

Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo
Departamento de Computação - UFSCar
1º semestre / 2013
Aula 5

Material adaptado das aulas do Prof. Daniel Lucredio

Programação imperativa

Programação imperativa

- Contraste à programação declarativa
 - Programador diz “o que” o computador deve fazer
 - Computador decide “como” funcionar
 - Computador tem mais trabalho
 - Programador tem menos trabalho
- Na programação imperativa
 - Programador diz “como” o computador deve funcionar
 - Computador tem menos trabalho
 - Programador tem mais trabalho
 - Permite maior eficiência nas máquinas

Programação imperativa

As linguagens desse paradigma são muitas vezes chamadas de linguagens convencionais, procedurais ou imperativas

- São linguagens que “reconstroem” uma máquina para torná-la mais conveniente para programação
- Assim como:
 - Programação lógica busca imitar lógica de primeira ordem
 - Programação funcional busca imitar funções matemáticas
 - Programação imperativa busca imitar o modelo de uma máquina real
- Na nossa realidade:
 - Modelo computacional de Turing
 - Máquinas vs Funções
 - Mecânico vs Analítico
 - Arquitetura de von Neumann

Programação imperativa

- Computadores de von Neumann
 - Unidade lógica e aritmética
 - Cálculos matemáticos e lógicos (simples)
 - Unidade de controle
 - Controle síncrono de processamento
 - Memória
 - Armazenamento de informação
 - Barramento
 - Transferência de informação entre os demais componentes

Programação imperativa

- Unidade de processamento
 - (Micro) instruções que executam pequenas tarefas
 - Processamento numérico (ULA)
 - Conceito de “comandos” (daí o nome “imperativa”)
- Unidade de controle
 - Execução das instruções / processamento sequencial
 - Conceito de “estados”
- Memória
 - Palavras de memória
 - Conceito de “variáveis”
- No geral
 - Instruções de armazenamento/atribuição
 - Programa armazenado

Programação imperativa

- Evoluiu para a programação estruturada
 - Estruturas / blocos
 - Trechos contíguos de um programa
 - Regras de escopo estático / dinâmico
 - Fluxo de controle mais avançado
 - Explícito na estrutura sintática do programa
 - Ex: “if” vs “jnz”
 - Conceito de subrotinas
 - Procedimentos / funções
 - Recursividade
 - Passagem de parâmetros
 - Outros conceitos mais avançados

Nomes

Nomes

- Muitas instruções precisam ler/escrever na memória
 - por intermédio de registradores
 - Ou pelo controle
 - Atribuição
 - Salto (GOTO)
 - Chamada de subrotina
- A memória é sequencial (imagine uma fita)
 - Cada célula de memória armazena um dado
 - Cada célula possui um endereço

Nomes

- Programar somente com endereços
 - Difícil (é necessário calcular endereços/deslocamentos)
 - Fácil de cometer erros
 - Inflexível (se for necessária alocação dinâmica)
- Uso de nomes facilita a escrita de programas
- São “apelidos” para determinadas construções de um programa
 - Geralmente dizem respeito a uma posição de memória (endereço) de:
 - Variáveis
 - Funções
 - Parâmetros
 - Rótulos
 - Blocos
 - Etc

Nomes

- Ex:
 - $x + y$
 - conteúdo do endereço apontado por “x” somado ao conteúdo do endereço apontado por “y”
 - GOTO rótulo1
 - execute em seguida a instrução indicada por “rótulo1”
 - soma(2, 2)
 - Execute a subrotina indicada por “soma” e em seguida retome a execução atual

Nomes

- Fortemente associados aos compiladores
 - Que efetivamente implementam o uso de nomes em um programa
- Ou seja, é papel do compilador
 - Traduzir nomes em endereços
 - Tabela de símbolos
 - Mais detalhes nas disciplinas de construção de compiladores

Nomes

- Normalmente são cadeias de caracteres
 - Começando com letra ou algum símbolo não-numérico

- Ex:

`Id : ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9')*;`

- Compilador reconhece um nome
 - Dependendo do contexto (regra sintática) onde ele aparece
 - Decide se é uma variável / procedimento / rótulo / etc
 - Define seu escopo (estático)
 - Decide se já foi declarado ou não
 - Outras tarefas

Nomes vs identificadores

- Nomes às vezes são chamados de identificadores
 - Mas existe uma distinção
 - Um identificador é normalmente uma unidade léxica
 - Tem um significado local
 - Para variáveis locais, identificadores são nomes
 - Mas um nome pode ser composto de vários identificadores
 - Ex: pacotes Java
 - `List` é um identificador, que aponta para uma estrutura (classe)
 - `java` e `util` são identificadores, que apontam para pacotes
 - `java.util.List` é um nome, que aponta para uma estrutura predefinida da biblioteca Java
- Ou seja, em um contexto global, um identificador nem sempre é um nome

Nomes

- Curiosidades
 - As primeiras linguagens usavam nomes de um único caractere
 - Natural para matemáticos
 - FORTRAN I permitia até seis caracteres em um nome
 - Java/Ada não possuem limite
 - O caractere sublinhado (“_”) é comumente aceitável em nomes
 - Semelhança com um espaço, permitindo nomes mais legíveis
- Algumas linguagens fazem distinção entre maiúsculas e minúsculas
 - Prejuízo à legibilidade / capacidade de escrita
 - Ou não! Essa questão é controversa

Palavras especiais

- Problema técnico de compiladores
 - Nomes/Identificadores são regras gerais
 - Ou seja, o analisador léxico “captura” todas as sequências de caracteres, achando que são identificadores
 - Mas é desejável termos palavras que “parecem” identificadores (lexicamente) mas que não são
 - Exs: if, return, int...
 - O compilador precisa analisar o contexto da regra sintática para decidir se a cadeia lida é uma palavra especial ou um nome
 - Exs: (FORTRAN)
 - INTEGER REAL
 - REAL INTEGER

Palavras reservadas

- Uma decisão no projeto de uma linguagem
 - Proibir o uso de algumas palavras como nomes/identificadores
 - Facilita o trabalho do compilador
 - Deixa o programa mais legível
- Ex: em Java é proibido declarar um método cujo nome é “return”
- Outro exemplo, a palavra “const” foi reservada em Java, mas não é usada para nada
 - Inicialmente seria usada como um suporte para programas em C++, mas a idéia foi descartada

Variáveis

Variáveis

- Representam células de memória
 - Ou um conjunto de células
- Unidade básica de armazenamento na arquitetura de von Neumann
- Projetada pra ser controlada explicitamente pelo programador
 - O conceito é DIFERENTE da incógnita matemática
 - Como acontece nos paradigmas lógico e funcional

Variáveis

- Uma variável pode ser caracterizada como uma 6-tupla:

<nome, endereço, valor, tipo, tempo de vida, escopo>

Nome de uma variável

- Um nome pode ser usado como um apelido para uma variável
- Mudança das linguagens de máquina para linguagens de montagem
 - Uso de nomes ao invés de endereços
 - Programas mais legíveis e fáceis de serem escritos e mantidos
 - Resolveu o problema do endereçamento absoluto
 - O tradutor que converte nomes para endereços pode escolher os endereços de forma conveniente / automática

Endereço de uma variável

- Local da memória associado a uma variável
 - Também chamado de “*l-value*” ou “*valor-l*”, devido ao uso em uma atribuição
- Uma variável pode ter endereços diferentes em momentos diferentes (diferentes chamadas para um subprograma que tem uma variável local), que corresponde a instâncias da variável.
- Várias variáveis podem ter o mesmo endereço, o que é chamado de *aliasing* / apelido
- Exemplos:
 - Union em C e C++
 - Dois ponteiros apontando para a mesma posição de memória

Tipo de uma variável

- Determina a faixa de valores que ela pode ter
- Conjunto de operações permitidas para os valores do tipo
- Ex:
 - tipo INTEGER em FORTRAN (algumas implementações) especifica um intervalo de valores de -32.768 a 32.767 e as operações aritméticas tradicionais
 - tipos de classes em Java (referências) definem operações de instanciação (new), chamadas de métodos, etc...

Valor de uma variável

- É o conteúdo da variável
 - Uma ou mais células de memória
- Às vezes é conveniente falarmos de células abstratas
 - Uma célula abstrata armazena exatamente o valor de uma variável
 - Útil para cálculos de deslocamento em vetores, por exemplo
- Também chamado de “*r-value*” ou “*valor-r*”, devido ao uso em uma atribuição

Tempo de vida e escopo

- Tempo de vida de variável
 - Intervalo de tempo durante o qual uma área de memória está amarrada a uma variável
- Escopo de variável
 - Trecho do programa onde uma variável é conhecida
- Para compreendermos melhor estes conceitos, é necessário estudar o conceito de amarração/vinculação

Conceito de vinculação

Vinculação

- Associação entre elementos
- Ex:
 - Entre variável e tipo
 - Entre símbolo e operação
 - Entre tipo de dados e valores possíveis
- Tempo de vinculação
 - Estática: em tempo de compilação, antes da execução, permanecendo inalterada
 - Dinâmica: definida em tempo de execução, ou pode ser modificada durante a execução
- Sebesta cita outros, mas não são importantes para a nossa discussão

Vinculação de atributos a variáveis

- 6-tupla
 - Nome
 - Endereço
 - Valor
 - Tipo
 - Tempo de vida
 - Escopo
- Todo atributo pode sofrer vinculação dinâmica/estática
- Estudaremos primeiro a vinculação de tipo

Tipos

Vinculação de tipos

- Variável possui um tipo de dados
- Dois aspectos importantes
 - Maneira como o tipo é especificado
 - Quando a vinculação ocorre

Declarações de variáveis

- Uma declaração de variável cria uma vinculação estática
 - Em tempo de compilação
 - Ou seja, analisando o código é possível definir (vincular) um tipo a uma variável
- Declarações explícitas
 - Instrução que lista nomes e especifica o tipo
- Declarações implícitas
 - Uso de convenções padrão em vez de instruções
 - A primeira ocorrência constitui sua declaração implícita
 - Ex: No FORTRAN, variáveis que começam com I, J, K, L, M ou N são consideradas inteiras

Vinculação dinâmica de tipos

- O tipo de uma variável é determinado durante a execução
 - Pode mudar durante a execução
- A variável é vinculada a um tipo durante a atribuição
 - Variável do lado esquerdo da atribuição é vinculada ao tipo da expressão do lado direito
- Vantagem: flexibilidade na programação
- Desvantagens:
 - diminui a capacidade de detecção de erros
 - custo maior para verificação de tipos durante a execução
- Normalmente usada em interpretadores
 - Mais fácil de detectar e mudar tipos
 - Overhead fica menos evidente

Vinculação dinâmica de tipos

- Exemplo em JavaScript:

```
list = [34.5 4.8 5.1];
```

- O tipo de list passa a ser um array unidimensional de tamanho 3

```
list = 47;
```

- Neste ponto, a mesma variável list se torna um tipo escalar.

Inferência de tipos

- Processo de determinação de um tipo, em tempo de compilação / execução
 - Exemplo: ML
- Ex: `fun circumf(r) = 3.14159 * r * r;`
 - Resultado será um número real
- Ex: `fun vezes10(x) = 10 * x;`
 - Resultado será um inteiro
- Ex: `fun quadrado(x) = x * x;`
 - Função rejeitada, pois é impossível inferir o tipo
- Ex:
 - `fun quadrado(x : int) = x * x;`
 - `fun quadrado(x) = (x:int) * x;`
 - `fun quadrado(x) = x * (x:int);`

Verificação de tipos

- Assegurar que os operandos de um operador sejam de tipos compatíveis
- São considerados operadores e operandos:
 - operadores usuais (aritméticos, relacionais, etc)
 - subprogramas (operadores) e parâmetros (operandos)
 - atribuição (operador) e variável / expressão (operandos)
- Os tipos de operandos são compatíveis com um operador se:
 - são aceitos pelo operador ou
 - podem ser convertidos implicitamente pelo compilador (coerção)
- Erro de tipo: aplicação de um operador a um operando de tipo não apropriado.

Verificação de tipos estática

- Feita antes da execução
- Vinculação de tipo estática permite quase sempre a verificação de tipo estática
 - Vantagem da detecção de erros em tempo de compilação: quanto antes o erro for encontrado, menor o custo.
 - Desvantagem: redução da flexibilidade para o programador

Verificação de tipos dinâmica

- Feita durante a execução do programa
- Vinculação de tipo dinâmica exige verificação de tipo dinâmica
- Sistema deve manter informações sobre tipos durante a execução
- Verificação de tipo deve ser dinâmica quando a mesma posição de memória pode armazenar valores de tipos diferentes em momentos diferentes durante a execução
 - Registros variantes (Ada/Pascal)
 - Union (C/C++)
 - Equivalence (FORTRAN)

Tipagem forte

- Uma linguagem é considerada fortemente tipada se erros de tipo podem sempre ser detectados.
- Isso requer que os tipos dos operandos possam sempre ser determinados, seja em tempo de compilação ou em tempo de execução
- Uma linguagem fortemente tipada deve permitir também a detecção, em tempo de execução, de uso de valores de tipo incorreto de variáveis que podem armazenar valores de mais de um tipo.

Tipagem forte

- FORTRAN 95 não é fortemente tipada
 - EQUIVALENCE permite acesso a mesma posição de memória por variáveis de tipos diferentes
- Pascal não é fortemente tipada
 - Registros variantes permitem omissão da “tag” que armazena o tipo corrente de uma variável
- C, C++ não são fortemente tipadas
 - Permitem funções para as quais os parâmetros não são verificados quanto ao tipo
 - Estruturas do tipo union não são checadas

Tipagem forte

- Ada é quase fortemente tipada
 - Permite que o programador suspenda as regras de verificação de tipos solicitando que a verificação não seja feita para um tipo particular de conversão
- Java e C# são fortemente tipadas no mesmo sentido que ADA
 - permitem conversão explícita de tipo, que pode resultar em erro de tipo

Conversão de tipos

- Conversão de tipo implícita (coerção)
 - iniciada pelo compilador
- Conversão de tipo explícita (casting)
 - solicitada pelo programador
- As regras de coerção de uma linguagem tem efeito na verificação de tipo
- O valor da tipagem forte é enfraquecido pela coerção
- Linguagens que permitem muitas conversões implícitas (Fortran, C, C++) são menos seguras/confiáveis do que as que não permitem (Ada, Java, C#)

Conversão de tipos

- Exemplo em JAVA:

```
int a;
```

```
float b, c, d;
```

```
....
```

```
d = b * a;
```

- Java permite expressões de modo misto, assim é feita coerção nesse caso e “a” é convertido para float
- Supondo que houve um erro de digitação e era para ser “c” no lugar de “a”, esse erro não é detectado

Conversão de tipos

- Exemplo em ADA:

```
A : Integer;
```

```
B, C, D : Float;
```

```
....
```

```
C := B * A;
```

- O compilador encontra um erro de tipo, pois Float e Integer são tipos que não podem ser misturados no operador “*”
- Ada permite poucos casos de conversão implícita, para melhorar a verificação de erros

Armazenamento (endereço) e tempo de vida

Vinculação de armazenamento e tempo de vida

- Alocação
 - Vinculação de uma área da memória a uma variável
 - Área da memória fica “reservada” à variável
- Desalocação
 - Desvincular a área da memória da variável
 - A memória fica disponível novamente
- Tempo de vida
 - Tempo durante o qual uma variável está vinculada a uma área da memória
 - Tempo entre alocação e desalocação

Vinculação de armazenamento e tempo de vida

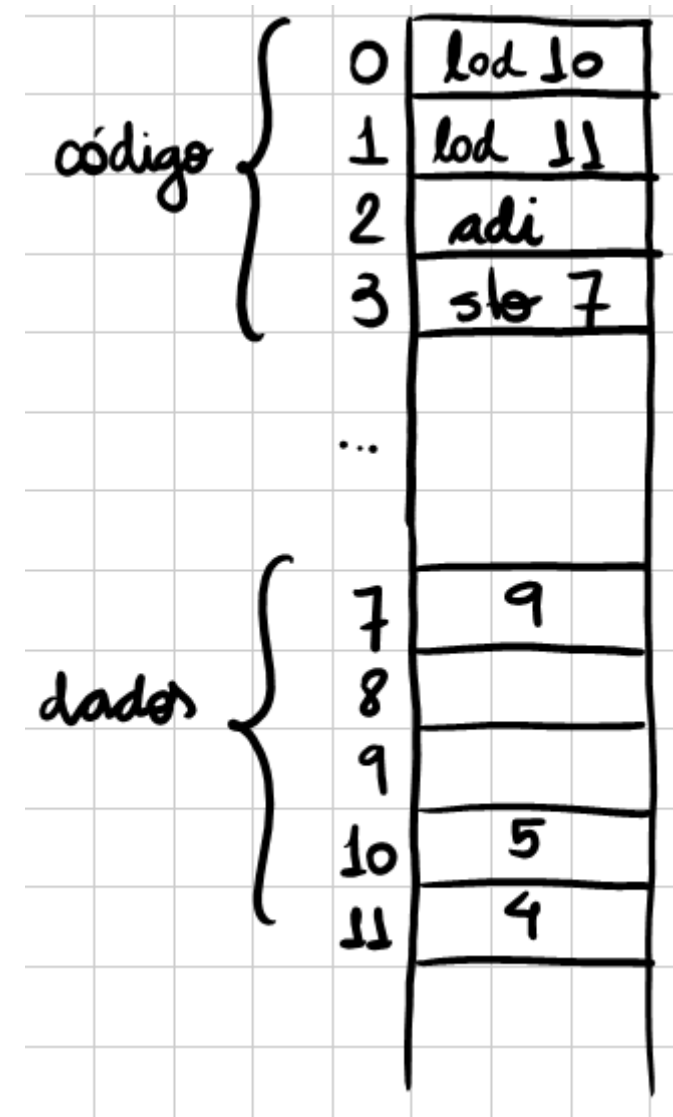
- Quatro categorias, com regras de vinculação distintas
 - Variáveis estáticas
 - Variáveis dinâmicas na pilha
 - Variáveis dinâmicas no monte (heap) explícitas
 - Variáveis dinâmicas no monte (heap) implícitas
- Mas antes, vamos estudar um pouco sobre como a memória é organizada durante a execução

Organização da memória

- Tipicamente
 - Registradores
 - Memória (RAM) – mais lenta e de acesso endereçável não sequencial
 - Área de código
 - Para a maioria das linguagens compiladas, não pode ser alterada durante a execução
 - É fixada durante a compilação
 - Área de dados
 - Parcialmente fixada durante a compilação (por exemplo, variáveis externas e estáticas em C)

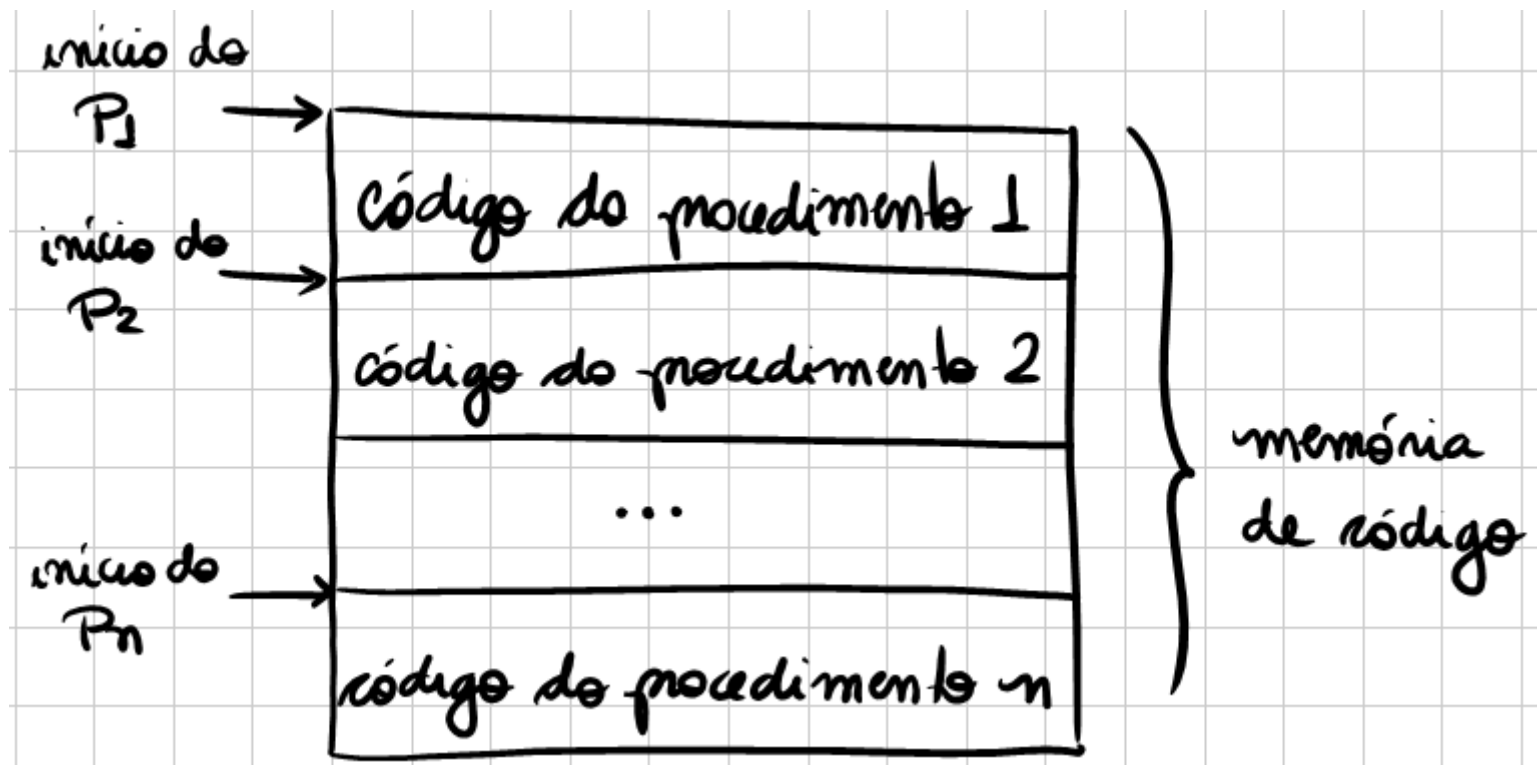
Organização da memória

- Porção de memória para se carregar o código gerado na compilação
- Porção de memória para se carregar e trabalhar com os dados:
 - variáveis,
 - parâmetros e dados de procedimento,
 - valores temporários e de manutenção do próprio ambiente



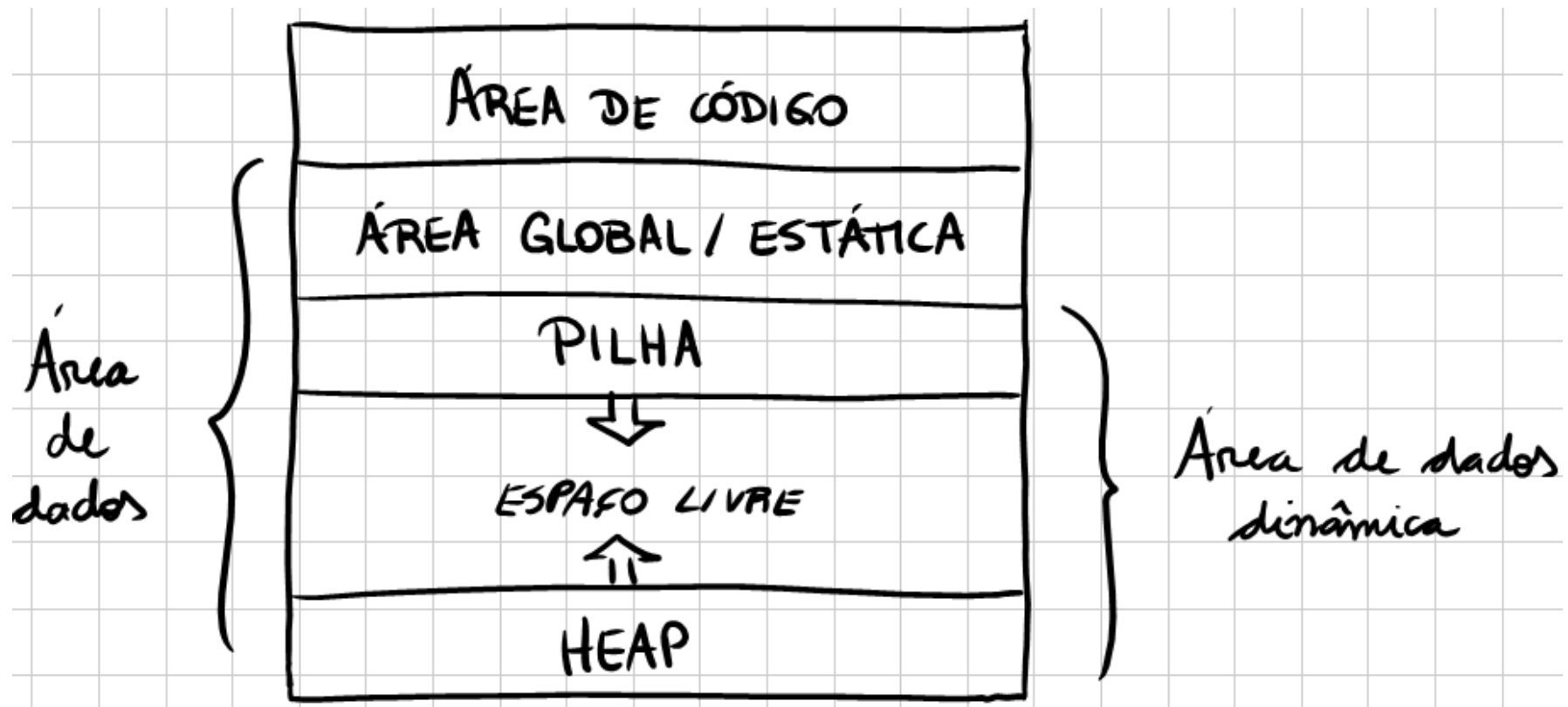
Organização da área de código

- Endereços computáveis durante a compilação



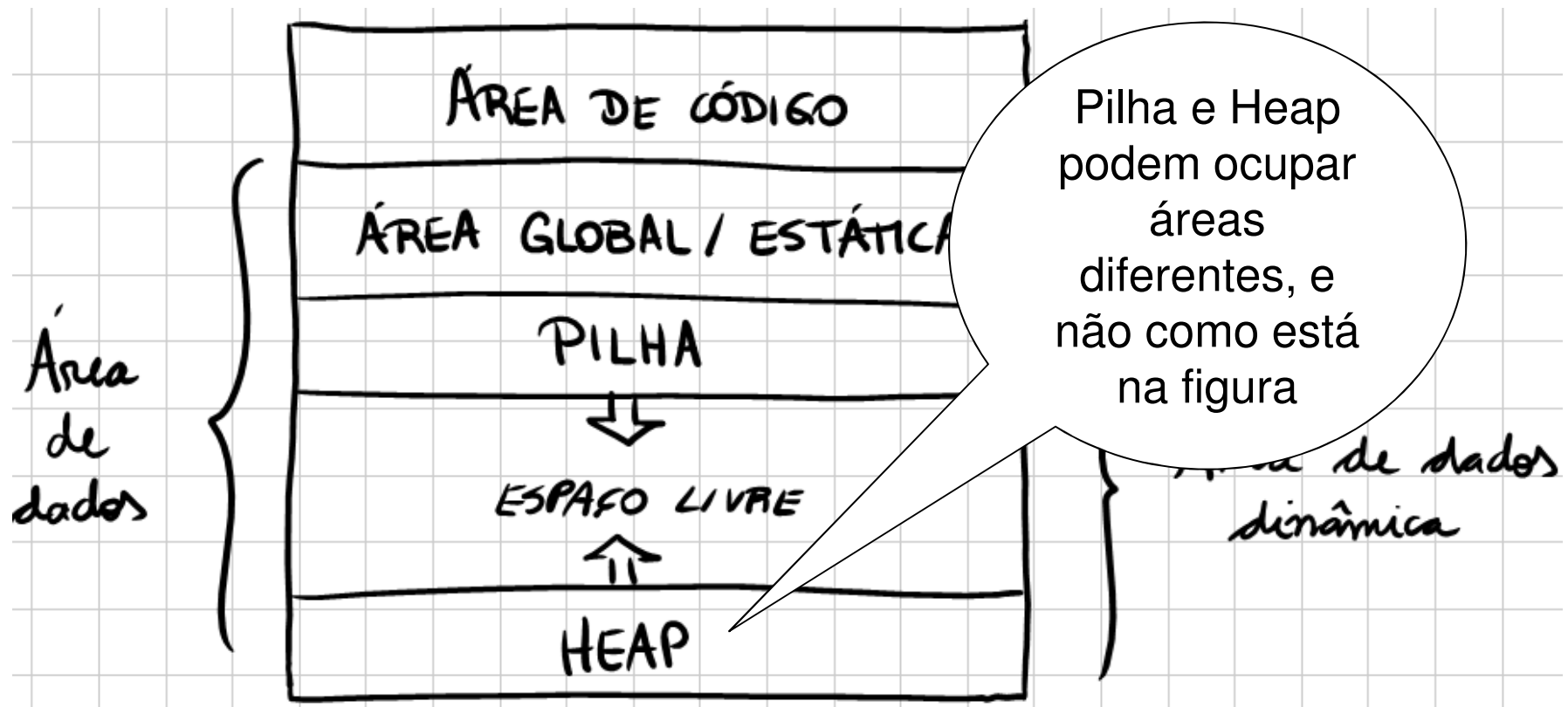
Organização da área de dados

- Área global/estática
- Pilha, para dados dinâmicos cuja alocação é do tipo LIFO (last in, first out)
- *Heap* (ou monte), para dados dinâmicos com alocação livre
 - Neste contexto, é uma memória linear simples (e não uma árvore)



Organização da área de dados

- Área global/estática
- Pilha, para dados dinâmicos cuja alocação é do tipo LIFO (last in, first out)
- *Heap* (ou monte), para dados dinâmicos com alocação livre
 - Neste contexto, é uma memória linear simples (e não uma árvore)



Variáveis estáticas

- A alocação de memória ocorre antes da execução do programa, nos casos de:
 - variáveis globais – devem ser acessíveis a todo o programa
 - variáveis locais estáticas – são declaradas dentro de um subprograma mas devem reter valores entre execuções separadas do subprograma (sensíveis à história).
- Vantagens:
 - eficiência – endereçamento direto (depende da implementação)
 - não exige custo adicional para alocação e liberação de memória

Variáveis estáticas

- Desvantagem:
 - pouca flexibilidade – linguagens que usam apenas alocação estática não dão suporte à subprogramas recursivos nem a compartilhamento de memória (Subprogramas que executam em momentos diferentes e tem variáveis de grande dimensão).
- Exemplos:
 - FORTRAN I, II, IV – todas as variáveis eram estáticas
 - C, C++, JAVA – permitem definir uma variável local como estática
 - Pascal – não possui variáveis estáticas

Variáveis dinâmicas de pilha

- Variáveis dinâmicas de pilha são alocadas na pilha de execução
 - A alocação de memória da variável é feita quando a declaração dessa variável é elaborada, mas o tipo da variável é amarrado estaticamente.
- Elaboração: processo de alocação que acontece no momento em que é iniciada a execução do código onde aparece a declaração (ativação).

Variáveis dinâmicas de pilha

- Variáveis declaradas no início de um procedimento (função, método):
- Chamada do procedimento
 - ativação
 - elaboração
 - execução
- Término da execução do procedimento
 - retorno do controle à unidade chamadora
 - desalocação da memória

Variáveis dinâmicas de pilha

- Variáveis declaradas no início de um comando composto

```
main ( ) {  
    int i = 0, x = 10;  
    while (i++ < 100) {  
        float z = 3.34;  
        ...  
    }  
}
```

- A elaboração (alocação de memória) acontece no início da execução do comando ou no início da execução da unidade.

Variáveis dinâmicas de pilha

- Algumas linguagens (C++, JAVA) permitem que variáveis sejam declaradas em qualquer lugar do programa.
- Em algumas implementações, todas as variáveis declaradas em uma função ou método são amarradas à memória no início da execução da função, mesmo que sua declaração não apareça no começo.
- A variável se torna visível a partir da sua declaração
- Conveniente para procedimentos recursivos

Variáveis dinâmicas no monte explícitas

- Variáveis dinâmicas de heap (monte) são células de memória sem nome
 - coleção de células de memória de uso desorganizado, pois é imprevisível determinar o local exato de cada variável
- São alocadas explicitamente por instruções do programador, por meio de:
 - Um operador (C++ ou Ada)
 - Chamada para um subprograma (C)
- Algumas linguagens tem também um operador para liberar a memória
 - Só podem ser referenciadas por ponteiros ou variáveis de referência
 - São usadas frequentemente para implementar estruturas dinâmicas que crescem e diminuem durante a execução

Variáveis dinâmicas no monte explícitas

- Exemplo: C++

```
new operando : tipo
```

- Aloca uma posição de memória na heap e retorna um ponteiro para essa posição

```
int *intnode;
```

```
...
```

```
intnode = new int;
```

```
/* aloca uma posição de memória tipo int */
```

```
...
```

```
delete intnode;
```

```
/* desaloca a posição de memória para a  
qual intnode aponta */
```

- C++ tem o operador delete porque não tem liberação de memória implícita (*garbage collection*)

Variáveis dinâmicas no monte explícitas

- Em JAVA:
- Todos os dados, exceto primitivos, são objetos, portanto são dinâmicos de heap e acessados por variáveis de referência
- A desalocação não é feita explicitamente, mas implicitamente pelas rotinas de coleta de lixo (garbage collection)
- Desvantagens:
 - Dificuldade de usar corretamente (ponteiros e variáveis de referência)
 - Custo adicional de referência, alocação e desalocação

Variáveis dinâmicas no monte implícitas

- São alocadas na heap quando valores são atribuídos
- Todos os atributos são definidos cada vez que os valores são atribuídos
- Exemplo em JavaScript:
 - Lista = [56.1 7.8 43.5 5.6]
- Vantagem: alta flexibilidade
- Desvantagens:
 - Custo elevado, todos os atributos são dinâmicos
 - Não permite verificação de erros pelo compilador

Escopo / vinculação de nome

Escopo

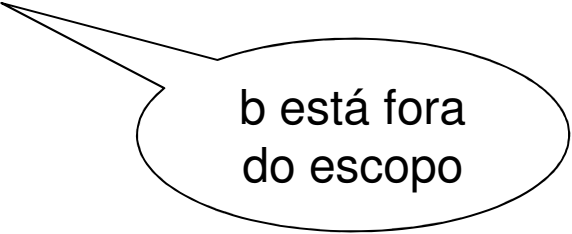
- Determinação da visibilidade de uma variável
 - Uma variável é visível em uma instrução se puder ser referenciada nessa instrução
- Escopo de uma variável
 - Conjunto de instruções nas quais a variável é visível
- Regras de escopo determinam a vinculação de **nomes** a variáveis

Escopo

- O que é escopo?
 - É a parte do programa em que uma determinada declaração (variável, função, procedimento) é “válida”

- Ex:

```
int a = 3;  
{  
    int b = 4;  
}  
int c = a + b;
```



b está fora
do escopo

Escopo estático e estrutura de blocos

- Muitas LPs utilizam escopo estático
 - Ou seja, o compilador consegue verificar o escopo (em tempo de compilação, óbvio)
- Ex:
 - C e Java utilizam estrutura de blocos { e }
 - Algol e Pascal utilizam begin e end
- Estrutura de blocos
 - Permite a criação de blocos “aninhados”
- Subprogramas
 - Seguem a mesma regra de escopos

Escopo

- Algumas linguagens (C++, JAVA) permitem que variáveis sejam definidas em qualquer lugar do bloco

```
void f( ) {  
    int a = 1;  
    a = a + 1;  
    int b = 1;  
    b = b + a;  
}
```

- O escopo dessas variáveis é do ponto em que são definidas até o final do bloco (ou função) em que aparecem.

Escopo

- Uma variável é local a uma unidade/bloco/subprograma, se for **declarada** nessa unidade/bloco/subprograma
- Uma variável é não local a uma unidade/bloco/subprograma se é **visível** mas não é declarada nessa unidade/bloco/subprograma

Escopo estático e estrutura de blocos

Regra básica:

- Uma declaração D “pertence” a um bloco B se B for o bloco aninhado mais próximo contendo D
- Um nome x declarado em B é válido em todo B, exceto quando for redeclarado em um bloco aninhado B'
 - Neste caso, passa a valer o x redeclarado em B'

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Escopo estático e estrutura de blocos

Declaração	Escopo
<code>int a = 1;</code>	B1 – B3
<code>int b = 1;</code>	B1 – B2
<code>int b = 2;</code>	B2 – B4
<code>int a = 3;</code>	B3
<code>int b = 4;</code>	B4

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Escopo estático e estrutura de blocos

Vai imprimir:
32

Vai imprimir:
14

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Escopo estático

- Em ADA, variáveis escondidas, de escopos mais gerais, podem ser acessadas com referências seletivas:

`P1.x`

- Em C / C++, variáveis estáticas globais podem ser “escondidas” por variáveis locais
 - Operador de escopo pode ser usado para acessar globais ocultas:

`::x`

Escopo dinâmico

- O escopo é definido em função da execução do programa. O efeito de uma declaração se estende até que uma nova declaração com o mesmo nome seja encontrada.

```
procedure big;  
  var x : integer;  
  procedure sub1;  
    begin { sub1 }  
      ... x ...  
    end; { sub1 }  
  procedure sub2;  
    var x : integer;  
    begin { sub2 }  
      ...  
    end; { sub2 }  
  begin { big }  
    ...  
  end; { big }
```


Escopo dinâmico

- As referências a um identificador não podem ser identificadas na compilação
- Para a sequência de chamadas: big - sub2 - sub1
 - a referência a x em sub1 é ao x declarado em sub2
- Para a sequência de chamadas: big - sub1
 - a referência a x em sub1 é ao x declarado em big

Escopo dinâmico

- Exemplo: variáveis especiais em LISP

```
> (defun baz () x)
> (defun foo ()
    (let ((x 2))
      (declare (special x))
      (baz)))
> (foo)
```

- Resposta: 2

Resumo

Resumo

- Vimos conceitos de nomes e vinculações, aplicados às variáveis
 - 6-tupla:
 - <nome, endereço, valor, tipo, tempo de vida, escopo>
- Parte importante da programação imperativa
 - Controle explícito da atribuição
 - Vinculação de valores
- Outras vinculações discutidas
 - Nomes / escopo
 - Endereço / tempo de vida
 - Tipos

Próximas aulas

- Controle de fluxo
 - Programador dizendo “como” o computador deve proceder
- Implementação da chamada de subprogramas
 - Caracteriza as linguagens estruturadas

Fim