Paradigmas de Linguagens de Programação

Tipos de Dados

Introdução

- Um **tipo de dado** define uma coleção de valores de dados e um conjunto de operações predefinidas sobre eles.
- **Importância**:
  - Determinam como dados são manipulados pelos programas.
  - Aproximam a programação do mundo real do problema tratado.
- **Evolução histórica**:
  - Primeiras linguagens: poucos tipos de dados suportados.
  - Fortran pré-90: listas ligadas e árvores binárias implementadas via vetores.
  - COBOL: permitiu precisão em dados decimais e tipos estruturados para registros.
  - PL/I: ampliou tipos de dados e precisão para inteiros e ponto flutuante.
  - ALGOL 68: inovou ao oferecer tipos básicos + operadores para criação de estruturas flexíveis.

Introdução

- **Tipos definidos pelo usuário**: permitem abstrações, legibilidade e verificação automática.
- **Tipo abstrato de dados** (anos 1980): separa interface (visível ao usuário) da implementação (oculta).
- **Uso prático dos tipos de dados**:
  - Estruturas (vetores, matrizes, registros).
  - Funções de linguagens funcionais (ex.: listas em Lisp).
  - Tipos complexos em linguagens modernas (C++, Python, C#).
- **Objetivo**: fornecer suporte a estruturas que:
  - Simplifiquem a modelagem de problemas.
  - Garantam consistência e segurança na manipulação de dados.
  - Aumentem a legibilidade e manutenção do código.

Introdução

- **Descritor**
  - Conjunto de atributos de uma variável.
  - Exemplos de atributos: nome, tipo, endereço, escopo, tempo de vida, valor.
  - Tipos de descritor:
    - **Estático (compilação):** construído pelo compilador, armazenado na tabela de símbolos.
    - **Dinâmico (execução):** mantido pelo sistema em tempo de execução.
- **Objeto**
  - Representa uma instância de um tipo de dado.
  - Inclui tipos primitivos e também tipos abstratos definidos pelo usuário.
  - É o “valor concreto” associado ao descritor e ao tipo.

Tipos de Dados Primitivos

- Definição: tipos não definidos em termos de outros tipos.
- Todas as linguagens oferecem um conjunto de tipos primitivos.
- Podem ser:
- Reflexos diretos do hardware (ex.: inteiros).
- Implementados com pouco suporte externo ao hardware (ex.: ponto flutuante).
- São usados junto com construtores de tipo para especificar tipos estruturados.

Tipos de Dados Primitivos: Inteiro

- Tipo numérico primitivo mais comum.
- Hardware suporta diferentes tamanhos: linguagens refletem isso.
  - Java: byte, short, int, long (com sinal).
  - C/C++/C#: também oferecem versões sem sinal.
- Inteiros sem sinal: usados p/ dados binários.
- Python: inteiros podem ter tamanho ilimitado.
- Representação usual: **complemento de dois**.
  - Ex.: -3 → representado por inversão de bits + 1.

Tipos de Dados Primitivos: Inteiro

- Tabela com os tipos primitivos inteiros de Java

Tipo	**Tamanho (bits)**	**Intervalo (Início)**	**Intervalo (Fim)**
byte	8	-128	127
short	16	-32.768	32.767
int	32	-2.147.483.648	2.147.483.647
long	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Tipos de Dados Primitivos: Ponto Flutuante

- Modelam números reais como aproximações.
- Usados em aplicações científicas: geralmente suportam pelo menos float e double.
- Executam rapidamente e representam uma faixa muito ampla de valores (de números muito pequenos a muito grandes).
- Representação binária pode gerar erros de precisão (ex.: 0.1).
  - Exemplo: (0.1 + 0.2) != 0.3 em muitas linguagens.
- Erros de arredondamento se acumulam em cálculos extensivos.
- Para cálculos financeiros: usar **decimal/numeric** ou armazenar valores em inteiros (centavos).

Tipos de Dados Primitivos: Ponto Flutuante

- Padrão IEEE 754:
  - Precisão simples: 32 bits (1 sinal, 8 expoente, 23 fração).
  - Precisão dupla: 64 bits (1 sinal, 11 expoente, 52 fração).

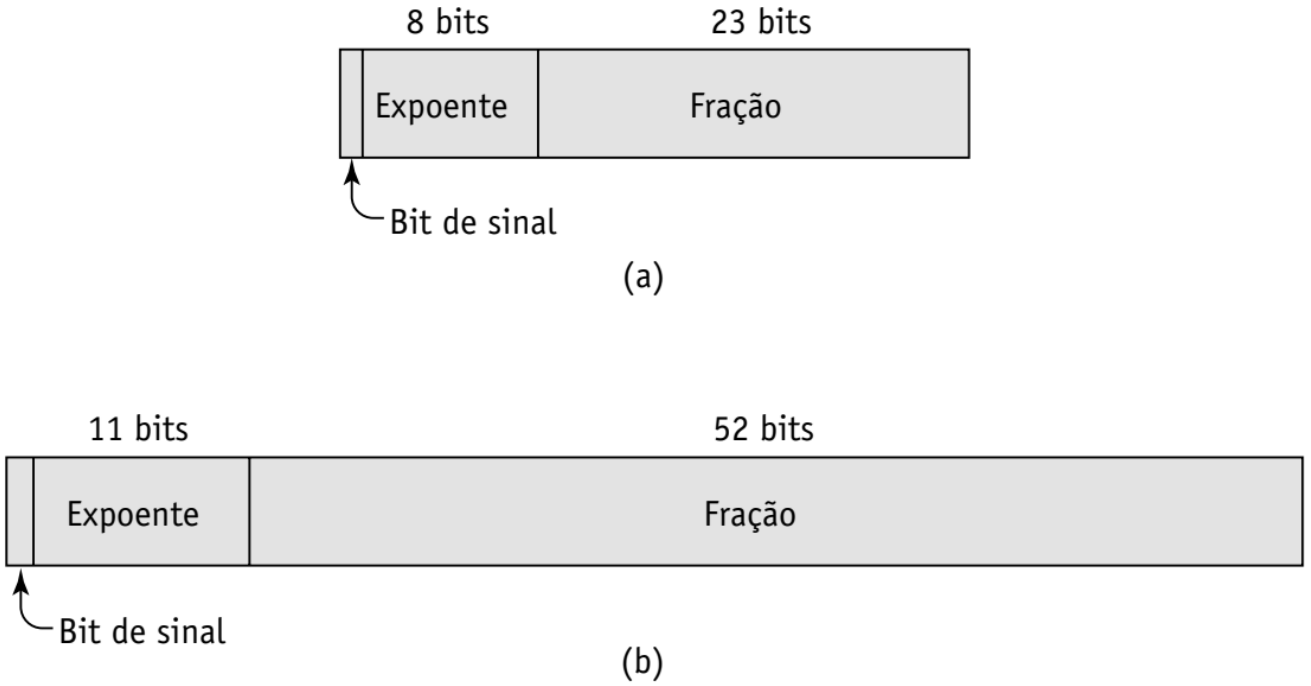


FIGURA 6.1
Formatos de ponto flutuante da IEEE: (a) precisão simples, (b) precisão dupla.

Tipos de Dados Primitivos: Complexo

- Suportado em linguagens como Python, Fortran e C99.
- Representados como pares ordenados de números reais (parte real + parte imaginária).
- Em Python: sufixo j para parte imaginária.
  - (7 + 3j),
  - onde7 é a parte real e 3 é a parte imaginária
- Linguagens que suportam tipo complexo permitem operações aritméticas diretamente com valores complexos.

Tipos de Dados Primitivos: Decimal

- Uso: aplicações de negócios e sistemas financeiros (ex.: COBOL, C#, F#).
- Definição: armazenam um número fixo de dígitos decimais, ponto decimal em posição fixa.
- Representação:
  - **BCD (Binary Coded Decimal)** – cada dígito decimal representado em binário.
  - Exigem mais memória do que representações binárias.
- Vantagens:
  - Representação **precisa** de valores decimais (ex.: 0,1 pode ser exato).
- Desvantagens:
  - **Faixa restrita** de valores.
  - Ocupam mais memória e são menos eficientes.

Tipos de Dados Primitivos: Booleano

- Mais simples de todos.
- Valores possíveis:
  - verdadeiro (true)
  - falso (false)
- Introduzidos no ALGOL 60.
- Em C até C99, booleanos eram simulados com inteiros
  - (0 = falso, ≠0 = verdadeiro).
- Representação típica:
  - **1 bit** (teoricamente)
  - Mas geralmente armazenados como **1 byte** por eficiência.
- Vantagem: maior **legibilidade** do código em comparações e condições.

Tipos de Dados Primitivos: Caractere

- Representação: armazenados como **códigos numéricos**.
- Codificações comuns:
  - **ASCII (8 bits):** até 256 caracteres.
  - **Unicode (16 bits ou mais):** inclui caracteres de diversas línguas.
  - **UTF-32 / UTF-8:** padrões atuais de codificação.
- Exemplo histórico: Java foi a primeira linguagem amplamente usada a empregar Unicode.
- Hoje:
  - C#, JavaScript, Python, Perl, etc. suportam Unicode.
  - Python: não possui tipo char; cada caractere é uma string de tamanho 1
- Importância: essencial para sistemas globais e aplicativos que lidam com múltiplos idiomas.

Tipos de Dados Primitivos: Caractere

ASCII
American Standard Code for Information Interchange
(Puro: 7 bits, 0–127)

Código	**Caractere**
65	A
66	B
67	C
97	a
98	b
99	c

Unicode
(padrão universal; codificações UTF-8/16/32)

Código	**Caractere**
U+0041	A
U+00E9	é
U+03B1	α
U+0416	Ж
U+4E2D	中
U+1F449	👉

Cadeias de Caracteres

- Valores são **sequências de caracteres**.
- Questões de projeto:
  - As cadeias devem ser um **tipo especial de vetor de caracteres** ou um **tipo primitivo**?
  - O tamanho das cadeias deve ser **estático** ou **dinâmico**?
- Em algumas linguagens são **tipos primitivos**: Fortran, Python (str), Java (via String).
- Em outras, são implementadas como **vetores de char** com funções auxiliares: C e C++.

Cadeias de Caracteres

- Operações típicas:
  - Atribuição e cópia.
  - Concatenação.
  - Comparação (>, <, =).
  - Referência a substrings.
  - Casamento de padrões (pattern matching).
- Em muitas linguagens, **bibliotecas** fornecem essas operações.
- Python e Java possuem suporte direto por meio de objetos (str, String).

Cadeias de Caracteres

- C e C++: não são primitivas; implementadas como **vetores de char** + biblioteca (strcpy, strlen, etc.).
- Fortran e Python: tratam cadeias como **tipos primitivos**, com operações diretas de atribuição e comparação.
- Java: tipo primitivo via classe **String**.
- SNOBOL4: linguagem clássica de manipulação de cadeias, com operações avançadas de casamento de padrões.
- Perl, Ruby, PHP, JavaScript: oferecem **regex nativas** para casamento de padrões.

Cadeias de Caracteres

- Expressões Regulares (Regex)
- Permitem casamento de padrões dentro de cadeias.
- Exemplo:

/[A-Za-z][A-Za-z\d]+/

- Casa identificadores que começam com letra e podem ter letras ou dígitos.
- Exemplos que casam: Java, Python3
- Exemplos que não casam: 1abc, _nome

Opções de amanho de cadeias

- **Cadeia de tamanho estático**
  - Comprimento definido na criação.
  - Python (strings imutáveis), Java String, C++ std::string, COBOL.
- **Cadeia de tamanho dinâmico limitado**
  - Tamanho variável até um máximo fixo.
  - C (char[]), estilo C em C++.
  - Uso de '\0' como terminador em vez de armazenar o tamanho.
- **Cadeia de tamanho dinâmico**
  - Tamanho pode variar sem limite pré-definido.
  - JavaScript, Perl, SNOBOL4, biblioteca padrão de C++.
  - Mais flexível, mas exige alocação/liberação dinâmica.
- Observação extra:
  - Algumas linguagens, como Ada, suportam as três opções.

Cadeias de Caracteres

- Strings são fundamentais para a escrita de programas.
- Trabalhar só com vetores de caracteres é trabalhoso (ex.: strcpy em C).
- Strings como tipo primitivo com tamanho estático:
  - Baratas de implementar.
  - Tornam a linguagem mais prática, difícil justificar sua ausência em linguagens modernas.
- Cadeias dinâmicas:
  - Mais flexíveis, mas trazem sobrecarga de custo e complexidade.

Implementação de cadeias de caracteres

- **Cadeia estática**:
  - Atributos conhecidos em tempo de compilação.
  - Descritor contém: **nome, tamanho e endereço**.
  - objetos String imutáveis em Java, Python, C# (classe String).
- **Cadeia de tamanho dinâmico limitado**:
  - Varia até um máximo declarado.
  - Descritor em execução contém: **tamanho máximo, tamanho atual e endereço**.
  - estilo C (char[] com \0 terminador).
- **Cadeia de tamanho dinâmico**:
  - Sem limite fixo: exige alocação/liberação dinâmica.
  - Máxima flexibilidade, mas maior custo de gerenciamento.

Implementação de cadeias de caracteres

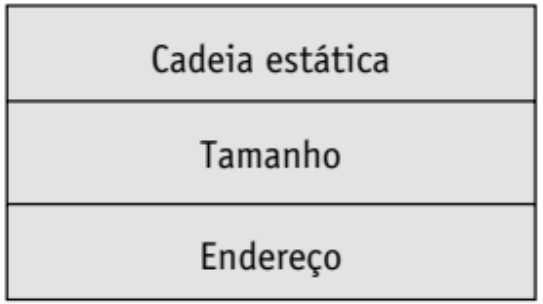


FIGURA 6.2
Descritor em tempo de compilação para cadeias estáticas.

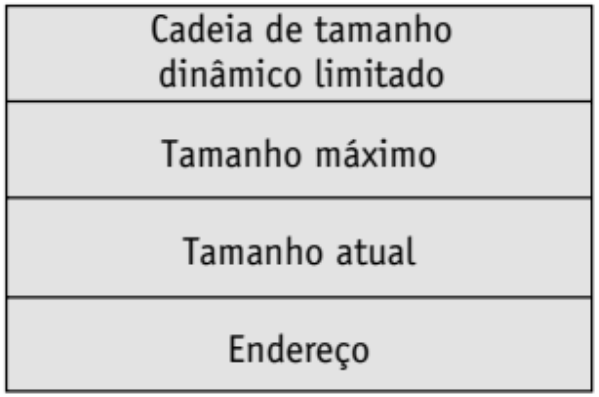


FIGURA 6.3
Descritor em tempo de execução para cadeias de tamanho dinâmico limitado.

Implementação de cadeias de caracteres

Cadeia de tamanho dinâmico

- Abordagens para alocação/liberação:
- **Lista encadeada**: novas células alocadas conforme a cadeia cresce.
  - Desvantagem: mais memória + custo extra de ponteiros.
- **Vetor de ponteiros para caracteres**: todos no heap.
  - Mais rápido que listas encadeadas, mas ainda usa memória extra.
- **Blocos contíguos**: cadeia realocada em novos blocos quando cresce.
  - Mais eficiente em acesso, mas exige cópia dos dados ao expandir.
- Problema central: gerenciar **segmentos variáveis** de memória.

Tipos Enumeração

- “Um **tipo enumeração** é aquele no qual todos os valores possíveis, os quais são **constantes nomeadas**, são fornecidos, ou enumerados, na definição.”
- Permite enumerar valores através de constantes simbólicas
- Pascal

  type cor = (vermelho, azul, branco, preto);
- C, C++

  enum cor { vermelho, azul, branco, preto };
- C#, Java >= 5.0 (Implementado como classe)

  enum cor { vermelho, azul, branco, preto; };

Tipos Enumeração

Questões de Projeto

- Pode uma constante simbólica pertencer a mais de uma definição de tipo? Se sim, como verificar?
- As enumerações podem ser convertidas em inteiros?
- Algum outro tipo pode ser convertido para uma enumeração?

```
enum Cores { Vermelho, Verde, Azul };
enum Semaforo { Vermelho, Amarelo, Verde };

enum Dia {Segunda, Terca, Quarta};
int x = Terca;   // permitido, vira 1

enum Dia {Segunda=1, Terca=2, Quarta=3};
Dia d = (Dia) 42;   // o que acontece
```

Tipos Enumeração

- Legibilidade: substituem números mágicos por nomes significativos.
- Confiabilidade: compilador restringe operações inválidas.
  - impedir atribuição de valores fora da faixa.

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

Tipos de Matrizes

- Uma **matriz** é um agregado homogêneo de elementos de dados.
- Cada elemento é identificado pela **posição (índice)** relativa ao primeiro elemento.
- Todos os elementos de uma matriz são do **mesmo tipo**.
- **Referências aos elementos**: feitas por expressões de índice.
- Se o índice for variável, o endereço precisa ser calculado em **tempo de execução**.
- **Exemplo**:

```
int A[5] = {1, 2, 3, 4, 5}; // C/C++
int[] B = {10, 20, 30}; // Java/C#
```

Tipos de Matrizes

Questões de Projeto

- Quais tipos são permitidos como índices?
- As expressões de índice são verificadas quanto à faixa válida?
- Quando as faixas de índices são vinculadas?
- Quando ocorre a alocação da matriz?
- Matrizes multidimensionais: regulares ou irregulares são permitidas?
- Matrizes podem ser inicializadas junto com a alocação?
- Que tipos de fatias (slices) são permitidas?

Tipos de Matrizes

- Uma **matriz** é acessada por meio de índices (*subscritos*).
- Estrutura geral:

  array_name(subscript_value_list) → element
- Se todos os índices forem **constantes:** seleção **estática**.
- Se algum índice for **variável:** seleção **dinâmica**.
- Matrizes podem ser vistas como **mapeamentos finitos**: associam um conjunto de índices a um elemento de dados.

Tipos de Matrizes

- FORTRAN, PL/I, Ada: usam parênteses ().
- C, Java, Python, etc.: usam colchetes [].
- Ada adota parênteses também para chamadas de funções: uniformidade, mas pode gerar afetar a legibilidade.
- Exemplo em Ada:

```
Sum := Sum + B(I);
```

Tipos de Matrizes

- Fortran pré-90 e PL/I: escolheram parênteses por limitações de perfuração em cartões, que não tinham símbolo de colchetes.
- Fortran I (IBM 704): limitou o número de índices e dimensões para acelerar acesso (via registradores de índices).
- Fortran IV (IBM 7094): permitiu matrizes com até sete índices, mas mantinha foco em desempenho.
- A maioria das linguagens modernas não limita explicitamente o número de índices.

Tipos de Matrizes

- A vinculação do tipo do índice a uma variável matriz é normalmente **estática**, mas os valores do índice podem variar **dinamicamente**.
- Em algumas linguagens, o limite inferior do índice é fixo (ex: C → 0).
- Outras permitem que limites inferiores sejam especificados pelo programador.
- Categorias:
  - Matriz Estática
  - Matrizes Dinâmicas de Pilha (stack) Fixa
  - Matrizes Dinâmicas do Monte (heap) Fixa
  - Matrizes Dinâmicas do Monte (heap)

Tipos de Matrizes

Matriz Estática

- Faixas de índices e alocação são fixas em tempo de compilação.
- Vantagem: eficiência (sem alocação dinâmica).
- Desvantagem: ocupa memória fixa durante toda a execução.

Tipos de Matrizes

Matrizes Dinâmicas de Pilha Fixa

- Faixas de índices são estáticas, mas a alocação é feita em tempo de elaboração da declaração, durante a execução.
- Exemplo: em C, uma matriz declarada dentro de uma função local pode ter alocação em tempo de execução.
- Uma matriz grande em um subprograma pode usar o mesmo espaço que uma matriz grande em um subprograma diferente, desde que ambos os subprogramas não estejam ativos ao mesmo tempo.
- Vantagens: uso mais eficiente de espaço, pois memória é liberada ao final do bloco.
- Desvantagens: tempo adicional de alocação/liberação.

Tipos de Matrizes

Matrizes Dinâmicas do Monte Fixa

- Faixas de índices são estáticas, mas a alocação de armazenamento é feita no **heap** em tempo de execução.
- Uma vez alocada, o tamanho não pode ser alterado.
- Exemplo: uso de malloc/free em C, ou new/delete em C++.
- Também disponível em linguagens como Java e C# (objetos de coleção).

Tipos de Matrizes

Matrizes Dinâmicas do Monte

- Faixas de índices e alocação são dinâmicas.
- Podem crescer ou encolher em tempo de execução.
- Vantagem: flexibilidade.
- Desvantagem: maior custo de gerenciamento.
- Exemplo: ArrayList em Java, list em Python, vetores em JavaScript com push e pop.

Tipos de Matrizes

- C e C++
  - Matrizes com static: estáticas.
  - Matrizes locais: dinâmicas de pilha fixa.
  - malloc/free: dinâmicas do monte fixo.
- C#
  - Fornece arrays fixos e também ArrayList (dinâmico do monte).
- Java
  - Arrays normais: dinâmicos do monte fixo.
  - Coleções como ArrayList: dinâmicos do monte.
- Perl, Python, Ruby, JavaScript
  - Arrays/listas implementados como dinâmicos do monte.

Tipos de Matrizes

- Algumas linguagens permitem inicializar matrizes no momento da alocação (C, C++, Java, C#).
- Exemplo em C:

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

- O compilador define o tamanho com base nos valores.
- Conveniente, mas elimina verificações de consistência (ex.: esquecer um valor pode passar despercebido).

Tipos de Matrizes

- Em C/C++, cadeias de caracteres podem ser inicializadas como matrizes de char:

```
char name[] = "freddie";
```

- Contém 8 elementos: "freddie" + caractere nulo (\0).
- Matrizes de cadeias (array de strings):

```
char *names[] = {"Bob", "Jake", "Darcie"};
```

- names é um array de ponteiros para char.
- Cada elemento aponta para o primeiro caractere de uma string literal.

Tipos de Matrizes

- Matrizes de String em Java:

```
String[] names = {"Bob", "Jake", "Darcie"};
```

- Diferente de C: aqui String é uma classe.
- Cada elemento é uma referência para um objeto String.

Tipos de Matrizes

Operações comuns:

- Atribuição (=)
- Concatenação (+ em Python, Ruby, etc.)
- Comparação (==, !=)
- Fatiamento

- As linguagens baseadas em C não fornecem operações de matrizes, exceto por meio de métodos em Java, C++ e C#.
- Perl oferece suporte à atribuição de matrizes, mas não para comparações.
- Python e Ruby:
  - Matrizes são listas dinâmicas (heterogêneas).
  - Atribuição muda apenas a referência.
  - Suportam concatenação ([1,2]+[3,4]) e operadores como *in* (pertence).
- Ada: suporta atribuição e concatenação de arrays.

Tipos de Matrizes

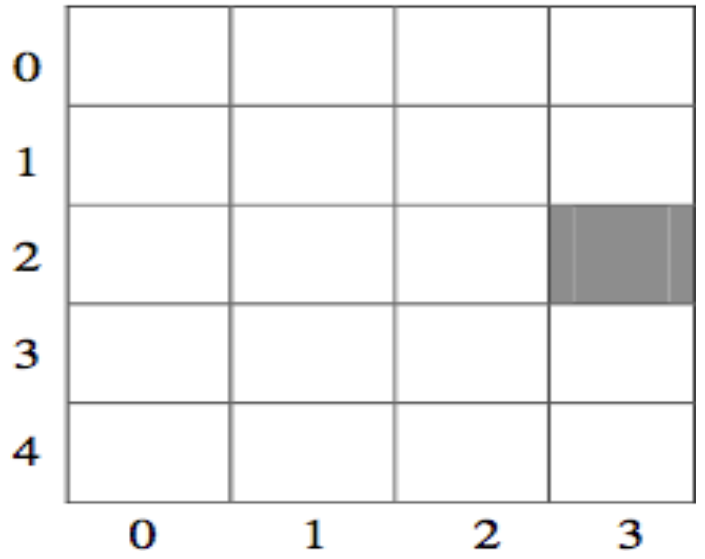
- Fortran:
  - Suporta operações elementares (por pares de elementos).
  - Ex.: A + B soma elemento a elemento.
- F#:
  - Várias operações em Array (append, copy, length).
- APL:
  - Linguagem com foco em vetores/matrizes.

ϕV	inverte os elementos de V
ϕM	inverte as colunas de M
θM	inverte as linhas de M
⌽M	transpõe M (suas linhas viram suas colunas e vice-versa)
⌹M	`inverte M`

Tipos de Matrizes

- Matriz retangular
  - Todas as linhas têm o mesmo número de elementos.
  - Modelam tabelas retangulares (ex.: 3x4, 5x10).
  - Suportadas por: Fortran, Ada, C#, F#.
  - Ex: C#

```
int[,] retangular = new int[3,4];
retangular[0,0] = 10;   // canto superior esquerdo
retangular[2,3] = 20;   // última linha, última coluna
```



Tipos de Matrizes

- Matriz irregular (jagged)
  - Linhas podem ter comprimentos diferentes.
  - Possível quando arrays multidimensionais são implementados como arrays de arrays.
  - Suportadas por: C, C++, Java
  - Ex: Java

```
int[][] irregular = new int[3][];
irregular[0] = new int[2];  // linha com 2 colunas
irregular[1] = new int[5];  // linha com 5 colunas
irregular[2] = new int[3];  // linha com 3 colunas
```

Tipos de Matrizes

Matriz heterogênea

- Os elementos não precisam ser do mesmo tipo.
- Linguagens dinâmicas como Perl, Python, JavaScript e Ruby permitem esse recurso.
- Exemplo em Python:

```
lista = [42, "texto", 3.14, True]
print(lista)
# Saída: [42, 'texto', 3.14, True]
```

Tipos de Matrizes

- Uma **fatia** é uma subestrutura de uma matriz.
- Exemplo: em uma matriz A, a primeira linha ou a primeira coluna podem ser consideradas fatias.
- Não é um novo tipo de dado, mas um mecanismo de referência a parte da matriz.
- Só faz sentido em linguagens que permitem manipular matrizes como unidade.
- Usado para legibilidade e eficiência ao trabalhar com subconjuntos.

Tipos de Matrizes

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
print(vector[3:6])   # [8, 10, 12]

mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(mat[0][:2])    # [1, 2] -> fatia da primeira linha
print(mat[1])        # [4, 5, 6] -> fatia de uma linha inteira
```

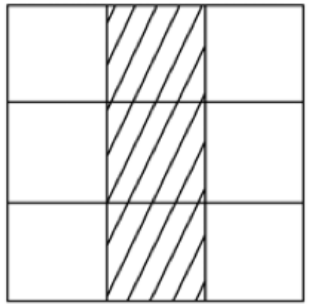
- Python: suporta fatias com **:** (slice notation).
- Perl: suporta fatias via subíndices ou intervalos de índices.
- Ruby: método slice (intervalos ou pares de índices).

Tipos de Matrizes

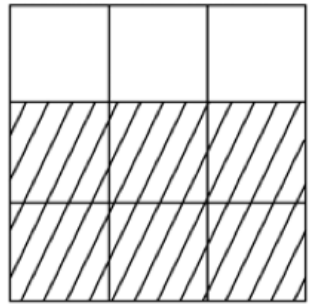
- Fortran oferece suporte direto a fatias multidimensionais
- Similar em Matlab e NumPy (Python).

Figure 6.4

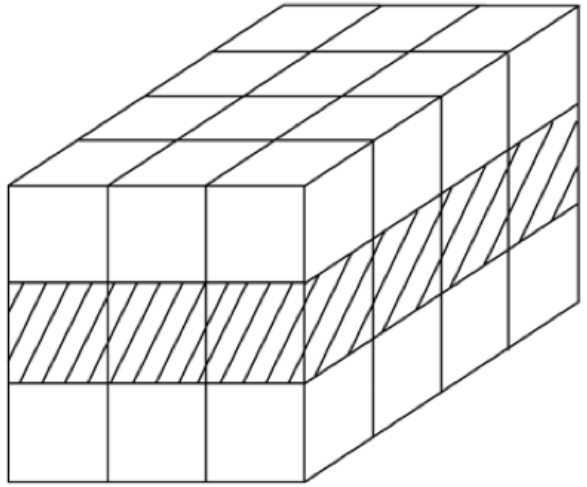
Example slices in Fortran 95



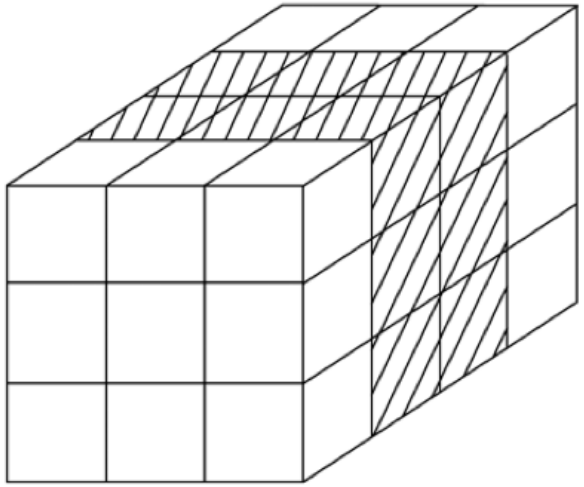
Mat (:, 2)



Mat (2:3, :)



Cube (2, :, :)



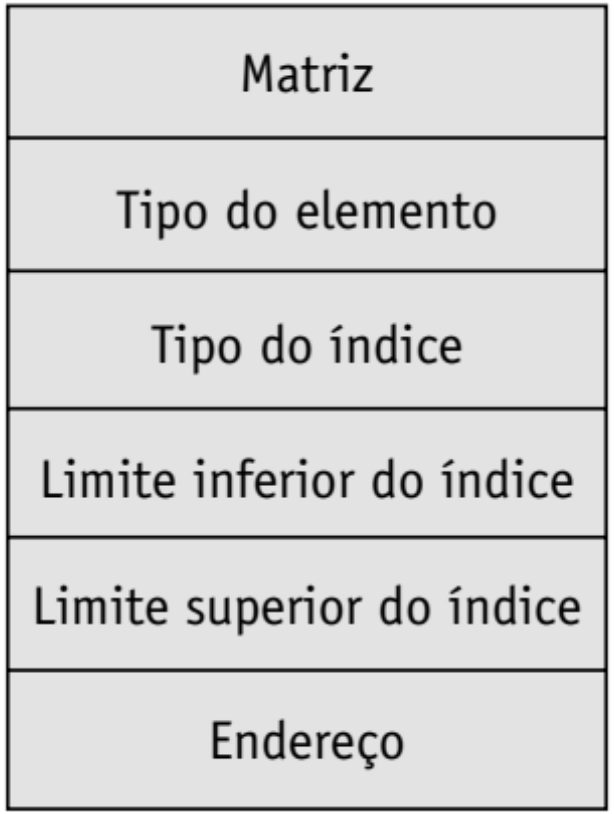
Cube (:, :, 2:3)

Implementação de Matrizes

- Implementar matrizes requer esforço adicional em comparação com tipos primitivos.
- O compilador precisa gerar funções de acesso que traduzem índices em endereços de memória.
- Função de acesso mapeia a expressão de índices para o endereço de um elemento.
- Função de acesso (vetor 1D):

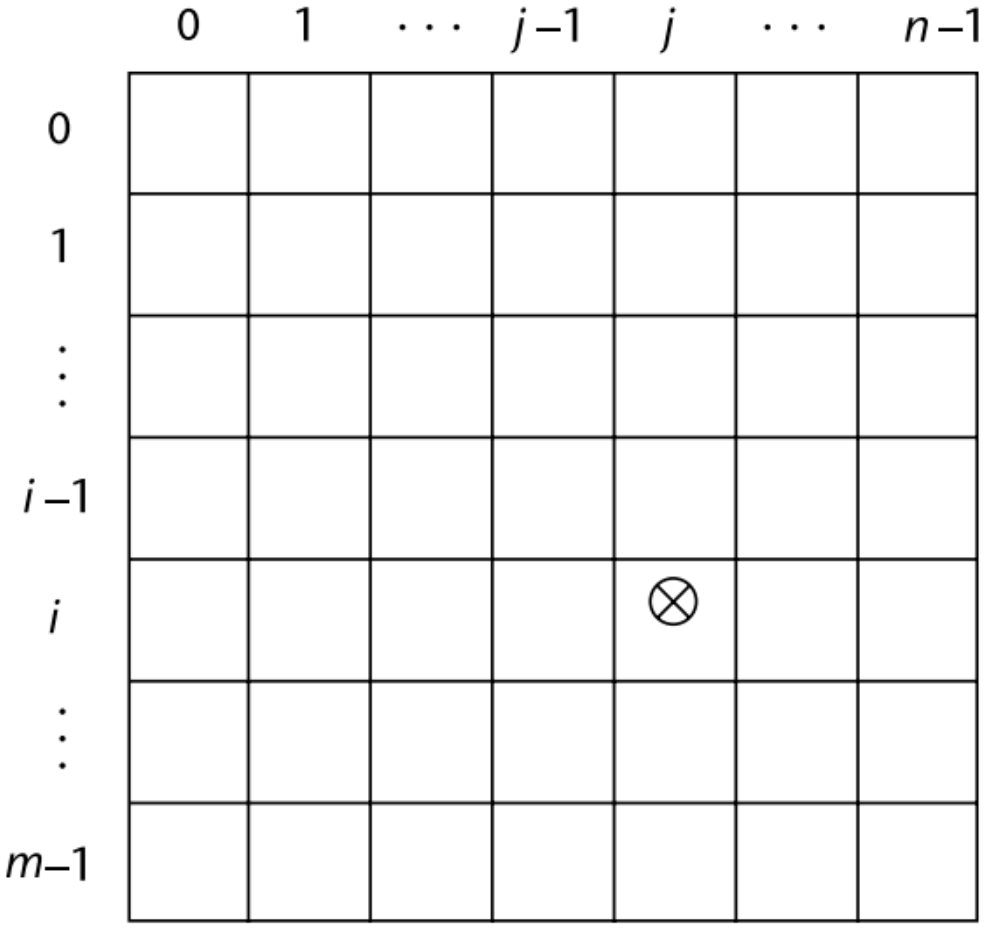
endereço(list[k]) =endereço(list[limite_inferior])

+ (k - limite_inferior) * tamanho_elemento



Implementação de Matrizes

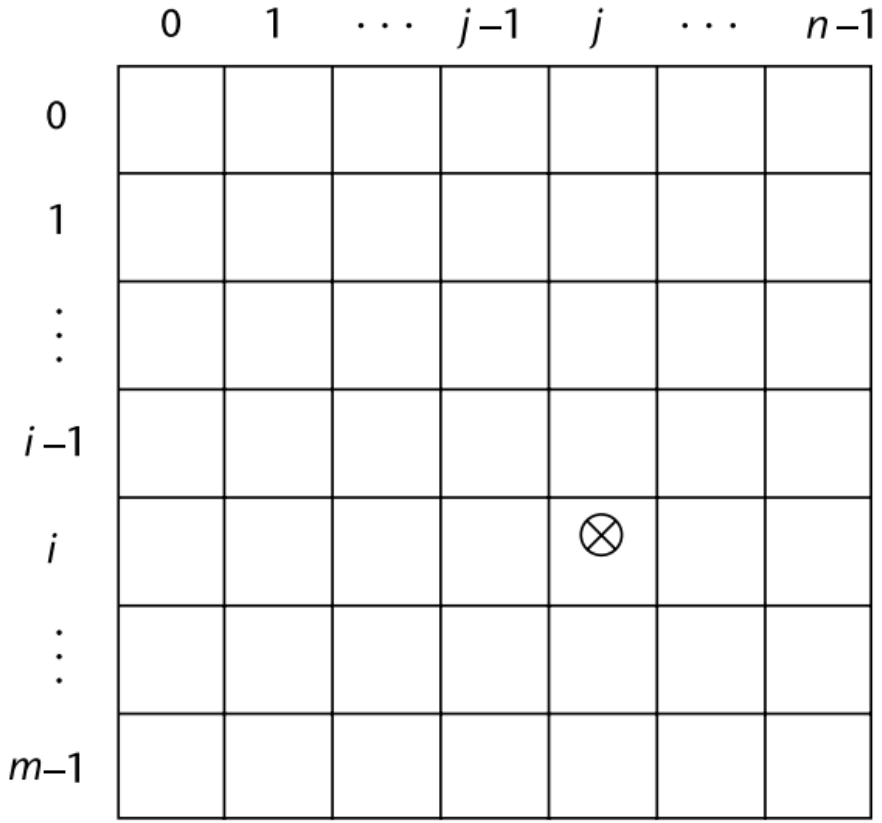
- Memória do hardware é linear: matrizes 2D ou mais precisam ser mapeadas em 1D.
- Dois esquemas principais:
  - Row-major order (por linhas): usado pela maioria das linguagens (C, Java, etc.).
  - Column-major order (por colunas): usado em Fortran.
- O compilador gera fórmulas de mapeamento a partir do endereço base + índices.



Implementação de Matrizes

- Fórmula geral para ordem principal de linha (row-major):

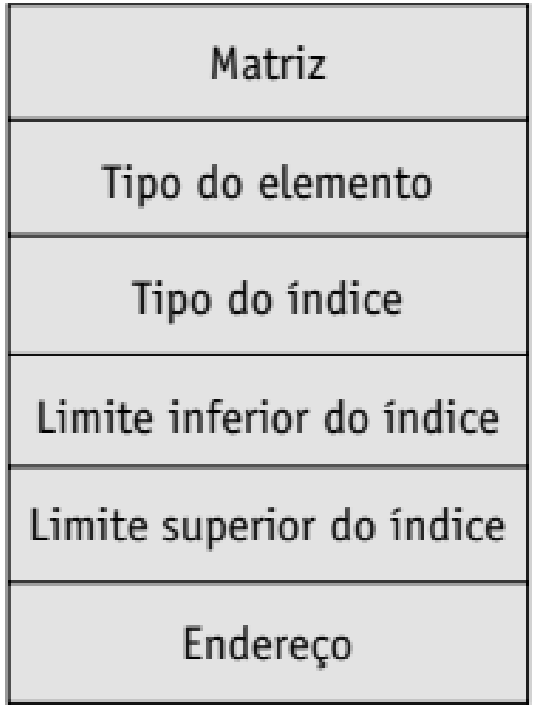
endereço(A[i,j]) = endereço(A[0,0]) + ((i * n) + j) * tamanho_elemento

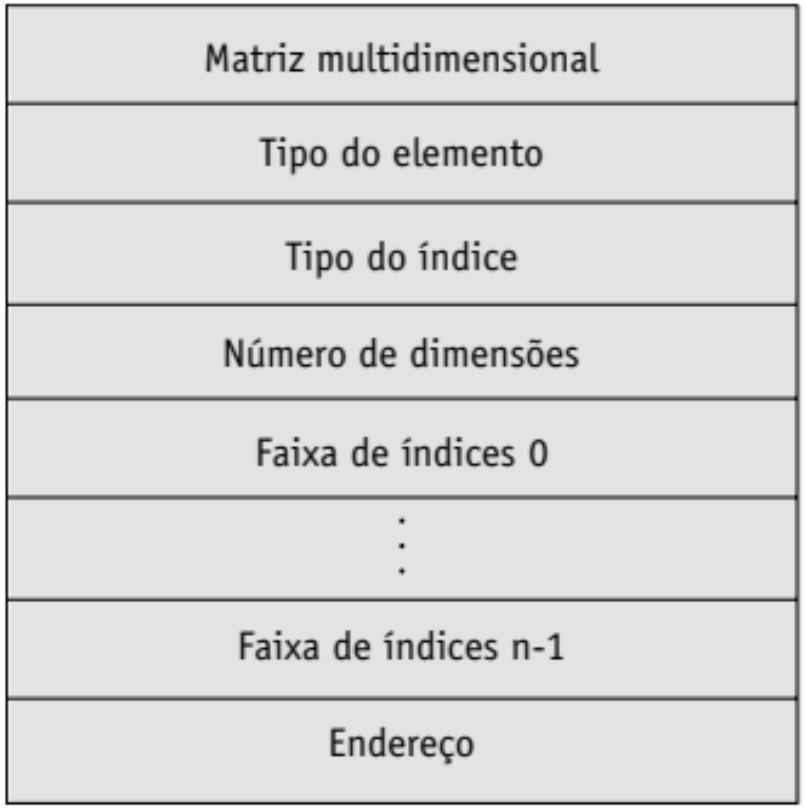


Implementação de Matrizes

- Descritor para matrizes 1D:
  - Tipo do elemento, tipo do índice, limites inferior/superior, endereço base.
- Descritor para matrizes multidimensionais:
  - Inclui também número de dimensões e faixas de cada índice.
- Necessários para gerar e validar as funções de acesso em tempo de compilação ou execução.

Implementação de Matrizes





Matrizes Associativas

- Coleção não ordenada de elementos indexados por **chaves**, em vez de índices numéricos.
- Cada elemento = **(chave, valor)**
- Vantagem: índices não precisam ser armazenados (chaves definidas pelo usuário).
- Estruturas dinâmicas: tamanho cresce ou diminui em tempo de execução:
  - normalmente armazenadas na heap.
- Suporte:
  - Direto: Python, Ruby, Lua, Perl.
  - Bibliotecas de classes: Java, C++, C#, F#.

Matrizes Associativas

- Exemplo em Perl
- Elementos dinâmicos: crescem e diminuem.
- Operadores úteis: exists, keys, values, each.

```
%salaries = (
   "Gary"   => 75000,
   "Perry"  => 57000,
   "Mary"   => 55750,
   "Cedric" => 47850
);

# Acesso
$salaries{"Perry"} = 58850;

# Remoção
delete $salaries{"Gary"};

# Verificação de existência
if (exists $salaries{"Shelly"}) { ... }
```

Matrizes Associativas

- Exemplo em Python (dicionário)
- Em Python, **dict** = equivalente direto às matrizes associativas.
- Muito usado por legibilidade, flexibilidade e eficiência em buscas.

```
salaries = {
    "Gary": 75000,
    "Perry": 57000,
    "Mary": 55750,
    "Cedric": 47850
}

# Acesso
salaries["Perry"] = 58850

# Remoção
del salaries["Gary"]

# Verificação
if "Shelly" in salaries:
    ...
```

Matrizes Associativas

- Operações básicas:
  - Atribuição: dict["chave"] = valor
  - Acesso: dict["chave"]
  - Remoção: del dict["chave"]
  - Iteração: for k, v in dict.items():
- Eficiência: busca rápida usando funções de dispersão (hash functions).
- Implementação: Perl otimiza reordenação quando cresce; Python organiza dinamicamente.

Registros

- Um **registro** é um agregado de elementos de dados possivelmente heterogêneos.
- Elementos são chamados de **campos**, identificados por **nomes**, não por índices.
- Usados para modelar dados compostos (ex.: informações de um estudante: nome, matrícula, média).
- Diferença de matrizes heterogêneas:
  - Matrizes heterogêneas: elementos são referências espalhadas no heap.
  - Registros: elementos armazenados de forma contígua na memória.
- **Questões de projeto**:
  - Qual a forma sintática das referências a campos?

Registros

- Registros em COBOL usam **números de nível** para mostrar hierarquia.
- PICTURE define o formato de armazenamento (neste caso, strings e números).
- Hierarquia:
  - 01: registro principal.
  - 02: subregistro.
  - 05: campos individuais.
  - Referência a campo:
    - EMPLOYEE-NAME OF EMPLOYEE-RECORD.

```
01 EMPLOYEE-RECORD.
  02 EMPLOYEE-NAME.
    05 FIRST  PICTURE IS X(20).
    05 MIDDLE PICTURE IS X(10).
    05 LAST   PICTURE IS X(20).
  02 HOURLY-RATE PICTURE IS 99V99.
```

Registros

- Ada usa registros aninhados de forma ortogonal.
- Clareza na declaração: registros podem conter outros registros.
- Mostra composição de tipos complexos com maior legibilidade.

```
type Emp_Name_Type is record
   First : String(1..20);
   Mid   : String(1..10);
   Last  : String(1..20);
end record;

type Emp_Rec_Type is record
   Emp_Name   : Emp_Name_Type;
   HourlyRate : Float;
end record;
```

Registros

- Em C e C++, registros são implementados como **struct**.
- Estruturas em C++ permitem aninhamento, similar a Ada.
- Em C++ e C#, struct também é usada para encapsulamento e organização de dados.
- Diferença para classes: struct não tem, por padrão, herança e métodos (em C++ pode ter).

```
struct Emp_Name_Type {
   string first;
   string middle;
   string last;
};

struct Emp_Rec_Type {
   Emp_Name_Type Emp_name;
   float hourly_rate;
};
```

Registros

Referências a Campos

- COBOL: notação OF
  - Middle OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
- Java / C#: notação por pontos
  - Employee_Record.Employee_Name.Middle
- Lua: campos de registros podem ser acessados como tabelas
  - employee["name"] ou employee.name

Registros

- Registros e matrizes: estruturas relacionadas, mas com usos diferentes.
- Matrizes: usadas quando todos os elementos são do mesmo tipo e/ou processados da mesma forma.
  - Suportam melhor índices dinâmicos e acesso sequencial.
- Registros: usados quando os elementos são heterogêneos e os campos são acessados de forma independente.
  - Campos têm nomes (índices literais) em vez de índices numéricos.
  - Fornecem acesso eficiente e seguro.

Registros

- Campos de registros são armazenados em posições de memória adjacentes.
- O tamanho dos campos pode variar: método de acesso diferente das matrizes.
- Cada campo possui um deslocamento relativo ao início do registro.
- Esse deslocamento é usado para calcular o endereço do campo.
- Não há necessidade de descritores em tempo de execução: basta a estrutura definida em tempo de compilação.

Verificação de Tipos

- Garante que os operandos de um operador sejam de tipos compatíveis.
- **Tipo compatível**:
  - Válido diretamente para o operador.
  - Ou pode ser convertido automaticamente (**coerção**).
- **Erro de tipo**: ocorre quando um operador é aplicado a um operando de tipo inválido.
  - Exemplo: em C original, passar um int para função que esperava float gerava erro.

Verificação de Tipos

- **Verificação Estática**
  - Feita em tempo de compilação.
  - Mais segura: detecta erros cedo.
  - Menos flexível para o programador.
- **Verificação Dinâmica**
  - Feita em tempo de execução (ex.: JavaScript, PHP).
  - Mais flexível, mas pode deixar erros passarem até a execução.
- Relação:
  - Se vínculos de tipos são estáticos: quase toda verificação é estática.
  - Se vínculos são dinâmicos: verificação deve ser dinâmica.

Tipagem Forte

- Tipagem forte se tornou uma característica valorizada após a revolução da programação estruturada (anos 1970).
- Uma linguagem é **fortemente tipada** se erros de tipos **são sempre detectados**.
- Isso exige que os tipos de todos os operandos sejam determinados em **tempo de compilação** ou **execução**.
- **Vantagens**:
  - Detecta usos incorretos de variáveis.
  - Garante segurança e consistência no uso dos tipos.

Tipagem Forte

- **Não fortemente tipadas**: C e C++ (permitem **uniões** sem verificação de tipos).
- **Fortemente tipadas**: ML, F#.
- **Quase fortemente tipadas**: Java e C# (apesar de baseados em C++, possuem menos brechas).
- **Dinamicamente fortes**: Ruby, Python (verificação em tempo de execução).

```
a = "10"
b = 5
print(a + b)
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Tipagem Forte

- **Coerção** = conversão automática de tipos (ex: int para float).
- **Problema**: coerções podem enfraquecer a tipagem forte, pois mascaram erros.
- Exemplo:

```
int a, b;
float d;
// a + d é aceito: 'a' é convertido implicitamente para float
```

Quanto **mais coerções** uma linguagem aceita, **menos forte** é sua tipagem.

Equivalência de Tipos

- Compatibilidade x Equivalência
  - Compatibilidade: quando um tipo pode ser convertido (coerção) para outro e usado em uma operação.
  - Equivalência: dois tipos só podem ser usados juntos se forem considerados exatamente equivalentes.
- **Definição**
  - Dois tipos são **equivalentes** em uma expressão se um operando de um tipo pode substituir outro, **sem coerção**.
- Importância
  - Influencia o projeto dos tipos de dados e das operações disponíveis.
  - Garantia de segurança e consistência: variáveis equivalentes podem receber valores umas das outras.

Equivalência de Tipos

- Tipos simples (inteiros, reais): regras são rígidas e claras.
- Tipos estruturados (matrizes, registros, tipos definidos pelo usuário): regras ficam mais complexas.
- Exemplo em Ada:

    type Celsius is new Float;

    type Fahrenheit is new Float;

Estruturas idênticas, mas **não são equivalentes** (tratados como tipos distintos).

Equivalência de Tipos

- **Equivalência de tipos por nome** significa que duas variáveis são equivalentes se são definidas na mesma declaração ou em declarações que usam o mesmo nome de tipo.
- Fácil de implementar, mas altamente restritiva.
  - Subintervalos de tipos inteiros não são equivalentes a tipos inteiros.
  - Exemplo em Ada:

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

- Os tipos das variáveis count e index não seriam equivalentes; count não poderia ser atribuída a index ou vice-versa.

Equivalência de Tipos

- **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.
- Exemplo em Ada:

```
type Celsius = Float;
type Fahrenheit = Float;
```

- Sob equivalência por estrutura, Celsius e Fahrenheit são considerados equivalentes, embora semanticamente representem conceitos diferentes.

Equivalência de Tipos

- Considere o problema de dois tipos estruturados:
  - Dois tipos de registros são equivalentes se forem estruturalmente iguais mas usarem nomes de campos diferentes?
  - Dois tipos de arrays são equivalentes se forem iguais exceto pelos subscritos diferentes? (ex.: [1..10] e [0..9])
- Dois tipos de enumeração são equivalentes se seus componentes forem escritos de forma diferente?
- Com equivalência por estrutura de tipos, não é possível diferenciar entre tipos da mesma estrutura
  - (ex.: diferentes unidades de velocidade, ambos float).

Paradigmas de Linguagens de Programação

Tipos de Dados