguiçosa** (*lazy*): avaliação quando parâmetro formal é usado pela primeira vez

Maioria das LPs (tais como C, PASCAL, JAVA e ADA) adota modo normal

Momento da Passagem

- Exemplo

```
int caso(int x, int w,
         int y, int z) {
    if (x < 0) return w;
    if (x > 0) return y;
    return z;
}

caso(p(), q(), r(), s());
```

- **Avaliação normal**
  - Avaliação desnecessária de funções na chamada caso
  - Pode reduzir eficiência e flexibilidade
- **Avaliação por nome**
  - Somente uma de **q**, **r** ou **s** seria avaliada
  - Problemas
    - **p** poderia ser avaliada duas vezes
    - **p** poderia produzir efeitos colaterais
- **Avaliação Preguiçosa**
  - Única execução de **p** e somente uma de **q**, **r** ou **s**

Momento da Passagem

- Outro exemplo

```
bool test(bool b1, bool b2)  {
   if (b1) return b2;
   else return false;
}
```

```
int t = 5;
int n = 0;
test(n > 0, t/n > 0.5);
```

- **Avaliação normal**
  - Gera exceção (divisão por zero)
- **Avaliação preguiçosa**
  - Retorna *false*

ISSO É TUDO, PESSOAL!