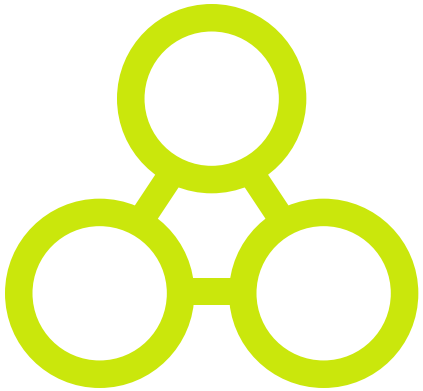


nemo

ontology & conceptual modeling research group

Linguagens de Programação

4 – Variáveis e Constantes

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

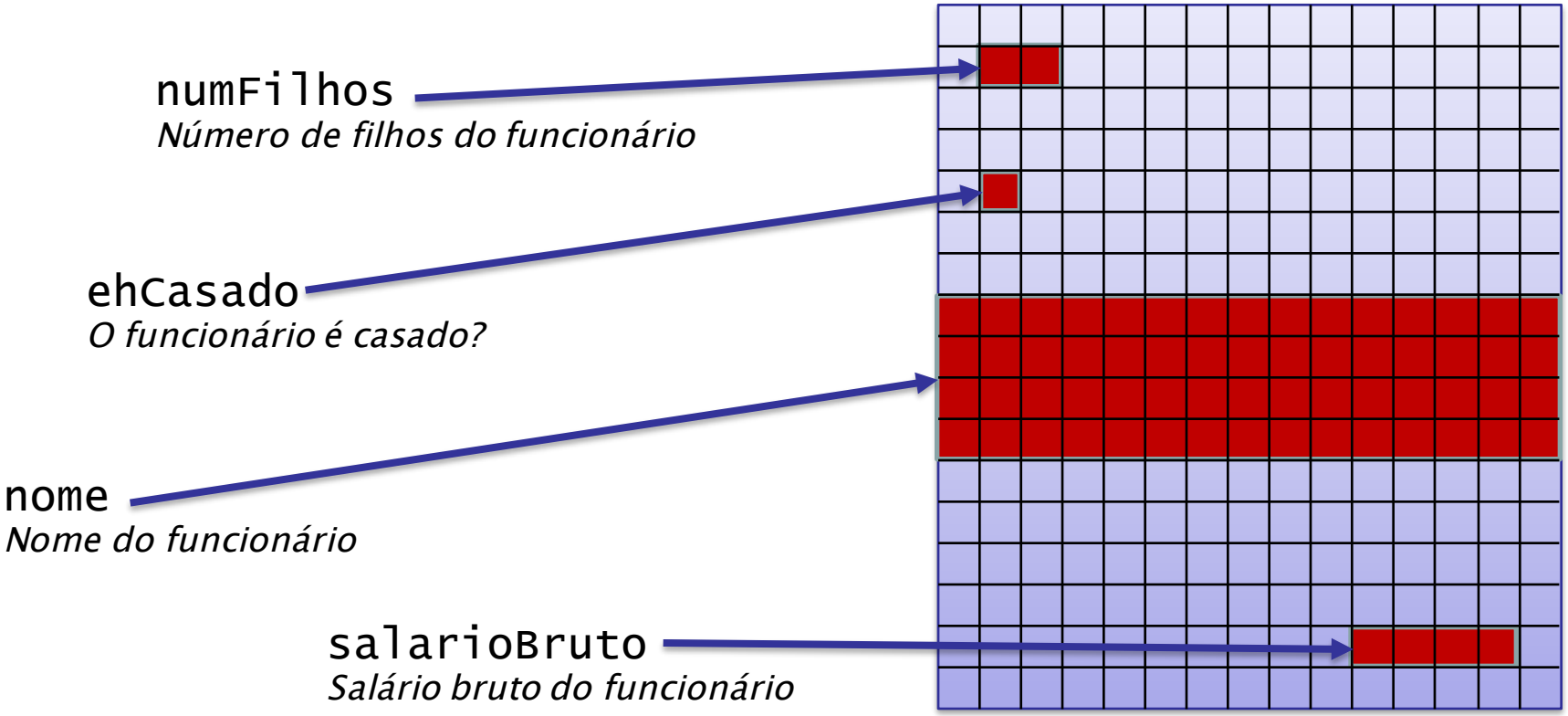
Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

Variáveis

- Abstração para uma ou mais células de memória responsáveis por armazenar o estado de uma entidade de computação;
  - Abstração? Estado? Entidade?



Variáveis

- Principal evolução das linguagens de montagem (*assembly*) em relação às linguagens de máquina: endereçamento local (o tradutor converte);
- Importância do conceito:

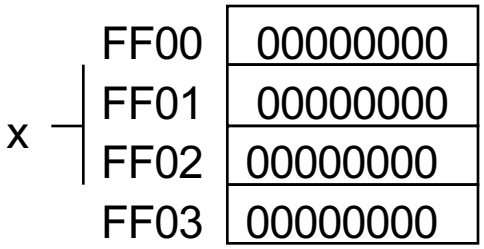
> "Uma vez que o programador tenha entendido o uso de variáveis, ele entendeu a essência da programação". – Dijkstra

  - Um pouco exagerado: em LPs funcionais e lógicas o conceito de variáveis é um pouco diferente;
  - Porém aplica-se perfeitamente ao paradigma imperativo, centrado em variáveis e atribuição de valores.

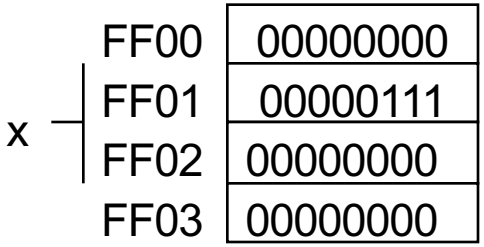
Variáveis

```
// Exemplo em C:
unsigned x;
x = 7;
x = x + 3;
```

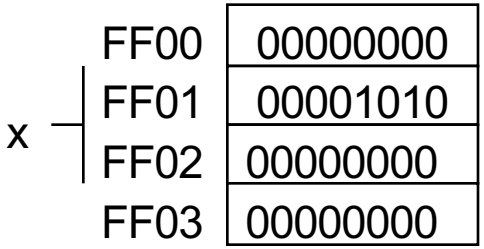
Passo 1 – Espaço de memória é alocado e associado à variável x



Passo 2 – Valor 7 é colocado no espaço de memória associado a x



Passo 3 – Valor corrente de x é somado a 3 e resultado é colocado no espaço de memória associado a x

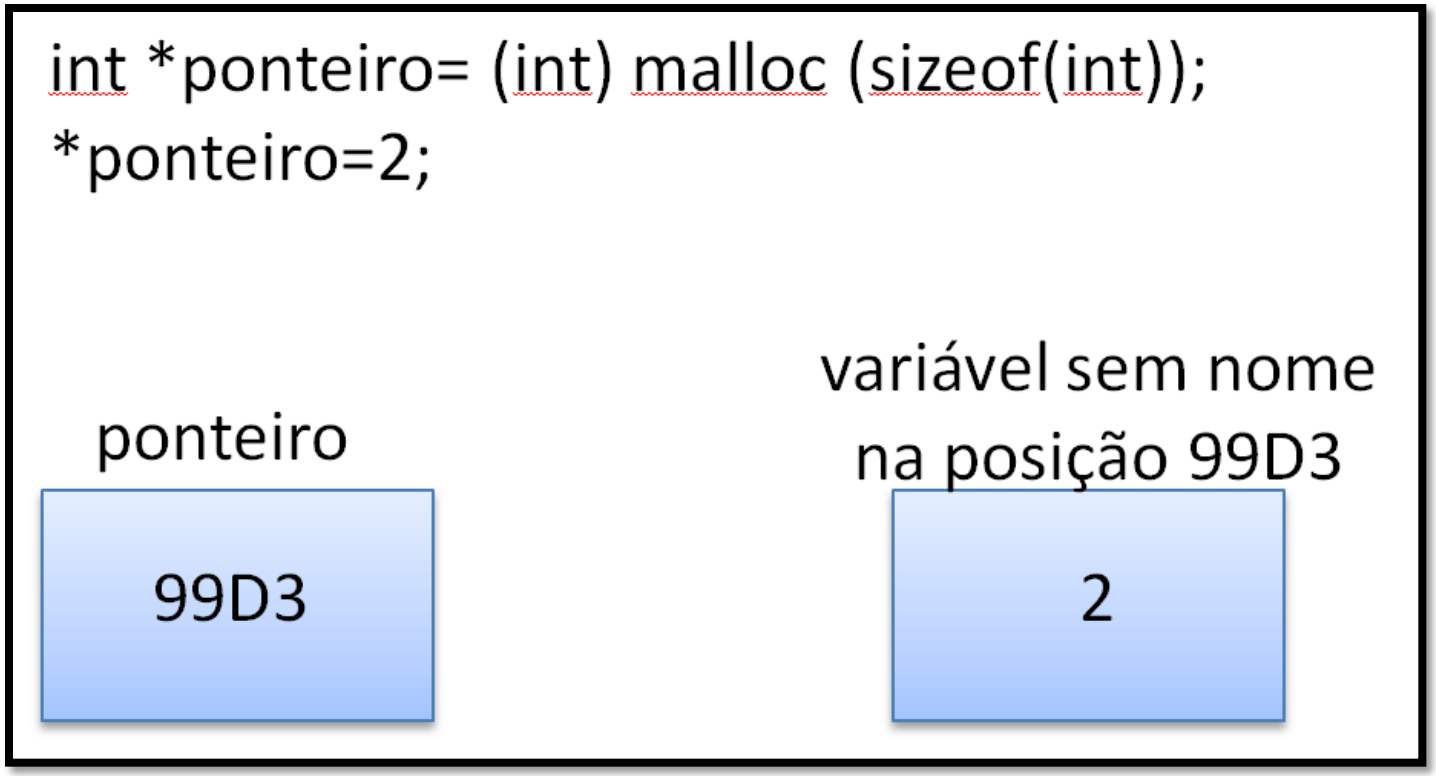


Propriedades de variáveis

- Variáveis podem ser caracterizadas por:
  - Nome;
  - Endereço;
  - Tipo;
  - Valor;
  - Tempo de vida;
  - Escopo de visibilidade.

Propriedades de variáveis

- Nome:
  - Nomeadas ou anônimas;
  - Ex.: variáveis criadas por malloc() são anônimas.



Propriedades de variáveis

- Endereço:
  - Onde o valor é armazenado;
  - Pode ser acessível ou não (aritmética de ponteiros):

```
// Exemplo em C:
char l;
char *m;
m = &l + 10;
printf("&l = %p\nm = %p\n*m = %c\n", &l, m, *m);

// Uma execução possível:
// &l = 0x7fff59cb0b8f
// m = 0x7fff59cb0b99
// *m = U
```

Propriedades de variáveis

- Sinonímia (*aliases*):

```
// Exemplo em C++:
int r = 0;
int &s = r;
s = 10;
```

- Em Java, tudo é referência, portanto é preciso ter atenção ao manipular cópias de objeto:

```
// Tipo primitivo:
int a = 10;
int b = a;
b = 20;

// Qual o valor de a?
```

X

```
// Objeto (java.awt.Point):
Point a = new Point(5, 5);
Point b = a;
b.x = 10;

// Qual o valor de a?
```

Propriedades de variáveis

- Tipo:
  - Especificação explícita: C, Java, etc.;
  - Sintática: versões iniciais de Fortran determinavam o tipo da variável pela letra inicial do seu nome. Applesoft BASIC exigia que o nome de variáveis tipo string terminassem com $;
  - Semântica: em ML o tipo é definido pelas operações que são efetuadas na variável.

Propriedades de variáveis

- Valor:
  - Determinado pela configuração de bits corrente e o tipo da variável:

```
// Exemplo em Java:
char c = 'a';
int i = c;
double d = i;
System.out.println("c=" + c + " i=" + i + " d=" + d);

// c=a i=97 d=97.0
```

Propriedades de variáveis

- Tempo de vida:
  - Locais, globais, anônimas, alocação estática;
  - Transientes ou persistentes.

Propriedades de variáveis

- Escopo de visibilidade:

```
// Exemplo em C/C++. O que é impresso ao final?
int x = 15;

void f() {
   int x = 10; static int y = 10;
   printf("x = %d, y = %d\n", x, y++);
}

int main() {
   f();
   f();
}
```

x = 10, y = 10
x = 10, y = 11

Não confundir escopo com tempo de vida!

Valores

- Atribuídos na definição ou durante o tempo de vida da variável (e dentro de seu escopo de visibilidade);
- Em variáveis compostas, atribuição pode ser completa ou seletiva:

```
// Exemplo em C:
struct data { int d, m, a; };
struct data f = {7, 9, 1965};
struct data g;

g = f;     // completa
g.m = 17;  // seletiva
```

E se g e f fossem referências a objetos Java? O que aconteceria com f.m ao final?

Constantes

- Aumentam a legibilidade dos programas;
- Podem ser:
  - Pré-definidas:

Não entender o que é o #define também causa isso!

```
// Exemplo em C:
char x = 'g';   // 'g' e 3 são constantes pré-definidas.
int y = 3;

// Erro de compilação: não é possível obter o endereço
// de uma constante pré-definida:
// int *w = &3;

// Erro de execução - falha de segmentação. Não se pode
// alterar uma string constante:
char* z = "bola";
z[0] = 'c';
printf("%s\n", z);
```

Constantes

– Declaradas:

```
// Exemplo em C:
const float pi = 3.1416;
float raio, area, perimetro;
raio = 10.32;
area = pi * raio * raio;
perimetro = 2 * pi * raio;
```

```
// Exemplo em C++:
int* x;
const int y = 3;
const int* z;
//   y = 4;     // Não permitidos pela linguagem.
//   y++;
//   x = &y;
z = &y;         // OK. const int* aponta para const int.
```

Armazenamento de variáveis e constantes

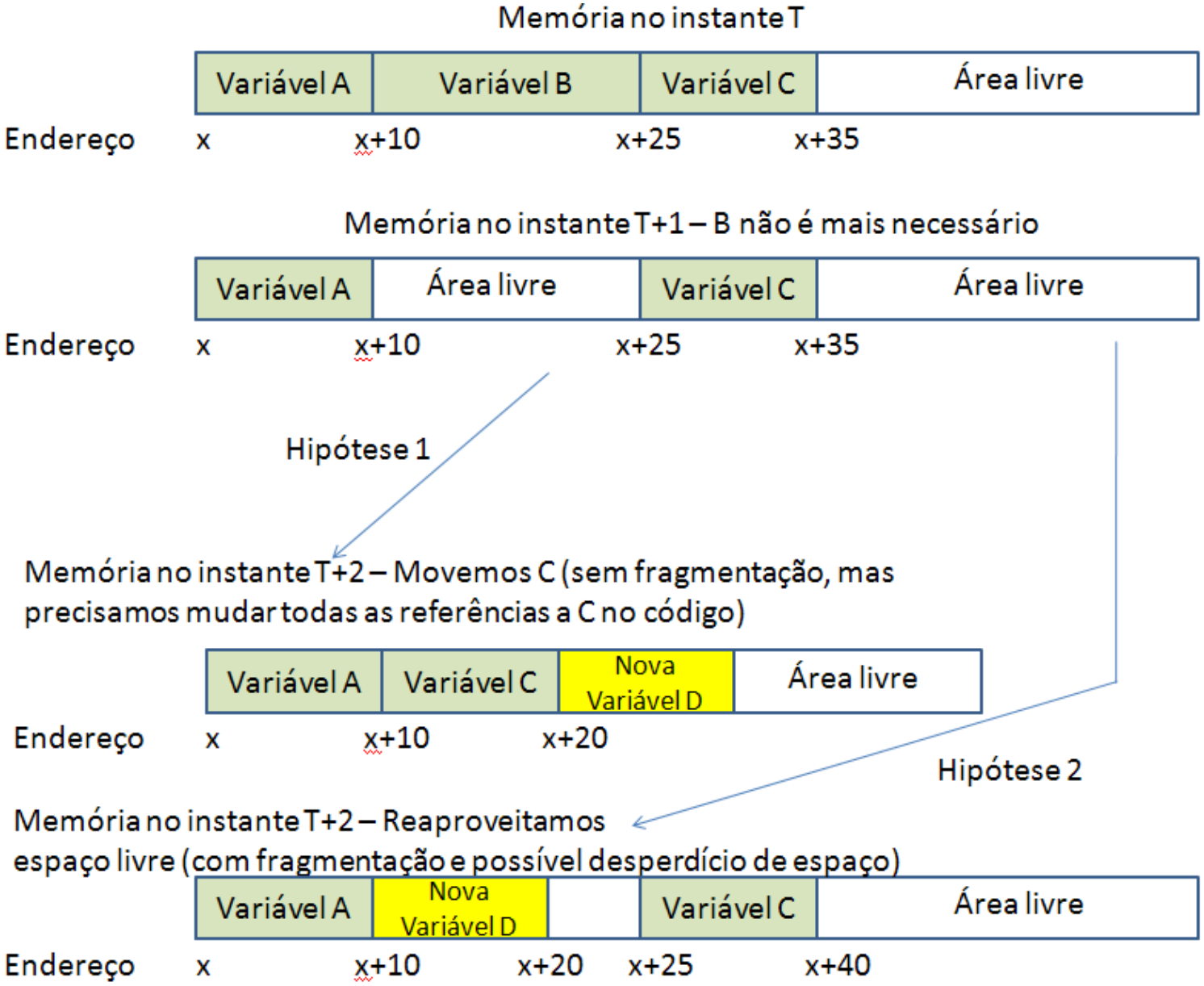
- Transientes:
  - Memória principal;
- Persistentes:
  - Memória principal;
  - Memória secundária.

Memória principal

- Enorme sequência contígua e finita de bits;
- Vetor de tamanho finito com elementos do tamanho da palavra do computador;
- Estratégias de alocação de variáveis e constantes:
  - Tempo de Carga (ex.: Fortran):
    - Super e subdimensionamento das variáveis;
    - Variáveis locais alocadas desnecessariamente;
    - Impedimento de uso de recursividade;
  - Alocação dinâmica contígua no vetor de memória:
    - Esgotamento rápido do vetor;
    - Desalocação e realocação pouco eficientes.

Memória principal

- Alocação dinâmica:



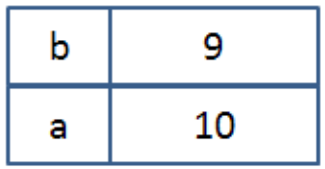
Pilha e monte

- Usada por linguagens Algol-like;
- Pilha:
  - Variáveis locais e parâmetros;
  - Regras de alocação bem definidas;
- Monte:
  - Variáveis de tamanho dinâmico;
  - Regra de alocação “indefinida”.

A pilha não é adequada para armazenar variáveis de tamanho dinâmico (realocação dos valores é ineficiente). O escopo destas variáveis não casa com suas regras bem definidas de alocação. Por isso se diz que o monte possui “regra de alocação indefinida”.

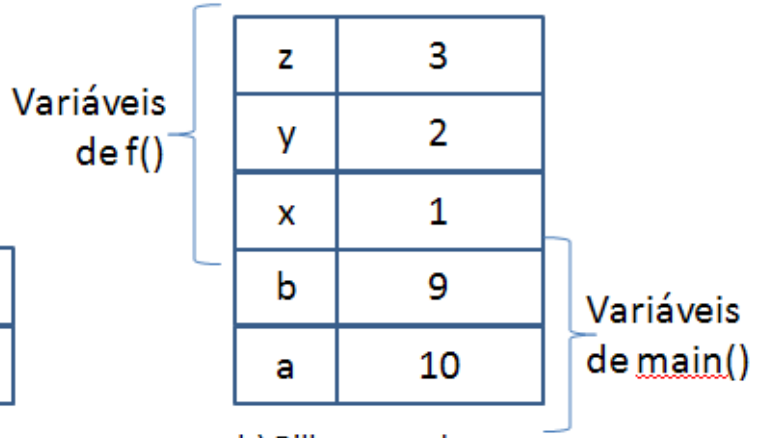
Pilha vs. monte

```
programa p :

f() {
  variáveis x, y e z
}

g() {
   variáveis x, y
}

main() {
  variáveis a,b
     :
     f()
     :
     g()
     :
}
```

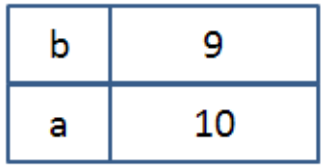


b	9
a	10

a) Pilha antes de main() chamar f()

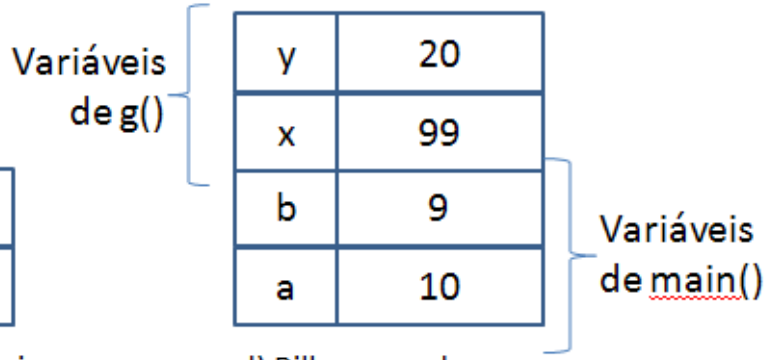


b) Pilha quando executamos f()



b	9
a	10

c) Pilha quando f() termina (na verdade, x, y e z ainda estão lá, mas ignoramos os valores, considerando suas posições vazias)

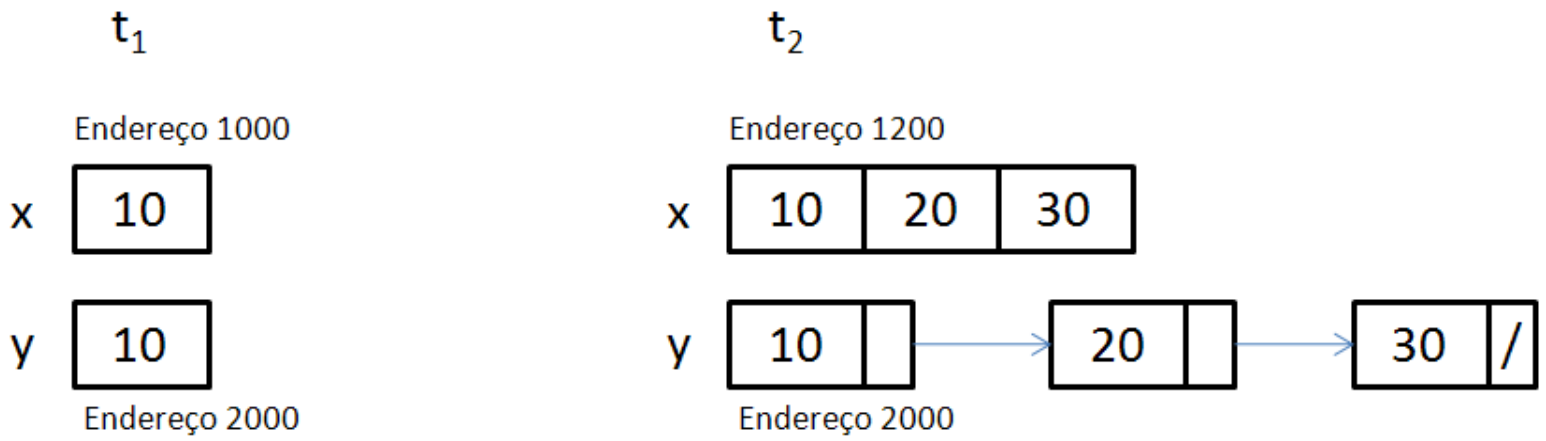


d) Pilha quando executamos g()

Pilha vs. monte

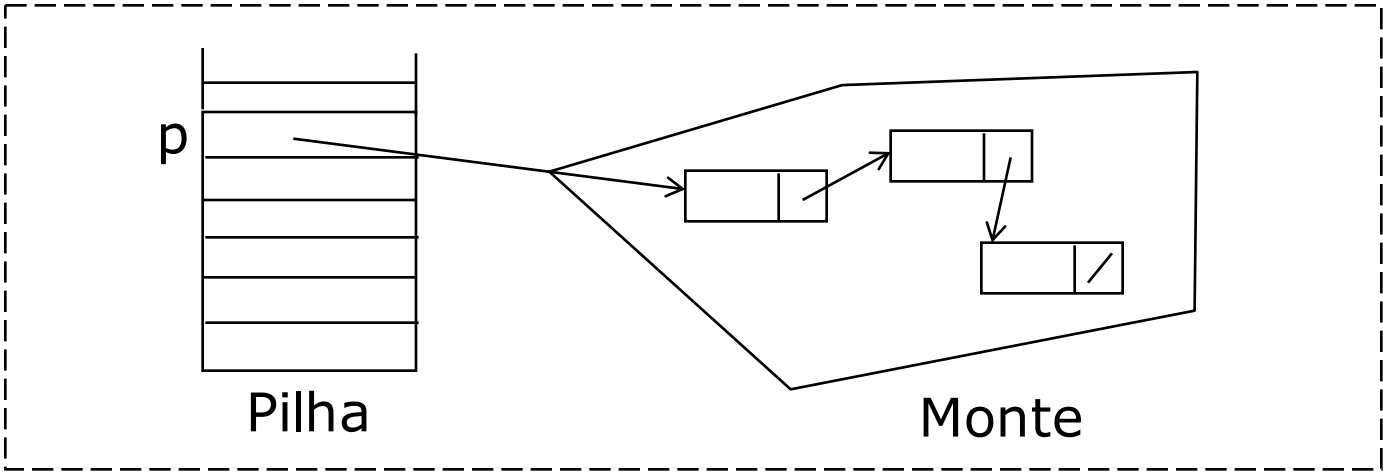
```
// Exemplo em C:
int *x = (int *)malloc(sizeof(int));
*x = 10;
x = (int *)malloc(3 * sizeof(int));
x[0] = 10; x[1] = 20; x[2] = 30;

// struct no *y = ...
// ...
```

Estratégias de aumento de tamanho: x vs. y. Trade-off entre custo do aumento e custo do acesso.



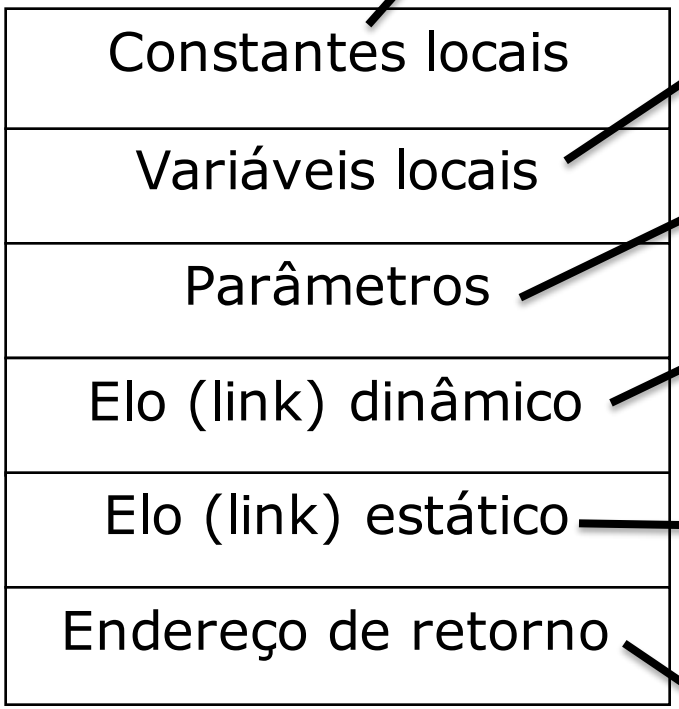
Por que pilha e monte?

- Só pilha: incompatível com alocação dinâmica;
- Só monte: manutenção da Lista de Espaços Disponíveis (LED) e Ocupados (LEO) é custosa;
  - Existem diversos algoritmos para LED/LEO.
- Ponteiros (referências) fazem ligação entre pilha e monte:



Gerenciamento da pilha

- Uso de registros de ativação (frame de pilha):



Constantes declaradas localmente.

Variáveis declaradas localmente.

Argumentos do subprograma.

Ponteiro para a base do RA anterior, para saber onde está o último RA quando o subprograma acabar;

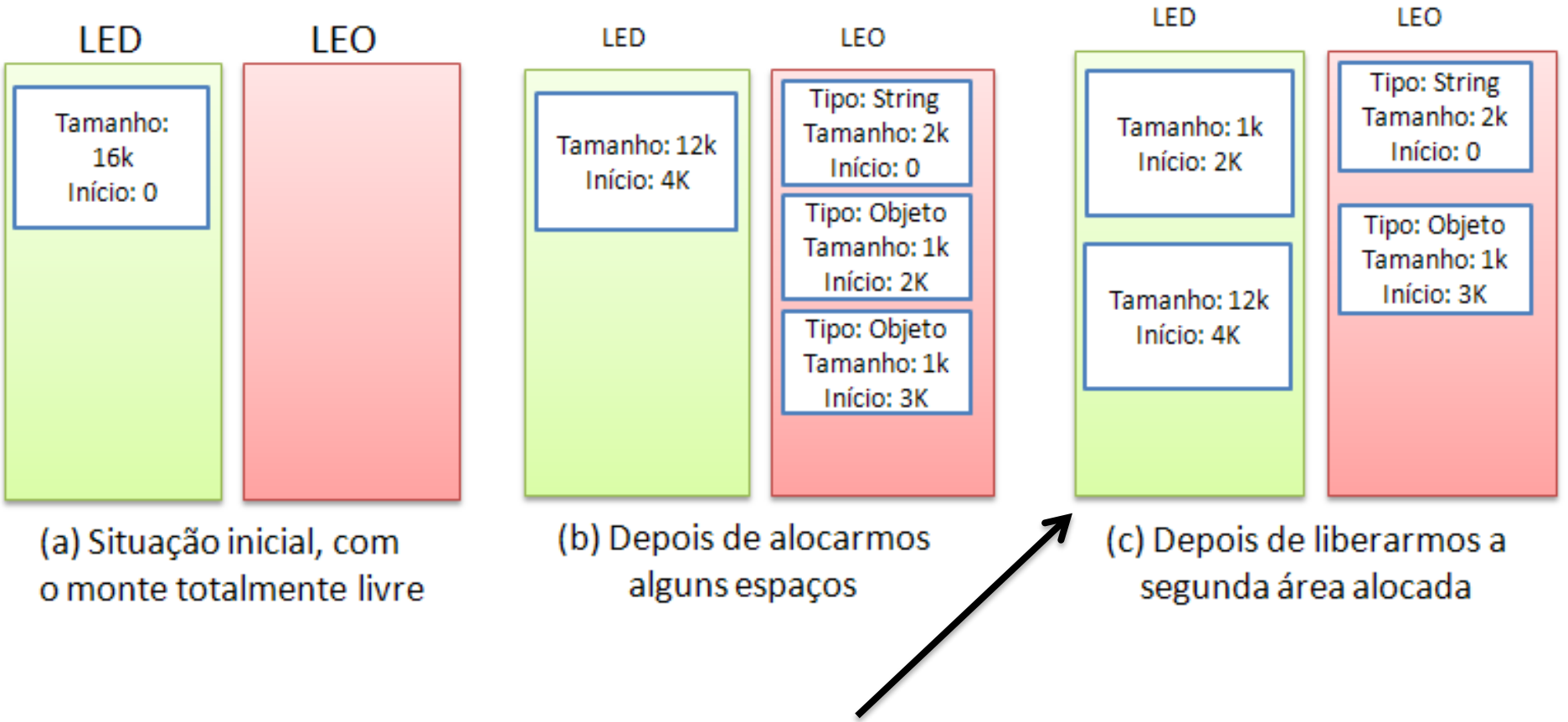
Ponteiro para a base do RA do subprograma externo, para acessar valores do escopo superior (somente para escopos aninhados).

Próxima instrução a ser executada depois que o subprograma terminar.

Exemplo: p. 102 do livro do Varejão.

Gerenciamento do monte

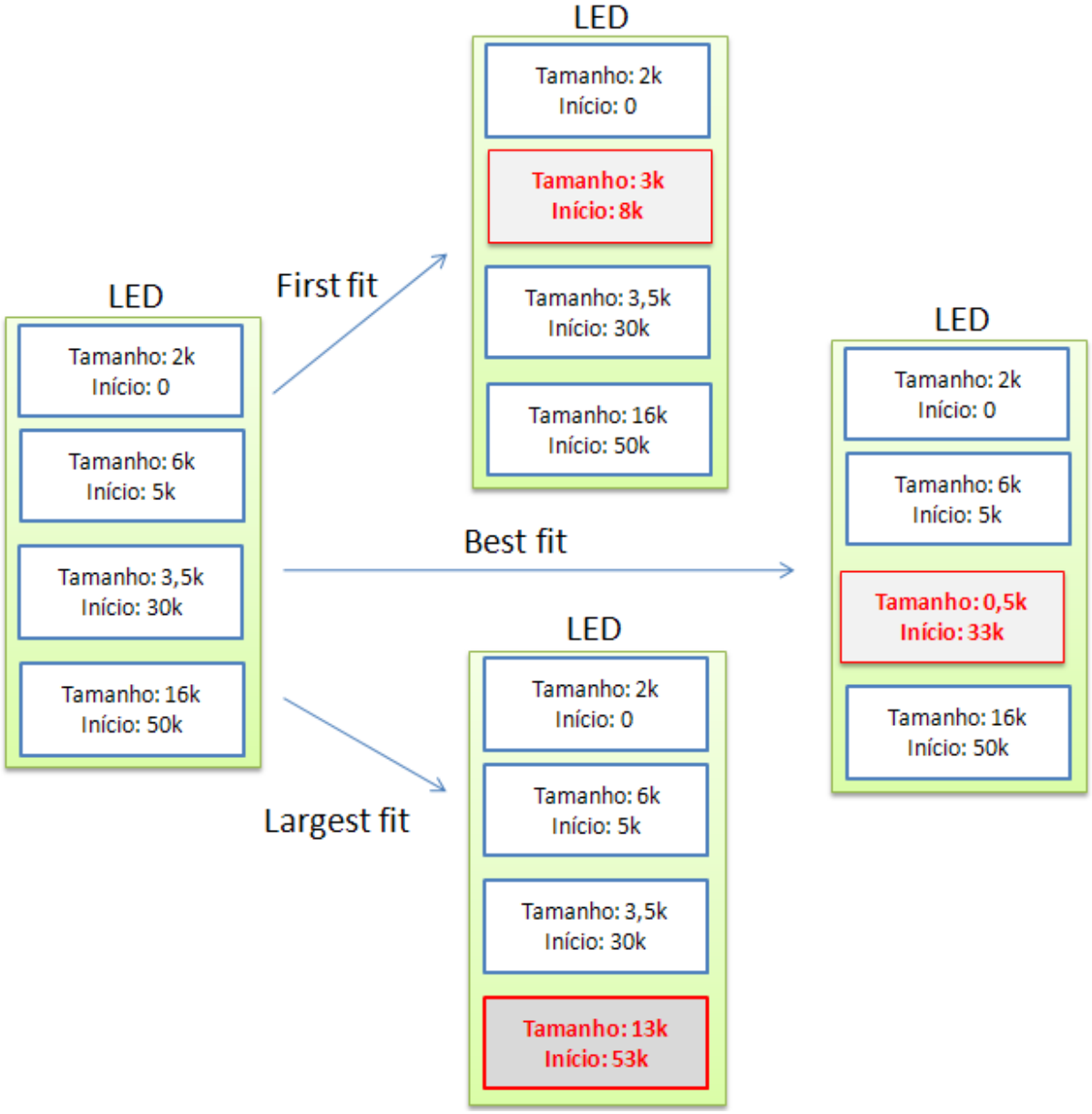
- Basicamente: gerenciamento de LED e LEO;



Monte fragmentado. Pode ser efetuada a desfragmentação. No entanto, é um processo custoso.

Gerenciamento do monte

- Políticas de alocação de memória:



Gerenciamento do monte

- Momento da desalocação definido ...:
  - Pelo programador: mais eficiente, menos confiável, mais trabalhoso, mais complexidade, possíveis erros (objetos pendentes, vazamento de memória, etc.);
  - Pela LP: implementação da LP mais complexa, falta de controle sobre a desalocação.
- Coletor de lixo de Java torna alocação no monte quase tão eficiente quanto na pilha.

Estratégias de coleta de lixo

- Contagem de referência:
  - Cada nó do monte contém um contador de referências atualizado em certos casos. Coletado quando contador = 0;
  - LED e LEO são listas encadeadas, portanto nós também possuem ponteiros;
  - Vantagem: distribui o overhead de coleta;
  - Desvantagens: não trata cadeias circulares, gasta memória e tempo com contadores.

Estratégias de coleta de lixo

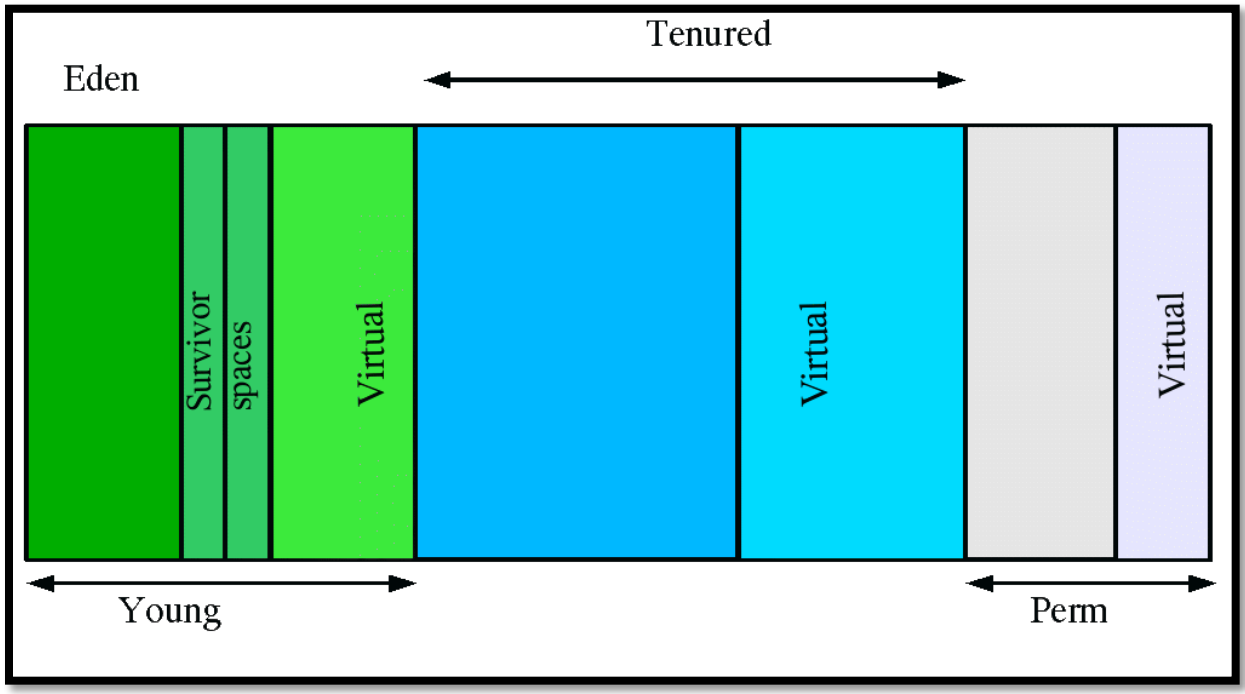
- Marcar-varrer:
  - Cada nó possui um bit de marca = 0;
  - Quando cheia, parte dos ponteiros na pilha e marca com 1 quem conseguiu alcançar;
  - Passa de novo (mas no monte todo) varrendo (coletando) nós com bit de marca 0;
  - Vantagens: recupera todo o lixo, só é chamado quando o monte está cheio;
  - Desvantagem: overhead concentrado.

Estratégias de coleta de lixo

- Coleta de cópias:
  - Como o marcar-varrer, porém com uma só passagem;
  - Copia os nós acessíveis para outra área de memória;
  - Vantagens: nenhum campo extra, passa uma só vez;
  - Desvantagem: gasta o dobro de memória.

Coletor de lixo do Java

- A partir da versão 1.2, Java separa o monte em gerações:
  - A maioria dos objetos morre cedo;
  - Quando uma coleta é feita, sobreviventes vão para a geração mais velha;
  - Coleta é feita por geração (menos objs).



Memória secundária

- Persistência de dados;
- Tipos de acesso:
  - Serial: começa na posição 0 e incrementa cursor;
  - Acesso direto/aleatório: pode navegar pelo arquivo, movendo-se para frente e para trás.
- Operações:
  - Abertura e fechamento;
  - Escrita e leitura, convertendo os dados de/para formato sequencial binário.

A área de Banco de Dados estuda especificamente este assunto, o que mostra a importância do mesmo.

Memória secundária

```
// Exemplo em C:
struct data { int d, m, a; };
struct data d = {7, 9, 1999};
struct data e;

int main() {
   FILE *p; char str[30];
   printf("Nome do arquivo: ");
   gets(str);
   if (!(p = fopen(str,"w"))) {
     printf("Erro!\n"); exit(1);
   }
   fwrite(&d, sizeof(struct data), 1, p);
   fclose(p); p = fopen(str,"r");
   fread(&e, sizeof(struct data), 1, p);
   fclose(p);
   printf("%d/%d/%d\n", e.a, e.m, e.d);
}
```

Memória secundária

```
// Exemplo de acesso a Banco de Dados em Java:
Connection conn = null;
String url="jdbc:hsqldb:hsql://localhost/javadiscos";
Class.forName("org.hsqldb.jdbcDriver");
conn = DriverManager.getConnection(url, "sa", "");

String sql = "SELECT nome, preco FROM CD;";
ResultSet rset = stmt.executeQuery(sql);

while (rset.next()) {
  String nome = rset.getString("nome");
  double preco = rset.getDouble("preco");
  System.out.println(nome + ": R$ " + preco);
}

rset.close();
conn.close();
```

Persistência ortogonal / transparente

- Mesmos tipos para variáveis persistentes e transientes;
- Nenhuma distinção entre o código que lida com variáveis persistentes e o que lida com variáveis transientes;
- Identificação de persistência através da percepção da continuidade do uso;
- Eliminação de conversões de entrada e saída (estima-se 30% do código);
- Não existem ainda na prática, “sonho” em LP.

Serialização / desserialização

- Variável transiente deve ser convertida de sua representação na memória primária para uma sequência de bytes na memória secundária;
- Ponteiros devem ser relativizados quando armazenados;
- Variáveis anônimas apontadas também devem ser armazenadas e recuperadas;
- Ao restaurar uma variável da memória secundária, os ponteiros devem ser ajustados de modo a respeitar as relações existentes anteriormente entre as variáveis anônimas;

Serialização / desserialização (cont.)

- Java oferece esse mecanismo na LP:
- Mecanismo de Java compensa diferenças das diferentes plataformas (ambientes computacionais);
- Serialização não é mecanismo ideal, mas facilita muito a vida do programador;
- Outros mecanismos de persistência:
  - Prevalência (vide http://prevayler.org);
  - Mapeamento Objeto/Relacional (vide http://hibernate.org ou procure “JPA” no Google).

Exemplo de serialização

```
public class Info implements Serializable {
  private String texto;
  private float numero;
  private Dado dado;

  public Info(String t, float n, Dado d) {
     texto = t; numero = n; dado = d;
  }

  public String toString() {
     return texto + "," + numero + "," + dado;
  }
}
```

Exemplo de serialização

```
import java.util.Date;

public class Dado implements Serializable {
  private Integer numero;
  private Date data;

  public Dado(Integer n, Date d) {
     numero = n; data = d;
  }

  public String toString() {
     return "(" + data + ":" + numero + ")";
  }
}
```

Exemplo de serialização

```
import java.util.Date;
import java.io.*;

public class Teste {
  public static void main(String[] args)
                                throws Exception {
    Info[] vetor = new Info[] {
       new Info("Um", 1.1f,
                new Dado(10, new Date())),
       new Info("Dois", 2.2f,
                new Dado(20, new Date()))
    };

    /* Continua... */
```

Exemplo de serialização

```
    ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("objs.dat"));
    out.writeObject("Os dados serializados foram:");
    out.writeObject(vetor);
    out.close();

    ObjectInputStream in = new ObjectInputStream(new
FileInputStream("objs.dat"));
    String msg = (String)in.readObject();
    Info[] i = (Info[])in.readObject();
    in.close();

    System.out.println(msg + "\n" + i[0]
                           + "\n" + i[1]);
  }
}
```

Conclusões

- Importância do papel de variáveis e constantes em LPs (imperativas);
  - Armazenamento em memória principal: pilha, monte, coleta de lixo;
  - Em memória secundária: I/O, persistência;
- Questões ao se estudar uma nova LP:
  - Permite o acesso ao endereço de memória? Permite definir constantes? Se comportam como constantes pré-existentes? Como é a desalocação de memória dinâmica? Como é o acesso à memória secundária? Oferece solução de persistência?

http://nemo.inf.ufes.br/