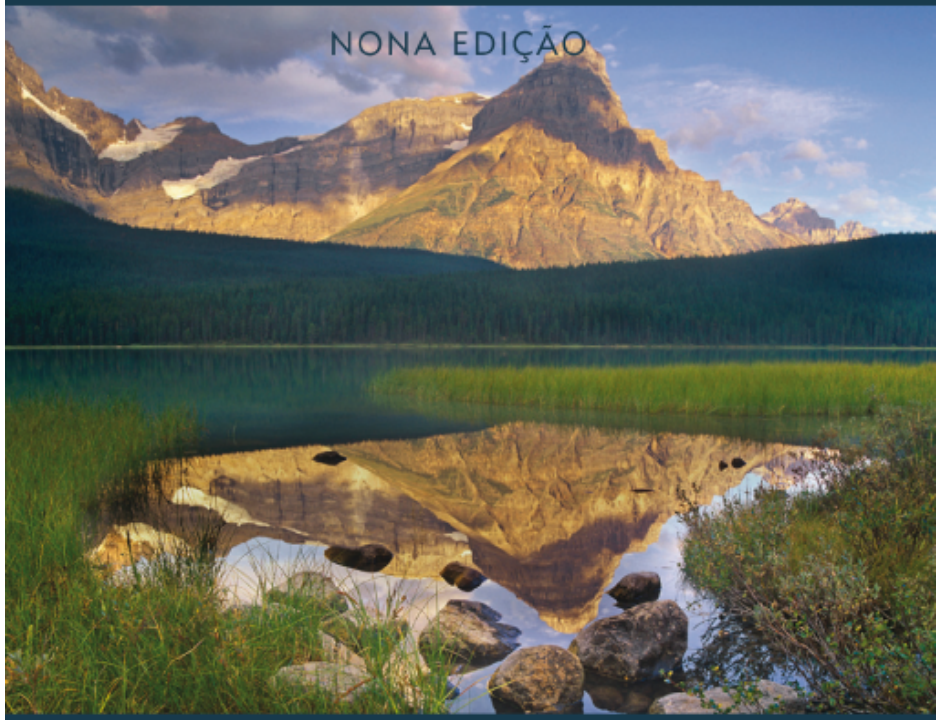
CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

NONA EDIÇÃO

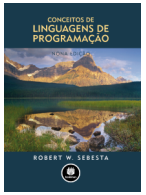


ROBERT W. SEBESTA

bookman

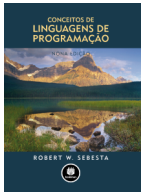
Capítulo 6

Tipos de Dados



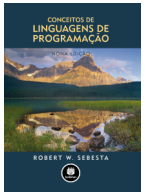
1. Introdução

- Tipo de Dados: define uma **coleção de valores** de dados e um **conjunto de operações** pré-definidas sobre eles
- Um ***descritor*** é a coleção de atributos de uma variável
- Um *objeto* representa uma instância de um tipo de dados definido pelo usuário
- Questão de projeto: Que operações são fornecidas para variáveis do tipo e como elas são especificadas?



2. Tipos de dados primitivos

- Dados Primitivos: são definidos em termos de outros tipos
- Praticamente todas as linguagens de programação fornecem um conjunto de *tipos de dados primitivos*
- Alguns dos tipos primitivos são meramente reflexos de hardware
- Outros requerem apenas um pouco de suporte externo ao hardware para sua implementação



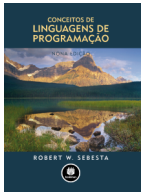
2.1. Inteiro

- Quase sempre um reflexo exato do hardware, logo o mapeamento é simples

- Muitos computadores suportam diversos tipos de tamanhos inteiros

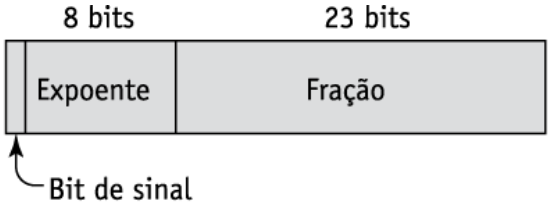
- Java inclui quatro tamanhos inteiros com sinal:

  `byte, short, int, long`

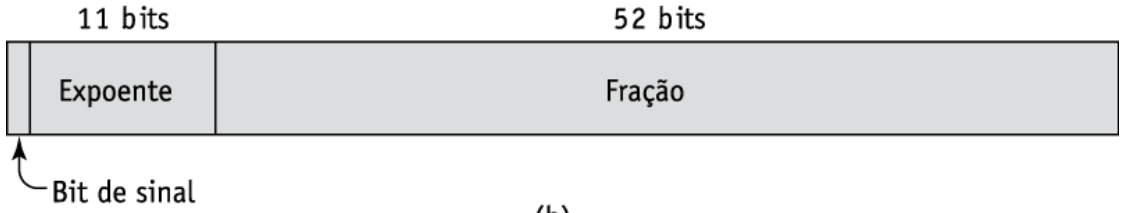


2.2. Ponto flutuante

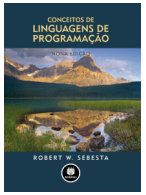
- Modelam números reais, mas as representações são apenas aproximações
- Linguagens para uso científico suportam pelo menos dois tipos de ponto flutuante (por exemplo, `float` e `double`)
- IEEE Padrão de Ponto Flutuante 754:



(a)



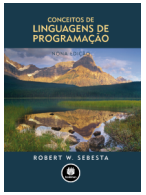
(b)



2.3. Complexo

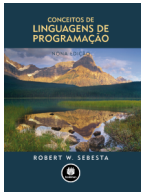
- Algumas linguagens suportam um tipo de dados (Fortran e Python)
- Valores complexos são representados como pares ordenados de valores de ponto flutuante
- Literal complexo (em Python):

  `(7 + 3j)`, onde `7` é a parte real e `3` é a parte imaginária



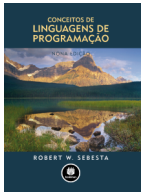
2.4. Decimal

- Para aplicações de sistemas de negócios
  - Essencial para COBOL
  - C# tem um tipo de dados decimal
- Decimais codificados em binário (BCD)
- *Vantagem*: precisão
- *Desvantagens*: faixa de valores restrita, gasto de memória



2.5. Booleanos

- Mais simples de todos
- Faixa de valores: dois elementos, um para “verdadeiro” e um para “falso”
- Poderiam ser representados por bits, mas são armazenados em bytes
- Vantagem: legibilidade



2.6. Caractere

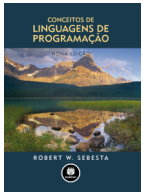
- Armazenados como codificações numéricas

- Codificação mais usada:

ASCII (American Standard Code for Information Interchange)

- Alternativa: conjunto de caracteres de 16 bits: Unicode (UCS-2)
  - Inclui caracteres da maioria das linguagens naturais
  - Originalmente usado em Java
  - C# e JavaScript também suportam Unicode

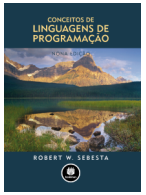
- Unicode 32 bits (UCS-4)
  - Suportada por Fortran, começando com 2003



3. Tipo Cadeia de Caracteres

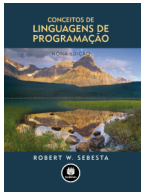
- Valores são sequências de caracteres

- Questões de projeto:
  - As cadeias devem ser apenas um tipo especial de vetor de caracteres ou um tipo primitivo?
  - As cadeias devem ter tamanho estático ou dinâmico?



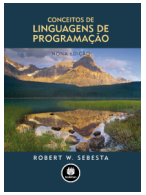
3.1. Cadeias e suas Operações

- Operações comuns:
  - Atribuição
  - Comparação (=, > etc.)
  - Concatenação
  - Referência a subcadeias
  - Casamento de padrões



Cadeias de caracteres em certas linguagens

- C e C++
  - Não são definidas como primitivas
  - Usam matrizes de caracteres **`char`** e uma biblioteca de funções que fornecem operações
- Fortran e Python
  - Tipo primitivo com atribuição e diversas operações
- Java
  - Primitiva via classe `String`
- Perl, JavaScript, Ruby e PHP
  - Incluem operações de casamento de padrões pré-definidas, usando expressões regulares



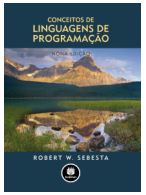
3.2. Opções de tamanho de cadeias

- *Estático*: COBOL, classe pré-definida `String`

- *Dinâmico limitado*: C and C++
  - Nessas linguagens, um caractere especial é usado para indicar o fim da cadeia de caracteres

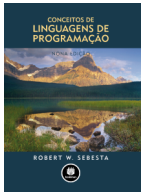
- *Dinâmico*: SNOBOL4, Perl, JavaScript

- Ada 95 suporta as três opções



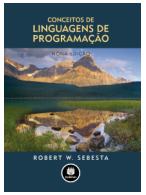
3.3. Avaliação

- Importantes para a facilidade de escrita
- Se tipos primitivos de tamanho estático não tem custo computacional, por que não tê-los em uma linguagem?
- Tamanho dinâmico é interessante, mas vale o custo?

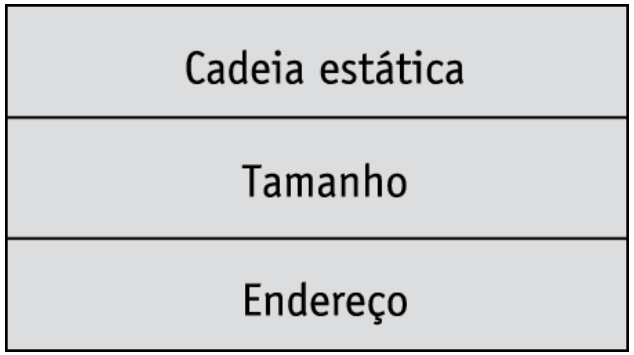


3.4. Implementação de cadeias de caracteres

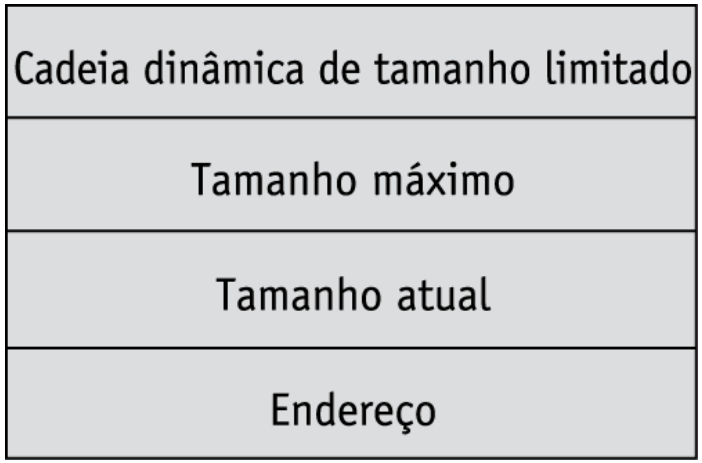
- **Tamanho estático**: descritor em tempo de compilação
- **Tamanho dinâmico limitado**: pode necessitar de um descritor em tempo de execução (mas não em C e C++)
- **Tamanho dinâmico**: necessita de descritor em tempo de execução; alocação/liberação é o maior problema de implementação



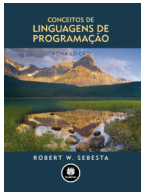
Descritores em tempo de compilação e de execução



Descritor em tempo de compilação para cadeias estáticas



Descritor em tempo de execução para cadeias dinâmicas de tamanho limitado

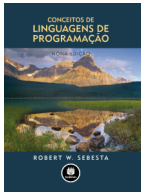


4. Tipos ordinais definidos pelo usuário

- Faixa de valores possíveis pode ser facilmente associada com o conjunto de inteiros positivos

- Exemplos de tipos primitivos ordinais em Java
  - `integer`
  - `char`
  - `Boolean`

- Dois tipos ordinais definidos pelo usuário:
  - `Enumeração`
  - `Subfaixas`



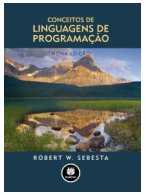
4.1. Tipos Enumeração

- Todos os valores possíveis (constantes nomeadas) são enumerados na definição

- Exemplo em C#

```
enum dias{seg, ter, qua, qui, sex, sab, dom};
```

- Questões de projeto
  - Uma constante de enumeração pode aparecer em mais de uma definição de tipo? Se pode, como o tipo de uma ocorrência de tal constante é verificado no programa?
  - Os valores de enumeração são convertidos para inteiros?
  - Existem outros tipos que são convertidos para um tipo enumeração?



4.1.1. Projetos

- Se a linguagem não tem tipos enumeração, poderíamos simular:

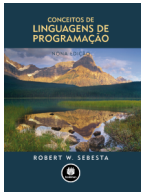
```
int vermelho = 0, azul = 1;
```

- Exemplo em C++

```
enum cores{vermelho, azul, verde, amarelo, preta};
Cores minhaCor = amarelo, suaCor = preta;
minhaCor++; //retorna preta
```

- Ilegal em C++

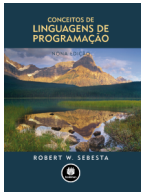
```
suaCor = 4;
```



4.1.2. Avaliação de tipos enumeração

- Melhora a legibilidade: não precisa codificar uma cor como um número

- Melhora a confiabilidade: compilador pode verificar:
  - operações (não permitir que as cores sejam adicionadas)
  - Nenhuma variável de enumeração pode ter um valor atribuído fora do intervalo definido
  - C# e Java 5.0 fornecem melhor suporte para enumeração do que C++, porque variáveis do tipo enumeração nessas linguagens não são convertidas para tipos inteiros

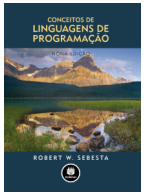


4.2. Tipos subfaixa

- Uma subsequência contígua de um tipo ordinal
  - Exemplo: 12..18 é uma subfaixa do tipo inteiro

- Projeto de Ada

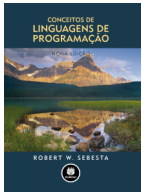
```
type Dias is (seg, ter, qua, qui, sex, sab, dom);
subtype DiaDeSemana is Dias range seg..sex;
subtype Index is Integer range 1..100;

Dia1: Dias;
Dia2: DiaDeSemana;
Dia2 := Dia1;
```



Avaliação do tipo subfaixa

- Melhora a legibilidade
  - Torna claro aos leitores que as variáveis de subtipos podem armazenar apenas certas faixas de valores

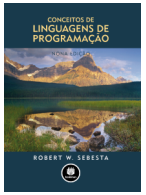
- Melhora a confiabilidade
  - A atribuição de um valor a uma variável de subfaixa que está fora da faixa especificada é detectada como um erro



4.3. Implementação de tipos ordinais definidos pelo usuário

- Tipos enumeração são implementados como inteiros

- Tipos subfaixas são implementados como seus **tipos ancestrais**, exceto que as verificações de faixas devem ser implicitamente incluídas pelo compilador em cada atribuição de uma variável ou de uma expressão a uma variável subfaixa

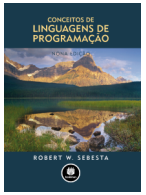


5. Tipos matrizes

- Uma matriz é um agregado **homogêneo** de dados no qual um **elemento** individual é identificado por sua **posição** na agregação, relativamente ao primeiro elemento

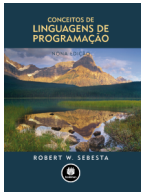
- Em C e Java, todos os elementos devem ser de um mesmo tipo.

- Em JavaScript, Python, Ruby, as variáveis são referências sem tipo para objetos.



5.1. Questões de projeto de matrizes

- Que tipos são permitidos para índices?
- As expressões de índices em referências a elementos são verificadas em relação à faixa?
- Quando as faixas de índices são vinculadas?
- Quando ocorre a liberação da matriz?
- As matrizes multidimensionais irregulares ou retangulares são permitidas?
- As matrizes podem ser inicializadas quando elas têm seu armazenamento alocado?
- Que tipos de fatias são permitidas?

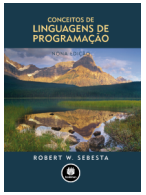


5.2. Matrizes e índices

- *Indexar* (ou subscrever) é um mapeamento de índices para elementos

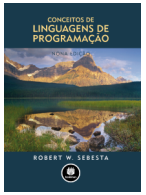
  `nome_matriz (lista_valores_índices) → elemento`

- Sintaxe de índices
  - FORTRAN, PL/I e Ada usam parênteses
    - Ada explicitamente usa parênteses para mostrar a uniformidade entre as referências matriz e chamadas de função, pois ambos são mapeamentos
  - A maioria das outras linguagens usa chaves



Tipos de índices de matrizes

- FORTRAN, C: apenas inteiros
- Ada: inteiro ou enumeração
- Java: apenas tipos inteiros
- Verificação de faixas de índices
  - C, C++, Perl e Fortran não especificam faixas de índices
  - Java, ML e C# especificam faixas de índices
  - Em Ada, o padrão é exigir a verificação de faixas de índice, mas pode ser desligada



5.4. Inicialização de matrizes

- Algumas linguagens fornecem os meios para inicializar matrizes no momento em que seu armazenamento é alocado

  – Exemplo em C, C++, Java, C#

```
int lista[] = {4, 5, 7, 83}
```

  – Cadeias de caracteres em C e C++

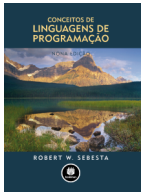
```
char nome[] = "Mafalda";
```

  – Matrizes de cadeias em C e C++

```
char *nomes[] = {"Bob", "Marley", "Joana"];
```

  – Inicialização de objetos String em Java

```
String[] nomes= {"Bob", "Marley", "Joana"};
```



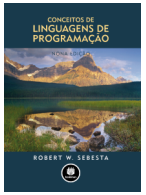
Inicialização de matrizes

- Ada
  - `List : array (1..5) of Integer :=`
    `(1 => 17, 3 => 34, others => 0);`

- Python
  - Compreensões de lista

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

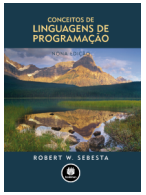
puts `[0, 9, 36, 81]` in `list`



Matrizes heterogêneas

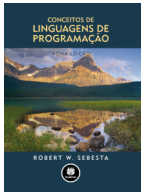
- Uma *matriz heterogênea* é uma em que os elementos não precisam ser do mesmo tipo

- Suportadas por Perl, Python, JavaScript e Ruby



5.5. Operações de matrizes

- **APL** é a linguagem de processamento de matrizes mais poderosa já desenvolvida. As quatro operações aritméticas básicas são definidas para vetores (matrizes de dimensão única) e para matrizes, bem como operadores escalares
- **Ada** permite atribuição de matrizes, mas também concatenação
- **Python** fornece atribuição de matrizes, apesar de ser apenas uma mudança de referência. Python também suporta operações para concatenação de matrizes e para verificar se um elemento pertence à matriz
- **Ruby** também fornece concatenação de matrizes
- **Fortran** inclui operações *elementais* porque elas ocorrem entre pares de elementos de matrizes



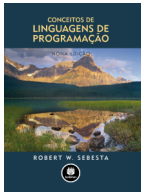
5.6. Matrizes retangulares e irregulares

- Uma matriz **retangular** é uma multidimensional na qual todas as linhas e colunas têm o mesmo número de elementos

- Na matriz **irregular**, o tamanho das linhas não precisa ser o mesmo
  - Possíveis quando as multidimensionais são matrizes de matrizes

- C, C++ e Java suportam matrizes irregulares

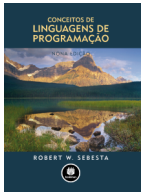
- Fortran, Ada e C# suportam matrizes retangulares (C# também suporta irregulares)



5.7. Fatias

- Uma fatia é alguma subestrutura de uma matriz; nada mais do que um mecanismo de referência

- Fatias são úteis apenas em linguagens que têm operações de matrizes



Exemplos de fatias

- Fortran 95

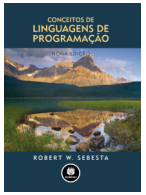
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` é uma matriz de quatro elementos

- Ruby suporta fatias com o método `slice`

```
list = [2, 4, 6, 8, 10];
```

`list.slice(2, 2)` retorna o terceiro e o quarto elementos de `list`

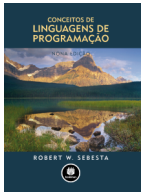


5.8. Implementação de matrizes

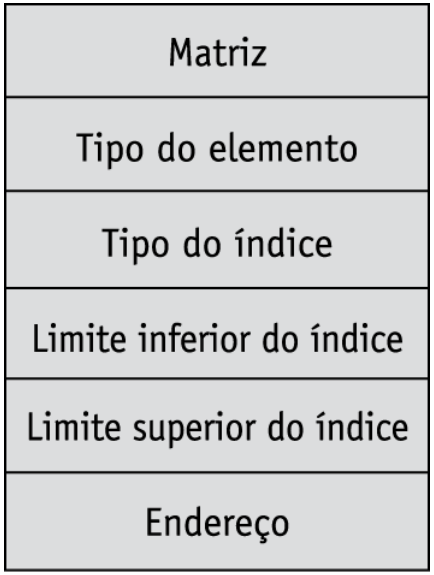
- Função de acesso mapeia expressões subscritas para um endereço na matriz

- Função de acesso para list:

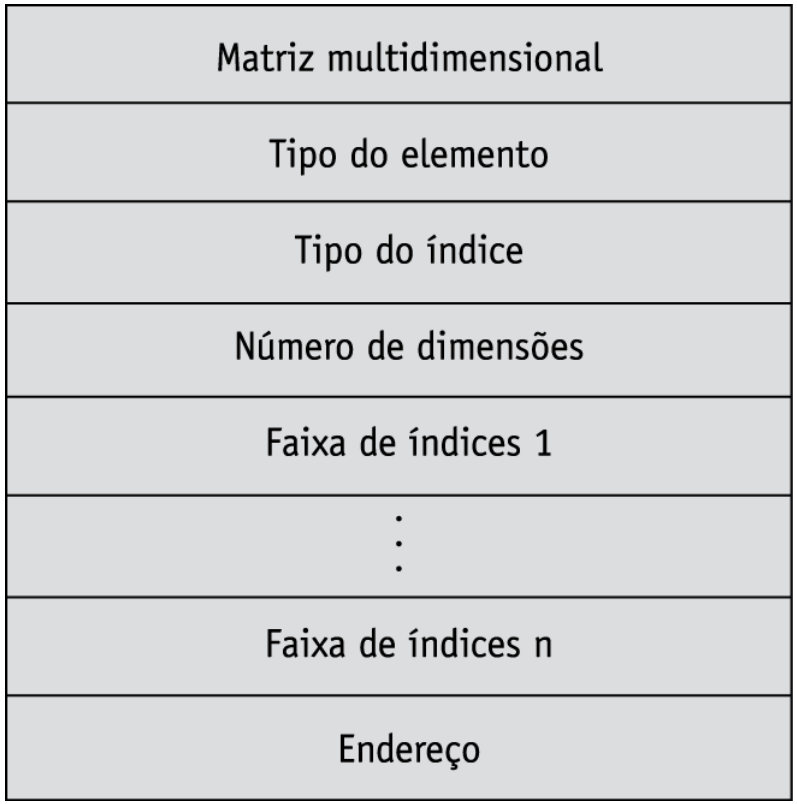
```
endereço(list[k]) =
 endereço (list[0]) + k * tamanho_do_elemento)
```



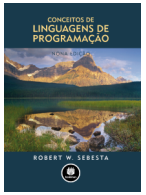
Descritores em tempo de compilação



Matriz de uma dimensão



Matriz multidimensional

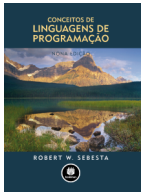


6. Matrizes associativas

- Coleção não ordenada de elementos de dados indexados por um número igual de valores chamados de *chaves*
  - Chaves definidas pelo usuário devem ser armazenadas

- Questões de projeto:
  - Qual é o formato das referências aos seus elementos?
  - O tamanho é estático ou dinâmico?

- Suportadas diretamente em Perl, Python, Ruby e Lua
  - Em Lua, suportadas por tabelas



6.1. Matrizes associativas em Perl

- Nomes começam com `%`; literais são delimitados por parênteses

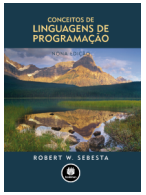
```
%temp = (”seg" => 77, ”ter" => 79, “qua” => 65, …);
```

- Índices são escritos com colchetes e chaves

```
$temp{”qua"} = 83;
```

- Elementos podem ser removidos com delete

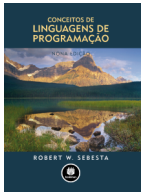
```
delete $temp{”ter"};
```



7. Registros

- Agregado de elementos de dados no qual os elementos individuais são identificados por nomes e acessados por meio de deslocamentos a partir do início da estrutura

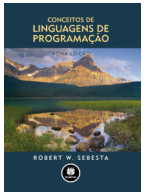
- Questões de projeto:
  - Qual é a forma sintática das referências a campos?
  - Referências elípticas são permitidas?



7.1. Definição de registros em COBOL

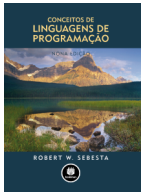
- COBOL usa números de nível para montar uma estrutura hierárquica de registros

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```



7.2. Operações em registros

- **Atribuição** é comum se os tipos são idênticos
- Ada permite **comparações** entre registros
- Registros em Ada podem ser **inicializados** com literais agregados
- COBOL fornece `MOVE CORRESPONDING`
  - Copia um campo do registro de origem especificado para o registro de destino se este tiver um campo com o mesmo nome

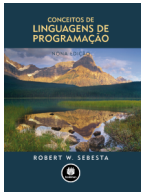


7.3. Avaliação e comparação de registros

- Registros são utilizados quando a coleta de valores de dados é heterogênea

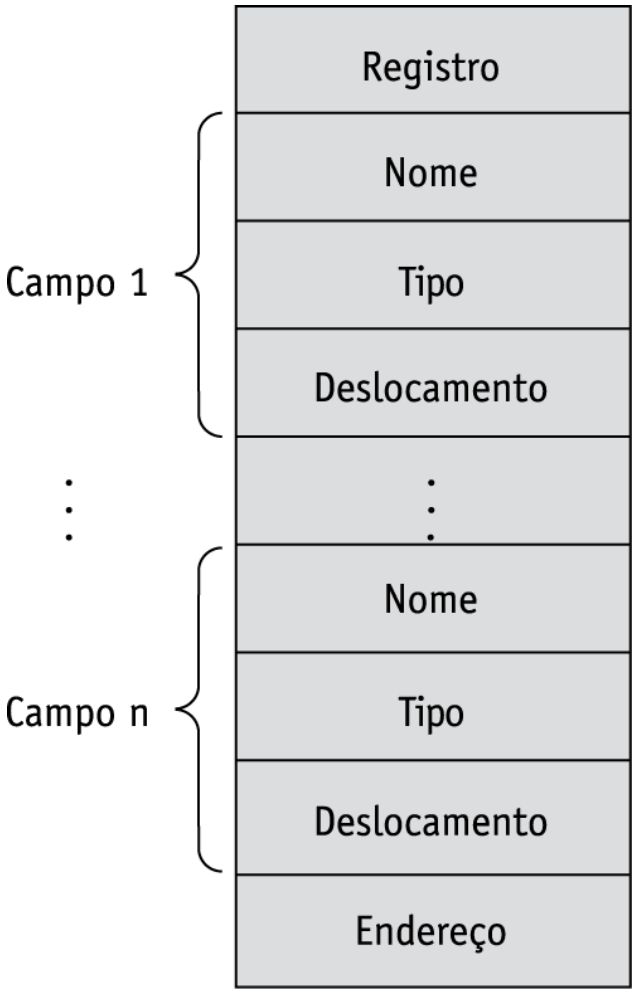
- Acesso a elementos de matriz são muito mais lentos do que campos
de um registro porque os índices são dinâmicos (nomes de campos
são estáticos)

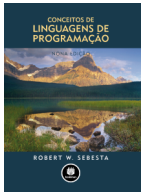
- Índices dinâmicos podem ser usados com acessos a campos de registro, mas isso desabilitaria a verificação de tipos e ficaria mais lento



7.4. Implementação de registros

O endereço de deslocamento relativo ao início do registro é associado com cada campo

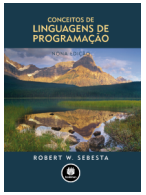




8. Uniões

- Uma *união* é um tipo cujas variáveis podem armazenar diferentes valores de tipos em diferentes momentos durante a execução de um programa

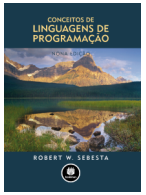
- Questões de projeto
  - A verificação de tipos deve ser obrigatória?
  - As uniões devem ser embutidas em registros?



8.1. Uniões discriminadas x Uniões livres

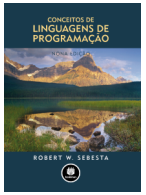
- Fortran, C e C++ fornecem construções para representar uniões nas quais **não existe um suporte** da linguagem para a verificação de tipos; as uniões nessas linguagens são chamadas de **uniões livres**

- A verificação de tipos união requer que cada construção de união inclua um indicador de tipo, chamado de *discriminante*
  - Uniões discriminadas são suportadas por Ada

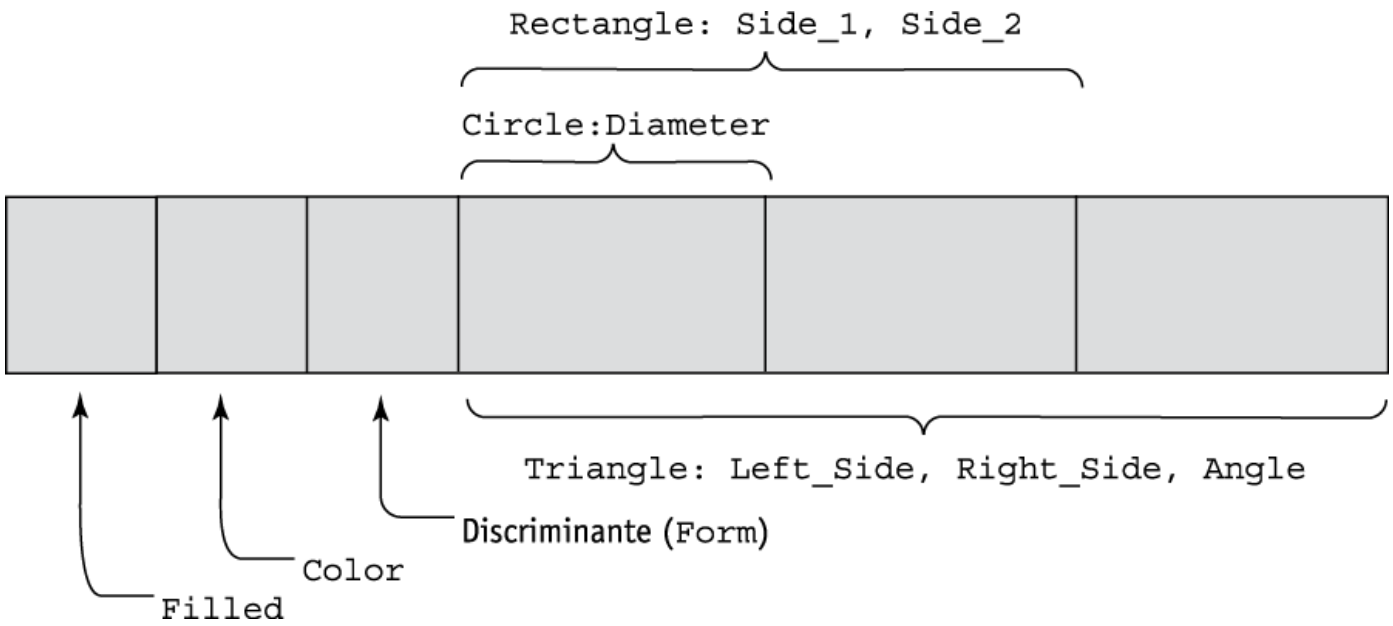


Uniões em Ada

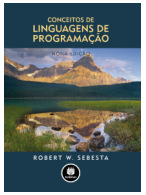
```
type Forma is (Circulo, Triangulo, Retangulo);
type Cores is (Vermelho, Verde, Azul);
type Figura (Form: Forma) is record
 Preenchido: Boolean;
 Cor: Colors;
 case Form is
      when Circulo => Diametro: Float;
      when Triangulo =>
            LadoEsquerdo, LadoDireito: Integer;
            Angulo: Float;
      when Retangulo => Lado1, Lado2: Integer;
 end case;
end record;
```



Uniões em Ada



Uma união discriminada de três variáveis do tipo Forma

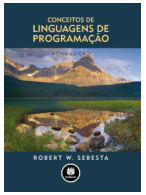


8.2. Avaliação de uniões

- Uniões são construções potencialmente inseguras
  - Não permite verificação de tipos

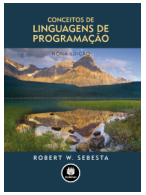
- Java e C# não suportam uniões
  - Reflexo da crescente preocupação com a segurança em linguagens de programação

- Em Ada, podem ser usadas com segurança



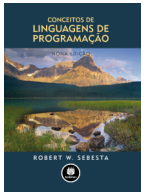
9. Ponteiros e referências

- Tipo no qual as variáveis possuem uma faixa de valores que consistem em endereços de memória e um valor especial, *nil*
- Fornecem alguns dos poderes do endereçamento indireto
- Fornecem uma maneira de gerenciar o armazenamento dinâmico
- Um ponteiro pode ser usado para acessar uma posição na área onde o armazenamento é dinamicamente alocado, o qual é chamado de monte (*heap*)



9.1. Questões de projeto

- Qual é o escopo e o tempo de vida de uma variável do tipo ponteiro?
- Qual e o tempo de vida de uma variável dinâmica do monte?
- Os ponteiros são restritos em relação ao tipo de valores aos quais eles podem apontar?
- Os ponteiros são usados para gerenciamento de armazenamento dinâmico, endereçamento indireto ou ambos?
- A linguagem deveria suportar tipos ponteiro, tipos de referência ou ambos?



9.2. Operações de ponteiros

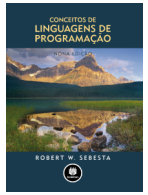
- Duas operações de ponteiros fundamentais: atribuição e desreferenciamento

- Atribuição modifica o valor de uma variável de ponteiro para algum endereço útil.

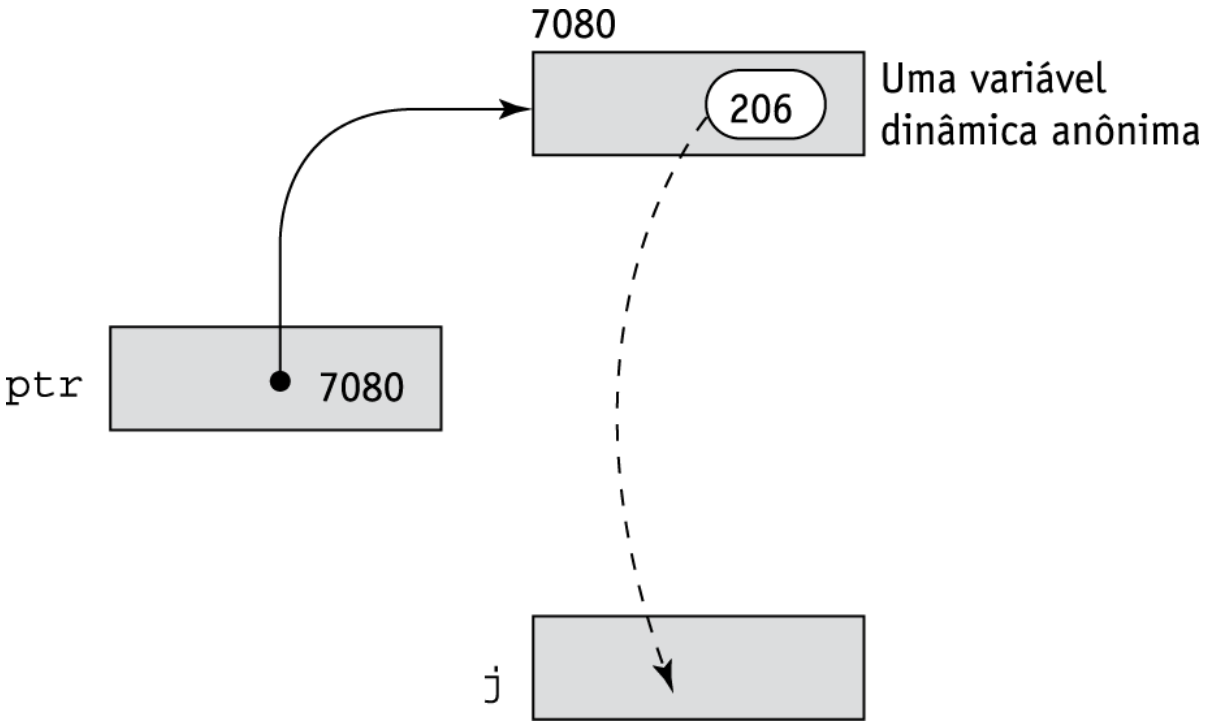
- Desreferenciamento leva uma referência por meio de um nível de indireção
  - Desreferencimento pode ser explícito ou implícito
  - Em C++, é explicitamente especificado com o asterisco (*)

    `j = *ptr`

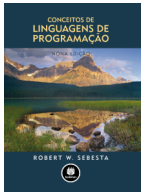
    modifica j para o valor de `ptr`



Atribuição de ponteiro



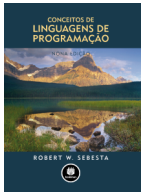
A operação de atribuição j = *ptr



9.3. Problemas com ponteiros

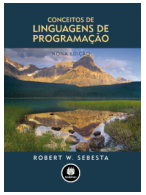
- Ponteiros soltos (perigoso)
  - É um ponteiro que contém o endereço de uma variável dinâmica do monte que já foi liberada

- Variáveis dinâmicas do monte perdidas
  - É uma variável dinâmica alocada do monte que não está mais acessível para os programas de usuário (geralmente chamadas de *lixo*)
    - O ponteiro `p1` é configurado para apontar para uma variável dinâmica do monte recém-criada
    - `p1` posteriormente é configurado para apontar para outra variável dinâmica do monte recém-criada
    - A primeira variável dinâmica do monte é agora inacessível, ou perdida. Isso às vezes é chamado de *vazamento de memória*



9.4. Ponteiros em C e C++

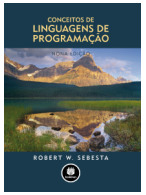
- Extremamente flexíveis, mas devem ser usados com muito cuidado
- Podem apontar para qualquer variável, independentemente de onde ela estiver alocada
- Usado para o gerenciamento de armazenamento dinâmico e endereçamento
- A aritmética de ponteiros é também possível de algumas formas restritas
- C e C++ incluem ponteiros do tipo **void ***, que podem apontar para valores de quaisquer tipos. São, para todos os efeitos, ponteiros genéricos



Aritmética de ponteiros em C e C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` é equivalente a `stuff[5]` e `p[5]`
`*(p+i)` é equivalente a `stuff[i]` e `p[i]`

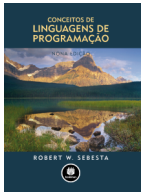


9.5. Tipos de referência

- Uma variável de *tipo de referência* é similar a um ponteiro, com uma diferença importante e fundamental: um ponteiro se refere a um endereço em memória, enquanto uma referência se refere a um objeto ou a um valor em memória

- Em Java, variáveis de referência são estendidas da forma de C++ para uma que as permitem substituírem os ponteiros inteiramente

- C# inclui tanto referências de Java quanto ponteiros de C++

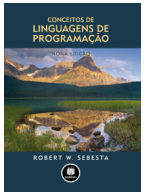


9.6. Avaliação

- Ponteiros soltos e lixo são problemas, tanto quanto o gerenciamento do monte

- Ponteiros são como a instrução `goto` - que aumenta a faixa de células que podem ser acessadas por uma variável

- Ponteiros e referências são necessários para estruturas de dados dinâmicas – não podemos projetar uma linguagem sem eles

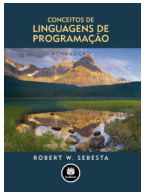


10. Verificação de tipos

- Conceito de **operandos** e **operadores** é generalizado para incluir subprogramas e sentenças de atribuição

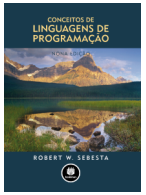
- *Verificação de tipos* é a atividade de **garantir** que os operandos de um operador são de tipos **compatíveis**

- Um *tipo compatível* é um que ou é legal para o operador ou é permitido a ele, dentro das regras da linguagem, ser implicitamente convertido pelo código gerado pelo compilador (ou pelo interpretador) para um tipo legal
  - Essa conversão automática é chamada de *coerção*



Verificação de tipos (continuação)

- Um *erro de tipo* é a aplicação de um operador a um operando de um tipo não apropriado
- Se todas as vinculações são estáticas, a verificação de tipos pode ser feita praticamente sempre de maneira estática
- Se as vinculações de tipo são dinâmicas, a verificação de tipos deve ser dinâmica
- Uma linguagem de programação é **fortemente tipada** se os erros de tipo são **sempre detectados**
- Vantagem de tipagem forte: permite a detecção da utilização indevida de variáveis que resultam em erros de tipo



11. Tipagem forte

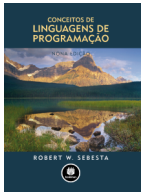
Exemplos de linguagens:

- FORTRAN 95 não é fortemente tipada: parâmetros, `EQUIVALENCE`

- C e C++ também não: ambas incluem tipos união, que não são verificados em relação a tipos

- Ada é quase fortemente tipada

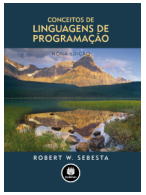
  (Java e C# são similares a Ada)



12. Equivalência de tipos por nome

- *Equivalência de nomes por tipo* significa que duas variáveis são equivalentes se elas são definidas na mesma declaração ou em declarações que usam o mesmo nome de tipo

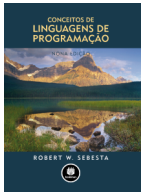
- Fácil de implementar, mas é mais restritiva:
  - Subfaixas de tipos inteiros não são equivalentes a tipos inteiros
  - Parâmetros formais devem ser do mesmo tipo que os seus correspondentes parâmetros reais



Equivalência de tipos por estrutura

- *Equivalência de tipos por estrutura* significa que duas variáveis têm tipos equivalentes se seus tipos têm estruturas idênticas

- Mais flexível, mas mais difícil de implementar



13. Teoria e tipos de dados

- A teoria de tipos é uma ampla área de estudo em matemática, lógica, ciência da computação e filosofia
- Um sistema de tipos é um conjunto de tipos e as regras que governam seu uso em programas
  - Tanto uma gramática de atributos quanto um mapa de tipos pode ser usado para as funções
  - Mapeamento finito – modela matrizes e funções
  - Produto cartesiano – modela tuplas e registros
  - União de conjunto – modela tipos de dados de união
  - Subconjuntos – modela subtipos