



S201 – Paradigmas de Programação

TIPOS DE DADOS

Marcelo Vinícius Cysneiros Aragão

marcelovca90@inatel.br

TÓPICOS

Tópicos

- Introdução
- Tipos de dados primitivos
- Cadeias de caracteres
- Tipos ordinais definidos pelo usuário
- Tipos de matrizes
- Matrizes associativas
- Registros
- Uniões
- Ponteiros e referências
- Verificação de tipos
- Tipagem forte
- Equivalência de tipos
- Teoria e tipos de dados

INTRODUÇÃO

Introdução

- Um **tipo de dado** define uma coleção de valores de dados e um conjunto de operações pré-definidas sobre eles.
- Programas de computador produzem resultados por meio da manipulação de dados.
- É interessante que os tipos de dados disponíveis na linguagem usada casem com os objetos no espaço do problema no mundo real.

Introdução

- Evolução:
 - Fortran I: listas ligadas e as árvores binárias eram implementadas com vetores.
 - COBOL permitia que os programadores especificassem a precisão dos valores de dados decimais e fornecessem um tipo de dados estruturado para registros de informação .
 - PL/I: capacidade da especificação de precisão também para tipos inteiros e de ponto flutuante (isso foi incorporado ao Ada e Fortran); grande quantidade de tipos básicos.
 - ALGOL 68: poucos tipos básicos e operadores de definição de estruturas flexíveis:

Introdução

- Tipos de dados definidos pelo usuário:
 - permitem o projeto de estruturas de dados adequadas a cada necessidade.
 - legibilidade melhorada, por meio do uso de nomes significativos para os tipos
 - aderentes com a verificação de tipos de variáveis
 - ajudam na facilidade de fazer modificações
- Tipos abstratos de dados: suportados pela maioria das LPs de 1985 em diante
 - Ideia: a interface de um tipo, visível para o usuário, é separada da representação e do conjunto de operações sobre valores desse tipo, ocultos do usuário.
 - Todos os tipos fornecidos por uma LP de alto nível são abstratos.

Introdução

- É conveniente pensarmos em variáveis em termos de descritores.
- Um **descritor** é a coleção de atributos de uma variável.
 - Atributos estáticos: os descritores são necessários apenas em tempo de compilação.
 - Atributos dinâmicos: parte ou todo o descritor deve ser mantido durante a execução.
- Em todos os casos, os descritores são usados para verificação de tipos e para construir o código para as operações de alocação e liberação.

Introdução

- A palavra *objeto* é normalmente associada com o valor de uma variável e com o espaço que ocupa.
- Aqui, reservamos a palavra *objeto* exclusivamente para instâncias de tipos de dados abstratos definidos pelo usuário, em vez de também usá-la para os valores de variáveis de tipos pré-definidos
- Em LPs orientadas a objetos, todas as instâncias de todas as classes (pré-definidas ou definidas pelo usuário) são chamadas de objetos.

TIPOS DE DADOS PRIMITIVOS

Tipos de Dados Primitivos

- Tipos de dados não definidos em termos de outros são chamados de **tipos de dados primitivos**.
- Praticamente todas as linguagens de programação fornecem um conjunto de tipos de dados primitivos.
- Alguns dos tipos primitivos são meramente reflexos de *hardware* – por exemplo, a maioria dos tipos inteiros.
- Outros requerem apenas um pouco de suporte externo ao *hardware* para sua implementação.

Tipos de Dados Primitivos: Inteiro

- O tipo de dados primitivo numérico mais comum é o **inteiro**.
- Muitos computadores agora suportam diversos tamanhos de inteiros.
 - Exemplo: Java inclui 4 tamanhos inteiros com sinal: **byte**, **short**, **int** e **long**.
- Algumas linguagens, como C++ e C#, incluem tipos inteiros sem sinal, que são geralmente usados para dados binários.
 - <https://dev.mysql.com/doc/refman/8.0/en/integer-types.html>
 - <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/builtin-types/integral-numeric-types>

Tipos de Dados Primitivos: Ponto flutuante

- Tipos de dados de **ponto flutuante** modelam números reais, mas as representações são apenas aproximações para muitos valores reais.
 - Por exemplo, nenhum dos números fundamentais π ou e pode ser corretamente representado em notação de ponto flutuante.
- Na verdade, nenhum desses números pode ser representado precisamente em qualquer espaço finito.
- Outro problema com tipos ponto flutuante é a perda de precisão por meio de operações aritméticas.
 - https://docs.oracle.com/cd/E19957-01/806-3568/ngc_goldberg.html
 - <https://softwareengineering.stackexchange.com/questions/62948/what-can-be-done-to-programming-languages-to-avoid-floating-point-pitfalls>

Tipos de Dados Primitivos: Ponto flutuante

- Valores de ponto flutuante são representados como frações expoentes, uma forma emprestada da notação científica.
- A maioria das máquinas mais novas usam o formato padrão para ponto flutuante da IEEE chamado IEEE Floating-Point Standard 754:
 - <https://standards.ieee.org/standard/754-2019.html>
 - <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>

Tipos de Dados Primitivos: Ponto flutuante

- A maioria das linguagens inclui dois tipos de ponto flutuante, chamados de **float** e **double**.
 - O tipo float é o tamanho padrão, e normalmente é armazenado em quatro bytes de memória.
 - O tipo double é fornecido para situações nas quais partes fracionárias maiores e/ou uma faixa de expoentes maior são necessárias.

Tipos de Dados Primitivos: Ponto flutuante

- Variáveis double costumam ocupar o dobro de armazenamento das float e fornecem ao menos o dobro do número de bits de fração.
- A coleção de valores que podem ser representados por um tipo ponto flutuante é definida em termos de precisão e faixa.
 - **Precisão** é a exatidão da parte fracionária de um valor, medida em bits.
 - **Faixa** é a combinação da faixa de frações e, mais importante, de expoentes.

Tipos de Dados Primitivos: Ponto flutuante

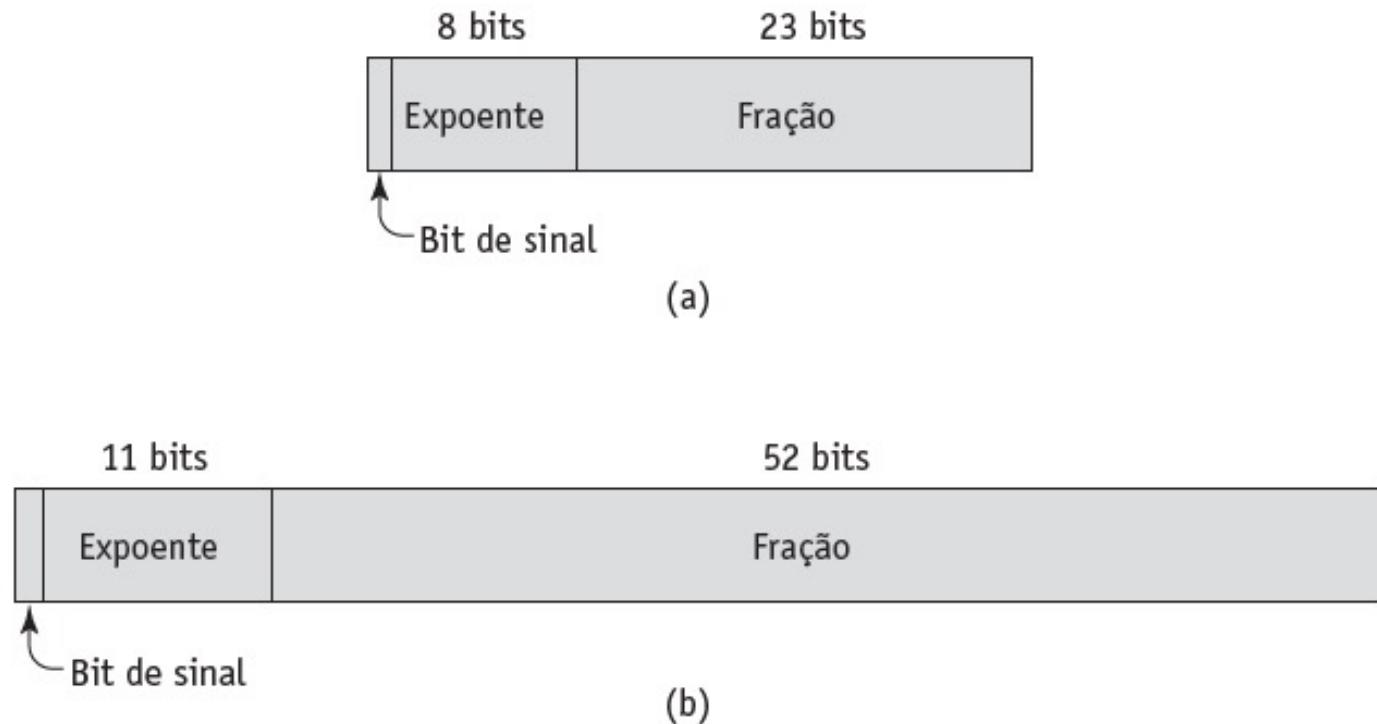


FIGURA 6.1

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

Tipos de Dados Primitivos: Complexo

- Algumas linguagens de programação suportam um tipo de dados complexo – por exemplo, Fortran e Python.
- Valores complexos são representados como pares ordenados de valores de ponto flutuante .
 - Em Python, a parte imaginária de um literal complexo é especificada seguindo a por j ou J – por exemplo, (7 + 3j).
 - https://github.com/marcelovca90-inatel/TP555-exercicios/blob/main/lista_0/resolucao.ipynb
 - <https://numpy.org/doc/stable/user/basics.types.html>
- Linguagens que suportam um tipo complexo incluem operações para aritmética em valores complexos.

Tipos de Dados Primitivos: Decimal

- Computadores de grande porte, responsáveis por aplicações de sistemas de negócios, costumam suportar tipos de dados **decimais**.
- Tipos de dados decimais armazenam um número fixo de dígitos decimais, com o ponto decimal em uma posição fixa no valor.
 - <https://www.mainframestechhelp.com/tutorials/cobol/assumed-decimal-point-data-type.htm>
- Esses são os tipos de dados primários para processamento de dados de negócios e, dessa forma, são essenciais para o COBOL.
- C# também tem um tipo de dados decimal.
 - http://www.macoratti.net/12/12/c_num1.htm

Tipos de Dados Primitivos: Decimal

- Tipos decimais têm a vantagem de serem capazes de **armazenar precisamente valores decimais**, ao menos dentro de **uma faixa restrita**, o que não pode ser feito com tipos de ponto flutuante.
 - Por exemplo, o número 0.1 (em decimal) pode ser representado exatamente em um tipo decimal, mas não em um tipo de ponto flutuante
 - <https://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/>
 - As desvantagens dos tipos decimais são que a **faixa de valores é restrita** porque nenhum expoente é permitido, e sua **representação em memória é dispendiosa**.

Tipos de Dados Primitivos: Decimal

- Tipos decimais são armazenados de maneira parecida com as cadeias de caracteres, usando códigos binários para os dígitos decimais.
- Essas representações são chamadas de **decimais codificados em binário (BCD – binary coded decimal)**.
 - <https://www.geeksforgeeks.org/bcd-or-binary-coded-decimal/>
- Em alguns casos, são armazenados um dígito por byte, mas em outros são agrupados em dois dígitos por byte.
- De qualquer forma, ocupam mais armazenamento do que as representações binárias.

Tipos de Dados Primitivos: Booleanos

- Tipos **booleanos** são talvez os mais simples.
- Sua faixa de valores tem apenas dois elementos: um para verdadeiro e outro para falso.
 - <https://stackoverflow.com/questions/2764017/is-false-0-and-true-1-an-implementation-detail-or-is-it-guaranteed-by-the>
- Eles foram introduzidos em ALGOL 60 e incluídos na maioria das linguagens de propósito geral projetadas desde 1960.
 - Uma exceção popular é o C89, no qual expressões numéricas são usadas como condicionais.
 - Em tais expressões, todos os operandos com valores diferentes de zero são considerados verdadeiros, e zero é considerado falso.

Tipos de Dados Primitivos: Booleanos

- Apesar do C99 e do C++ terem um tipo booleano, também permitem que expressões numéricas sejam usadas como se fossem booleanas.
 - Esse não é o caso com linguagens subsequentes, como Java e C#.
- Booleanos costumam representar **escolhas** ou ser usados como **flags**.
- Apesar de outros tipos, como inteiros, poderem ser usados para tais propósitos, o uso de tipos booleanos é mais **legível**.
- *Booleanos* poderiam ser representados por **um único bit**, mas como muitas máquinas não fornecem acesso direto a um único bit de memória, eles são armazenados na menor célula eficientemente endereçável (**um byte**).

Tipos de Dados Primitivos: Caracteres

- Dados na forma de caracteres são armazenados nos computadores como **codificações numéricas**.
- Tradicionalmente, a codificação mais usada era o **ASCII** (Padrão Americano de Codificação para Intercâmbio de Informação – *American Standard Code for Information Interchange*) de 8 bits, que usa os valores de 0 a 127 para codificar 128 caracteres diferentes.
- O **ISO 8859-1** é outra codificação de 8 bits para caracteres, mas ele permite 256 caracteres diferentes.
- Ada 95 usa o ISO 8859-1.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	.	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	:	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

Tipos de Dados Primitivos: Caracteres

- Por causa da **globalização** dos negócios e da necessidade de os computadores se comunicarem com outros computadores pelo mundo, o conjunto de caracteres ASCII se tornou inadequado.
- Em resposta, em 1991, o Consórcio Unicode publicou o padrão **UCS-2**, um conjunto de caracteres de 16 bits.
- Essa codificação de caracteres é geralmente chamada de **Unicode**.
- Unicode inclui os caracteres da maioria das linguagens naturais.
 - Exemplo: Unicode inclui o alfabeto Cirílico (Sérvia), e os dígitos Tailandeses. Os primeiros 128 caracteres do Unicode são idênticos aos do ASCII.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Latin script

Non-Latin European scripts

African scripts

Middle Eastern and Southwest Asian scripts

South and Central Asian scripts

Southeast Asian scripts

East Asian scripts

CJK characters

Indonesian and Oceanic scripts

American scripts

Notational systems

Symbols

Private use

UTF-16 surrogates

Unallocated code points

As of Unicode 13.0

Tipos de Dados Primitivos: Caracteres

- Java foi a primeira LP popular a usar o conjunto de caracteres Unicode.
- Desde então, ele foi adotado em JavaScript, Python, Perl e C#. .
- Após 1991, o Consórcio Unicode, em cooperação com a Organização Internacional de Padrões (ISO – *International Standards Organization*) desenvolveu uma codificação de caracteres de 4 bytes chamada UCS-4 ou **UTF-32**, que é descrita pelo padrão ISO/IEC 10646, publicado em 2000.
 - <https://stackoverflow.com/questions/496321/utf-8-utf-16-and-utf-32>
- Para fornecer os meios de processar codificações de caracteres simples, a maioria das linguagens de programação inclui um tipo primitivo para eles.
- Python suporta caracteres únicos apenas como cadeias de caracteres de tamanho 1.

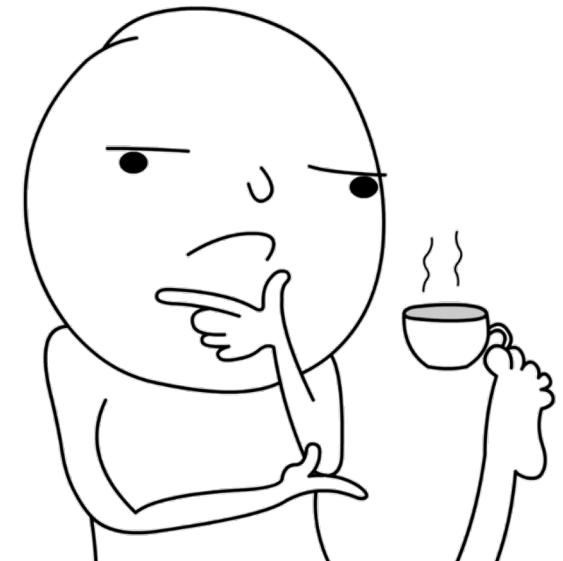
CADEIAS DE CARACTERES

Cadeias de Caracteres

- Um **tipo cadeia de caracteres** é um tipo no qual os valores consistem em sequências de caracteres.
- Constantes do tipo cadeias de caracteres são usadas para rotular a saída, e a entrada e saída de todos os tipos de dados é geralmente feita em termos de cadeias.
- São um tipo essencial para todos os programas que realizam manipulação de caracteres.

Cadeias de Caracteres: Questões de projeto

- As duas questões de projeto específicas às cadeias de caracteres mais importantes são:
 - As cadeias devem ser apenas um **tipo especial** de vetor de caracteres ou um **tipo primitivo**?
 - As cadeias devem ter **tamanho estático** ou **dinâmico**?



Cadeias de Caracteres: Operações

- As operações comuns em cadeias são:
 - Atribuição
 - Concatenação
 - Referência a subcadeias (*slices*)
 - Comparação
 - Casamento de padrões (*pattern matching*)
- Exemplos em Java:
 - <https://replit.com/join/dfswkfpk-marcelocysneiro>

Cadeias de Caracteres: Exemplos em algumas LPs

- C e C++
 - Não primitivo
 - Usa vetores de `char` e uma biblioteca de funções que prove operações
- SNOBOL4 (uma linguagem de manipulação de cadeias de caracteres)
 - Primitiva
 - Muitas operações, incluindo casamento de padrões elaborados
- Fortran e Python
 - Tipo primitivo com atribuição e diversas operações
- Java
 - Primitivo via a classe `String`
- Perl, JavaScript, Ruby e PHP
 - Fornece casamento de padrões fora da caixa usando expressões regulares

Cadeias de Caracteres: Opções de tamanho de cadeias

- As escolhas de projeto em relação ao tamanho das cadeias são:
 - Tamanho estático
 - Tamanho dinâmico limitado
 - Tamanho dinâmico



Cadeias de Caracteres: Opções de tamanho de cadeias

- **Tamanho estático**

- Neste caso, o **tamanho é definido quando a cadeia é criada**, sendo chamada de uma **cadeia de tamanho estático**.
- Essa é a escolha para as cadeias de Python, para os objetos imutáveis da classe String de Java, assim como para as classes similares na biblioteca de classes padrão de C++, a classe pré-definida String em Ruby e para a biblioteca de classes do .NET disponível para o C#.

Cadeias de Caracteres: Opções de tamanho de cadeias

- **Tamanho dinâmico limitado**

- Neste caso, é permitido que as cadeias tenham **tamanhos variáveis até um máximo declarado e fixado pela definição da variável**, como exemplificado pelas cadeias em C e pelas cadeias no estilo de C em C++.
 - https://www.cplusplus.com/reference/string/string/max_size/
- Essas são chamadas de **cadeias de tamanho dinâmico limitado**.
- Elas podem armazenar qualquer número de caracteres entre zero e o máximo.
- Lembre que as cadeias em C usam um **caractere especial** para indicar o final da cadeia, em vez de manter o tamanho da cadeia.

Cadeias de Caracteres: Opções de tamanho de cadeias

- **Tamanho dinâmico**

- Neste caso, as cadeias podem ter **tamanho variado sem um máximo**, como em JavaScript, Perl e a biblioteca padrão de C++.

- Essas são chamadas de **cadeias de tamanho dinâmico**, opção que requer a sobrecarga da alocação e liberação de armazenamento dinâmico, mas fornece a flexibilidade máxima.

- Ada 95 suporta as três opções de tamanho de cadeias.

- https://en.wikibooks.org/wiki/Ada_Programming/Strings

Cadeias de Caracteres: Avaliação

- Os tipos que representam cadeias são importantes para a **facilidade de escrita** de uma linguagem.
- Trabalhar com cadeias na forma de vetores pode ser mais trabalhoso do que com um tipo primitivo que representa cadeias.
 - Por exemplo, considere uma linguagem que trata cadeias como vetores de caracteres e não tem uma função pré-definida que faz o que `strcpy` em C faz.
 - A simples atribuição de uma cadeia para outra demandaria um laço de repetição.

Cadeias de Caracteres: Avaliação

- A adição de cadeias como um tipo primitivo de uma LP **não é custoso**, nem em termos da linguagem nem da complexidade do compilador.
- Logo, é difícil justificar a omissão de tipos primitivos para cadeias em algumas linguagens contemporâneas.
- É claro, fornecer cadeias por meio de uma **biblioteca padrão** é quase tão conveniente quanto tê-las como um tipo primitivo.
 - <https://www.cplusplus.com/reference/cstring/>

Cadeias de Caracteres: Avaliação

- Operações sobre cadeias como casamento de padrões simples e concatenação são **essenciais** e devem ser incluídas para valores deste tipo.
- Apesar de as cadeias de tamanho dinâmico serem obviamente as mais flexíveis, a **sobrecarga de sua implementação** deve ser pesada em relação à flexibilidade adicional.

Cadeias de Caracteres: Implementação

- **Tamanho estático:** descritor em tempo de compilação
- **Tamanho dinâmico limitado:** pode ser necessário um descritor em tempo de execução para armazenar o tamanho (mas não em C/C++).
- **Tamanho dinâmico:** necessidade de um descritor em tempo de execução; alocação / liberação é o maior problema de implementação.

Static string
Length
Address

Descriptor em tempo de compilação para strings estáticas

Limited dynamic string
Maximum length
Current length
Address

Descriptor em tempo de execução para strings dinâmicas limitadas

TIPOS ORDINAIS DEFINIDOS PELO USUÁRIO

Tipos Ordinais Definidos pelo Usuário

- Um **tipo ordinal** é um no qual a faixa de valores possíveis pode ser facilmente associada com o conjunto dos inteiros positivos.
 - Em Java, os tipos ordinais primitivos são **integer**, **char** e **boolean**.
- Um dos principais tipos ordinais definidos pelo usuário suportados pelas LPs são as **enumerações**.
 - <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

Tipos Ordinais Definidos pelo Usuário: Tipos Enumeração

- Um **tipo enumeração** é um no qual todos os valores possíveis, os quais são constantes nomeadas, são fornecidos, ou enumerados, na definição.
- Tipos enumeração fornecem uma maneira de definir e agrupar coleções de constantes nomeadas, chamadas de **constants de enumeração**.

<<enumeration>>
Couleur
blanc
noir
rouge
vert
bleu
jaune

Tipos Ordinais Definidos pelo Usuário: Tipos Enumeração

- A definição de um tipo enumeração típico é mostrada abaixo em C#:

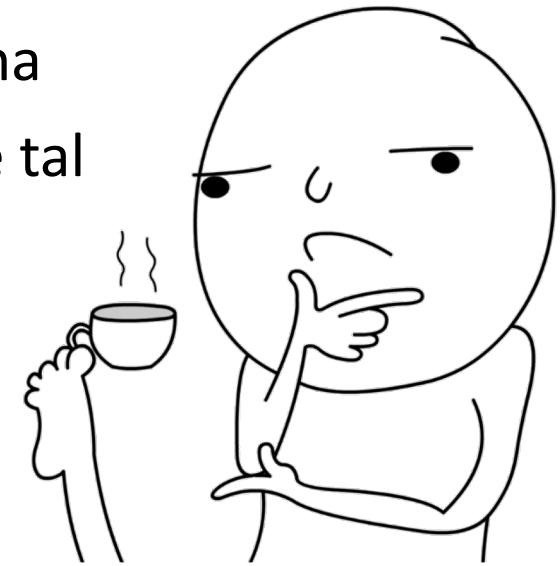
```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/builtin-types/enum>

- As constantes de enumeração são preenchidas implicitamente por atribuições de valores inteiros, 0, 1, ... mas podem ser atribuídos explicitamente quaisquer literais inteiros na definição do tipo.

Tipos Ordinais Definidos pelo Usuário: Questões de projeto

- As questões de projeto para tipos enumeração são:
 - 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?
 - Exemplo em Java (enums Color com representação int e float).
 - Os valores de enumeração são convertidos para inteiros?
 - <http://linguagemc.com.br/enum-em-c/>
 - Há outros tipos que são convertidos para um tipo enumeração?



Tipos Ordinais Definidos pelo Usuário: Questões de projeto

- Todas essas questões de projeto são relacionadas à **verificação de tipos**.
 - Se uma variável de enumeração é convertida para um tipo numérico, existe pouco controle sobre sua faixa de operações legais ou sobre sua faixa de valores.
 - Se um valor do tipo `int` é convertido para um tipo enumeração, uma variável do tipo enumeração pode ter quaisquer valores inteiros atribuídos a ela, independentemente de ela representar uma constante de enumeração ou não.

Tipos Ordinais Definidos pelo Usuário: Avaliação

- Tipos enumeração podem fornecer vantagens em dois sentidos:
 - A **legibilidade** é melhorada de uma maneira muito direta.
 - Valores nomeados são facilmente reconhecidos, enquanto valores codificados não.
 - Exemplo em Java (enum MyColor).

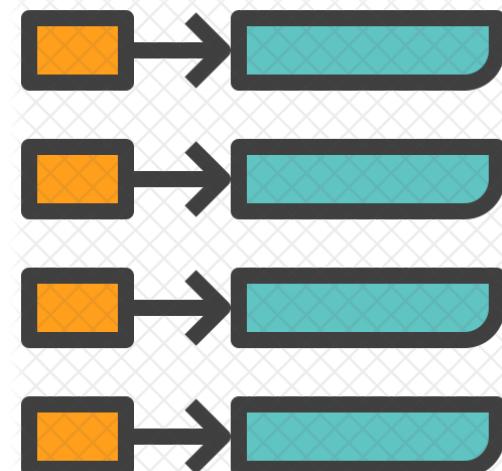
Tipos Ordinais Definidos pelo Usuário: Avaliação

- Tipos enumeração podem fornecer vantagens em dois sentidos:
 - Na área da **confiabilidade**, os tipos enumeração fornecem duas vantagens:
 - Nenhuma operação aritmética é permitida em tipos enumeração. Isso previne adicionar dias da semana, por exemplo.
 - Nenhuma variável de enumeração pode ter um valor atribuído a ela fora de sua faixa.
 - C# e Java 5.0 fornecem melhor suporte para enumeração do que C++ porque as variáveis de tipo de enumeração nessas linguagens não são coagidas em tipos inteiros
 - Continuação do exemplo em Java (enum MyColor).

MATRIZES ASSOCIATIVAS

Matrizes Associativas

- Uma **matriz associativa** é uma coleção não ordenada de elementos de dados indexados por um número igual de valores chamados de **chaves**.
- No caso das matrizes associativas, os índices nunca precisavam ser armazenados (por causa de sua regularidade).
- As chaves definidas pelos usuários, entretanto, devem ser armazenadas na estrutura.
- Então, cada elemento de uma matriz associativa é um **par de entidades, uma chave e um valor**.



Matrizes Associativas

- Usaremos matrizes associativas de **Perl** para ilustrar a estrutura de dados.
- Estas estruturas são também suportadas diretamente por diversas LPs:
 - Python: <https://docs.python.org/3/tutorial/datastructures.html#dictionaries>
 - Ruby: http://ruby-for-beginners.rubymonstas.org/built_in_classes/hashes.html
 - Lua: <https://www.lua.org/pil/2.5.html>
 - Java: <https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>
 - C++: <https://www.cplusplus.com/reference/map/>
 - C#: <https://docs.microsoft.com/pt-br/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0>
- A única questão de projeto específica para matrizes associativas é o **formato das referências aos seus elementos**.

Matrizes Associativas: Estrutura e operações

- Em Perl, as matrizes associativas são chamadas de **dispersões (hashes)**, porque na implementação seus elementos são armazenados e obtidos com **funções de dispersão (hash functions)**.
- O **espaço de nomes** para dispersões em Perl é distinto: cada variável de dispersão deve começar com um sinal de **percentual (%)**.
- Cada elemento de dispersão consiste em duas partes: **uma chave**, que é uma cadeia, e **um valor**, que é um escalar (número, cadeia ou referência).



Matrizes Associativas: Estrutura e operações

- Dispersões podem ter valores literais atribuídos a elas com a sentença de **atribuição**.

```
%salaries = ("Gary" => 75000, "Perry" => 57000,  
             "Mary" => 55750, "Cedric" => 47850);
```
- Valores de elementos individuais são referenciados da seguinte forma:
 - O valor da chave é colocado entre chaves e o nome da dispersão é substituído por um nome de variável escalar.
 - Apesar de dispersões não serem escalares, as partes que representam valores nos elementos de dispersão o são, então, **referências a valores de elementos de dispersão usam nomes escalares**.
 - Lembre-se de que nomes de variáveis escalares começam com **cífrão (\$)**.

Matrizes Associativas: Estrutura e operações

- Por exemplo,:

```
$salaries{"Perry"} = 58850;
```

- Um elemento é **adicionado** usando a mesma forma da sentença de atribuição.
- Um elemento pode ser **removido** com o operador **delete**, como em:

```
delete $salaries{"Gary"};
```

- A dispersão inteira pode ser **esvaziada** por meio da atribuição do literal vazio a ela, tal como em :

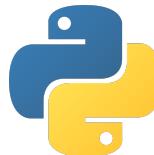
```
@salaries = ();
```

Matrizes Associativas: Estrutura e operações

- O **tamanho de uma dispersão em Perl é dinâmico**: aumenta quando um novo elemento é adicionado e encolhe quando um é apagado (e também quando ela é esvaziada por meio da atribuição do literal vazio).
- Operadores:
 - **exists**: retorna verdadeiro ou falso, dependendo se sua chave operanda é um elemento na dispersão;
 - Exemplo: `if (exists $salaries{"Shelly"}) . . .`
 - **keys**: retorna uma matriz com as chaves da dispersão;
 - **values**: faz o mesmo para os valores da dispersão;
 - **each**: itera sobre os pares de elemento de uma dispersão.

Matrizes Associativas: Estrutura e operações

- As matrizes associativas de Python (chamadas **dicionários**), são similares às de Perl, exceto que os **valores são todos referências a objetos**.
- As matrizes associativas do Ruby são similares às de Python, mas as **chaves podem ser quaisquer objetos**, em vez de apenas cadeias.
- Então, existe uma progressão sobre as chaves no seguinte sentido:
 - em Perl, as chaves devem ser cadeias, para as matrizes;
 - em PHP, as chaves podem ser inteiras ou cadeias;
 - em Ruby, qualquer objeto pode ser uma chave.
- Em algumas LPs, as matrizes são **tanto normais quanto associativas**, podendo ser tratadas como ambas.



Matrizes Associativas: Estrutura e operações

- Em Lua, o tipo **tabela** é a única estrutura de dados.
- Uma tabela Lua é uma matriz associativa na qual tanto as chaves quanto os valores podem ser de **quaisquer tipos**.
- Uma tabela pode ser usada como uma matriz tradicional, como uma matriz associativa ou como um registro.
 - Quando usada como uma matriz tradicional ou como uma matriz associativa, colchetes são colocados em torno das chaves.
 - Quando usada como um registro, as chaves são os nomes dos campos e as referências aos campos podem empregar a notação com pontos (`nome_registro.nome_campo`).



Matrizes Associativas: Estrutura e operações

- Uma matriz associativa é muito melhor do que uma matriz caso as buscas a elementos sejam necessárias, porque a operação de dispersão implícita usada para acessar os elementos é **muito eficiente**.
- Além disso, as matrizes associativas são ideais quando os dados a serem armazenados formam **pares**, por exemplo, com os nomes de empregados e seus salários.
- Entretanto, se cada elemento de uma lista deve ser **processado**, é mais eficiente usar uma matriz.



REGISTROS

Registros

- Um **registro** é um 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.
- Em geral, existe uma necessidade nos programas de modelar coleções de dados que não são do mesmo tipo ou tamanho.
 - Exemplo: informações sobre um estudante universitário
 - nome → cadeia de caracteres
 - número de estudante → inteiro
 - média de notas → ponto flutuante
 - Registros são projetados para esse tipo de necessidade.

Registros

- Os registros têm sido parte de todas as linguagens de programação mais populares, exceto pelas versões anteriores ao Fortran 90, desde os anos 1960, quando foram introduzidos pelo COBOL.
- Em algumas linguagens que têm suporte para a programação orientada a objetos, os registros são simulados com objetos.



Registros

- Em C, C++ e C#, os registros são suportados por meio do tipo **struct**.
- Em C++, estruturas são uma pequena variação das classes.

```
typedef struct // Cria uma STRUCT para armazenar os dados de uma pessoa
{
    float Peso;    // define o campo Peso
    int Idade;    // define o campo Idade
    float Altura; // define o campo Altura
} Pessoa; // Define o nome do novo tipo criado
```

- Link relevante: <https://www.geeksforgeeks.org/difference-c-structures-c-classes/>

Registros

- Em C#, as estruturas também se relacionam com classes, mas são diferentes.

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

- As estruturas em C# são tipos de valores alocados na pilha, de maneira oposta aos objetos de classe, os quais são tipos de referências alocados no monte.

Registros

- Em Python e Ruby, registros podem ser implementados como dispersões.
- Python:
 - <https://stackoverflow.com/questions/35988/c-like-structures-in-python>
 - <https://docs.python.org/3/library/typing.html#typing.NamedTuple>
- Ruby:
 - <https://ruby-doc.org/core-2.5.0/Struct.html>

Registros: Definição

- A diferença fundamental entre um registro e uma matriz é que os elementos de registro, ou **campos**, não são referenciados por **índices**.
- Em vez disso, os campos são nomeados com **identificadores**, e referências para os campos são feitas usando esses identificadores.
- Outra diferença entre matrizes e registros é que os registros em algumas linguagens podem incluir **uniões**, que veremos adiante.

Registros: Definição

- O formato do COBOL de uma declaração de registro, que é parte da divisão de dados de um programa COBOL, é ilustrado no exemplo:

```
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: Definição

- O registro EMPLOYEE-RECORD consiste no registro EMPLOYEE-NAME e no campo HOURLY-RATE.
 - Os numerais 01, 02 e 05 que iniciam as linhas da declaração de registro são números de nível, que indicam por seus valores relativos à **estrutura hierárquica do registro**.
 - Qualquer linha seguida por uma outra com um número de nível mais alto é ela própria um registro.
 - A cláusula PICTURE mostra os formatos das posições de armazenamento de campo, com X(20) especificando 20 caracteres alfanuméricos e 99V99 especificando quatro dígitos decimais com o ponto decimal no meio.

Registros: Definição

- Ada usa uma sintaxe diferente do COBOL para registros.
- Em vez de usar números de níveis, estruturas de registro são indicadas de uma maneira ortogonal ao aninhar declarações de registros dentro de declarações de registro.
- Em Ada, registros não podem ser anônimos – devem ser tipos nomeados.

```
type Employee_Name_Type is record
    First : String (1..20);
    Middle : String (1..10);
    Last : String (1..20);
end record;
type Employee_Record_Type is record
    Employee_Name: Employee_Name_Type;
    Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

Registros: Definição

- As declarações de registro do Fortran 95 requerem que quaisquer registros aninhados sejam previamente definidos como tipos.
 - Então, para o registro de empregado de exemplo, o registro do nome do empregado precisaria ser definido primeiro, e então o registro de empregado simplesmente o nomearia como o tipo de seu primeiro campo.

```
type fruit
    real      :: diameter      ! in mm
    real      :: length        ! in mm
    character :: colour
end type
```

- Em Java e C#, os registros podem ser definidos como classes de dados, com registros aninhados como classes aninhadas.
 - Membros de dados de tais classes servem como os campos do registro.

Registros: Referências a campos de registros

- Referências aos campos individuais dos registros são especificadas sintaticamente por métodos distintos para nomear o campo desejado e os registros que o envolve.
- As referências a campos em COBOL têm a forma
 - `nome_campo OF nome_registro_1 OF ... OF nome_registro_n`
 - onde o primeiro registro nomeado é o menor registro ou o mais interno que contém o campo. O próximo nome de registro na sequência é o do que contém o registro anterior e assim por diante.
 - Exemplo: o atributo **MIDDLE** no registro do exemplo anterior em COBOL pode ser referenciado com `MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD`.

Registros: Referências a campos de registros

- A maioria das outras linguagens usa **notação por pontos** para referências a campos, onde os componentes da referência são conectados por pontos.
- Nomes em notação por pontos têm a **ordem oposta** das referências em COBOL: usam o nome do maior registro que envolve os outros primeiro e o nome do campo por último.
 - Por exemplo, a seguir temos uma referência ao campo Middle no exemplo anterior de registro em Ada: `Employee_Record.Employee_Name.Middle`

Registros: Referências a campos de registros

- C e C++ usam essa mesma sintaxe para referenciar os membros de suas estruturas.
 - <https://www.inf.pucrs.br/~pinho/Laprol/Structs/Structs.htm>
- As referências a campos em Fortran 95 também têm essa forma, com sinais de percentual (%) em vez de pontos.
 - <https://en.wikibooks.org/wiki/Fortran/structures>
- Em PHP, por sua vez, são usadas “setas” (->):
 - <https://wiki.php.net/rfc/structs>

Registros:

Referências a campos de registros

- Uma **referência completamente qualificada** a um campo de um registro é uma referência em que todos os nomes de registro **intermediários**, desde o que envolve todos os outros até o campo específico, são nomeados.
 - Tanto as referências de campos nos exemplos de COBOL quanto de Ada acima são completamente qualificadas.
- Como uma alternativa para as referências qualificadas, o COBOL permite **referências elípticas** aos campos de registro.
 - Nessas, o campo é nomeado, mas qualquer um ou todos os nomes de registros que o envolvem podem ser omitidos, desde que a referência resultante seja não ambígua no ambiente de referenciamento.

Registros: Referências a campos de registros

- Por exemplo, FIRST, FIRST OF EMPLOYEE-NAME e FIRST OF EMPLOYEE-RECORD são referências elípticas para o primeiro nome do empregado no registro COBOL declarado acima.
- Apesar destas referências serem uma conveniência para o programador, elas **requerem que o compilador tenha estruturas de dados e procedimentos elaborados** de forma a identificar corretamente campo referenciado.
- Elas também são, de certa forma, **prejudiciais para a legibilidade**.

Registros: Operações

- A atribuição é uma operação comum em registros.
 - Na maioria dos casos, os tipos dos dois lados devem ser idênticos.
- Ada permite comparações entre registros para igualdade e diferença.
- COBOL fornece a sentença MOVE CORRESPONDING para mover registros.
 - Ela copia um campo do registro de origem especificado para o registro de destino apenas se este tiver um campo com o mesmo nome.

Registros: Operações

- Essa é uma operação útil em aplicações de processamento de dados, nas quais os registros de entrada são movidos para atributos de saída após algumas modificações.
- Como os registros de entrada geralmente têm **muitos campos com os mesmos nomes e propósitos**, como campos em registros de saída, mas **não necessariamente na mesma ordem**, a operação MOVE CORRESPONDING pode poupar muitas sentenças.

Registros: Operações

- Por exemplo, considere as seguintes estruturas em COBOL:

01 INPUT-RECORD.

 02 NAME.

 05 LAST

 PICTURE IS X(20).

 05 MIDDLE

 PICTURE IS X(15).

 05 FIRST

 PICTURE IS X(20).

 02 EMPLOYEE-NUMBER

 PICTURE IS 9(10).

 02 HOURS-WORKED

 PICTURE IS 99.

01 OUTPUT-RECORD.

 02 NAME.

 05 FIRST

 PICTURE IS X(20).

 05 MIDDLE

 PICTURE IS X(15).

 05 LAST

 PICTURE IS X(20).

 02 EMPLOYEE-NUMBER

 PICTURE IS 9(10).

 02 GROSS-PAY

 PICTURE IS 999V99.

 02 NET-PAY

 PICTURE IS 999V99.

- A sentença MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.
copia os campos FIRST, MIDDLE, LAST, e EMPLOYEE-NUMBER do registro de entrada
para o registro de saída.

UNIÕES

Uniões

- Uma **união** é um tipo cujas variáveis podem armazenar diferentes valores de tipos em vários momentos durante a execução de um programa.
- Como um exemplo da necessidade de um tipo união, considere uma tabela de constantes para um compilador, usada para armazenar as constantes encontradas em um programa que está sendo compilado.
- Um campo para cada entrada na tabela e para o valor da constante.
- Suponha que para uma linguagem em particular sendo compilada, os tipos das constantes fossem: inteiro, ponto flutuante e booleano.

Uniões

Nome da constante	Possíveis valores
x	3,1415
	60
	true

- Em termos de gerenciamento de tabela, seria conveniente se a mesma posição, um campo de tabela, pudesse armazenar um valor de qualquer um desses três tipos.
 - Então, todos os valores constantes podem ser endereçados da mesma maneira.
 - O tipo de tal posição é, em certo sentido, a união dos três tipos de valores que ela pode armazenar.

Uniões: Questões de projeto

- O problema da verificação de tipos para as uniões leva a uma importante questão de projeto.
- Outra questão é como representar sintaticamente uma união.
- Em alguns projetos, as uniões estão confinadas a serem partes de estruturas do tipo registro, mas em outras elas não estão.
- Então, as questões de projeto primárias particulares aos tipos união são:
 - A verificação de tipos deve ser obrigatória? Note que qualquer desses tipos de verificação deve ser dinâmico.
 - As uniões devem ser embutidas em registros?

Uniões: Uniões discriminadas *versus* uniões livres

- Fortran, C e C++ fornecem construções para representar uniões nas quais não existe um suporte da linguagem para a verificação de tipos.
- Em Fortran, a sentença `Equivalence` é usada para especificar uniões; em C e C++, é a construção `union`.
 - <https://craftofcoding.wordpress.com/2020/01/14/fortran-re-engineering-equivalence-statements-i/>
- As uniões nessas linguagens são chamadas de **uniões livres**, porque é permitido que os programadores tenham total liberdade em relação à verificação de tipos sobre o uso dessas uniões.

Uniões: Uniões discriminadas *versus* uniões livres

- Por exemplo, considere a seguinte união em C:

```
union flexType {  
    int intEl;  
    float floatEl;  
};  
union flexType el1;  
float x ;  
...  
el1.intEl = 27;  
x = el1.floatEl;
```

- Essa última atribuição não é verificada em relação ao seu tipo, porque o sistema não pode determinar o tipo atual do valor de el1, então ele atribui a representação em cadeia de bits de 27 para a variável **float** x, o que obviamente não faz sentido.

Uniões: Uniões discriminadas *versus* uniões livres

- A verificação de tipos união requer que cada construção de união inclua um indicador de tipo.
- Tal indicador é chamado de **etiqueta (tag)** ou **discriminante**, e uma união com um discriminante é chamada de união discriminada.
- A primeira linguagem a fornecer uniões discriminadas foi o ALGOL 68.
 - <http://euler.vcsu.edu:7000/2139/>
 - Elas são agora suportadas por Ada.

Uniões: Uniões em Ada

- O projeto de Ada para uniões discriminadas, baseado no projeto de sua linguagem antecessora, o Pascal, permite ao usuário especificar **variáveis de um tipo de registro variável** que armazenará apenas um dos valores de tipo possíveis na variação.
- Dessa maneira, o usuário pode dizer ao sistema quando a verificação de tipos pode ser estática.
- Tal variável restrita é chamada de **variável variante restrita**.

Uniões: Uniões em Ada

- A etiqueta de uma variável variante restrita é tratada como uma constante nomeada.
- Registros variantes sem restrições em Ada permitem que os valores de suas variantes **troquem de tipo durante a execução**.
- Entretanto, o tipo da variante pode ser modificado apenas pela atribuição do **registro inteiro**, incluindo o discriminante.
- Isso proíbe **registros inconsistentes**, visto que se o novo registro atribuído for um agregado de dados constante, o valor da etiqueta e o tipo da variante podem ser estaticamente verificados.

Uniões: Uniões em Ada

- O exemplo a seguir mostra um registro variante em Ada:

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
    record
        Filled : Boolean;
        Color : Colors;
    case Form is
        when Circle =>
            Diameter : Float;
        when Triangle =>
            Left_Side : Integer;
            Right_Side : Integer;
            Angle : Float;
        when Rectangle =>
            Side_1 : Integer;
            Side_2 : Integer;
    end case;
end record;
```

Uniões: Uniões em Ada

- A estrutura desse registro variante é mostrada na Figura 6.8.

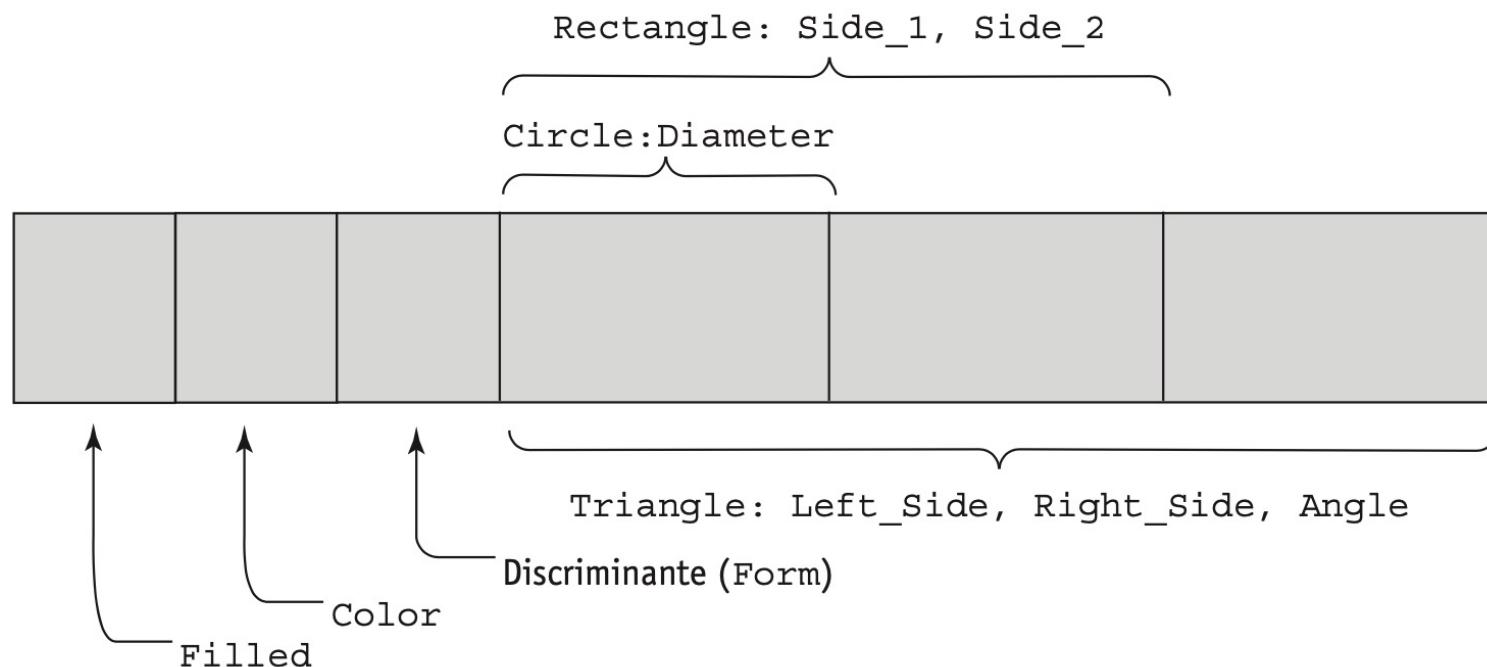


Figura 6.8 Uma união discriminada de três variáveis do tipo Shape (assuma que todas as variáveis são do mesmo tamanho).

Uniões: Uniões em Ada

- As duas sentenças a seguir declaram variáveis do tipo Figure:

```
Figure_1 : Figure;  
Figure_2 : Figure (Form => Triangle);
```

- `Figure_1` é declarada como um registro variante sem restrições que não tem um valor inicial.
- Seu tipo pode ser modificado pela atribuição de um registro completo, incluindo o discriminante:

```
Figure_1 := (Filled => True,  
             Color => Blue,  
             Form => Rectangle,  
             Side_1 => 12,  
             Side_2 => 3);
```

- O lado direito dessa atribuição é um agregado de dados.
- A variável declarada `Figure_2` é restrita a ser um triângulo e não pode ser modificada para outra variante.

Uniões: Uniões em Ada

- Essa forma de união discriminada é **segura**, porque sempre permite a verificação de tipos, apesar de as referências aos campos em variantes sem restrição precisarem ser verificadas dinamicamente.
- Por exemplo, suponha que tivéssemos a seguinte sentença:

```
if (Figure_1.Diameter > 3.0) ...
```

- O sistema em tempo de execução precisaria verificar `Figure_1` para determinar se sua etiqueta `Form` é `Circle`.
- Se não for, seria um erro de tipo referenciar seu diâmetro (`Diameter`).

Uniões: Avaliação

- Uniões são construções **potencialmente inseguras** em algumas linguagens e uma das razões pelas quais Fortran, C e C++ não são fortemente tipadas: essas linguagens não permitem que as referências para suas uniões sejam verificadas em relação aos seus tipos.
- Por outro lado, as uniões podem ser usadas **seguramente**, como em seu projeto na linguagem Ada.
- Em C e C++, as uniões devem ser usadas com cuidado.
- Nem Java nem C# incluem uniões, o que pode ser um reflexo da crescente preocupação com a segurança em linguagens de programação.

Uniões: Implementação

- Uniões são implementadas simplesmente por meio do **uso do mesmo endereço** para cada uma das variantes possíveis.
- Um espaço de armazenamento suficiente para a **maior variante** é alocado.
- No caso das variantes restritas na linguagem Ada, o **espaço exato** de armazenamento pode ser usado porque **não existe variação**.
- A etiqueta de uma união discriminada é armazenada com a variante em uma estrutura similar a um registro.

Uniões: Implementação

- Em tempo de compilação, a descrição completa de cada variante precisa ser armazenada, o que pode ser feito por meio da associação de uma tabela de escolha com a entrada da etiqueta no descritor.
- A tabela de escolha tem uma entrada para cada variante, a qual aponta a um descritor para a variante particular.
- Para ilustrar essa organização, considere o seguinte exemplo em Ada:

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```

Uniões: Implementação

- O descritor para esse tipo poderia ter a forma mostrada abaixo.

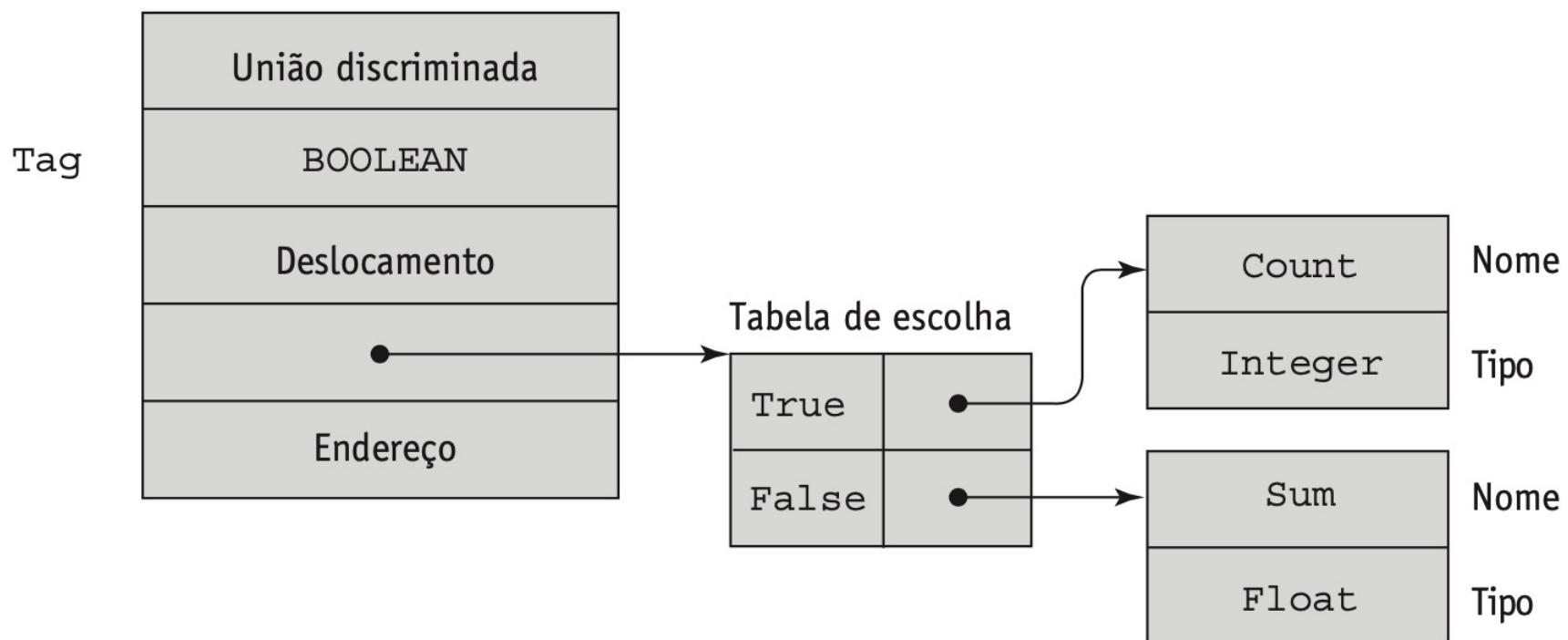


Figura 6.9 Um descritor em tempo de compilação para uma união discriminada.

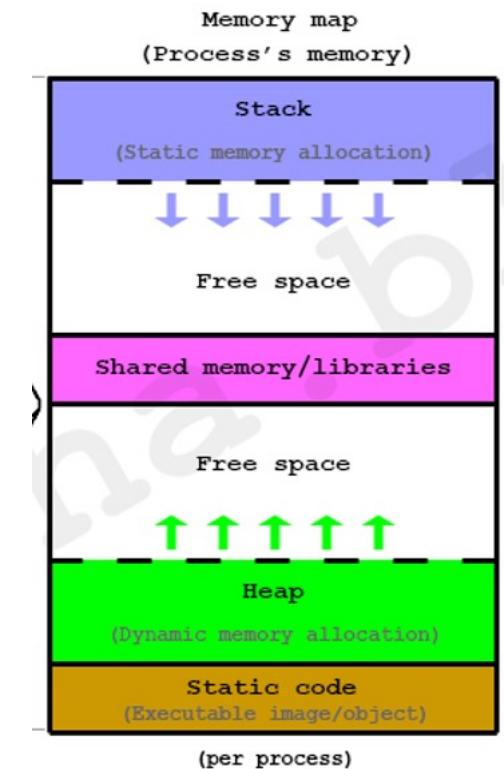
PONTEIROS E REFERÊNCIAS

Ponteiros e Referências

- Um tipo **ponteiro** é um no qual as variáveis têm uma faixa de valores que consistem em endereços de memória e um valor especial, **nil**.
 - O **valor nil não é um endereço válido** e é usado para indicar que um ponteiro não pode ser usado atualmente para referenciar uma célula de memória.
- Ponteiros são projetados para dois tipos de uso.
 1. Fornecem alguns dos poderes do **endereçamento indireto**, frequentemente usado em programação de linguagem de montagem.
 2. 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, chamado de **monte (heap)**.

Ponteiros e Referências

- As variáveis dinamicamente alocadas a partir do monte são chamadas de **variáveis dinâmicas do monte**.
- Em geral, elas não têm identificadores associadas a elas e logo podem ser referenciadas apenas por variáveis dos tipos ponteiro ou referência.
- Variáveis sem nomes são chamadas de **anônimas**.
- É nessa última área de aplicação de ponteiros que surgem as questões de projeto mais importantes.



Ponteiros e Referências

- Ponteiros, diferentemente de matrizes e registros, **não são tipos estruturados**, apesar de serem definidos usando um **operador de tipo** (***** em C e C++ e **access** em Ada).
 - <https://craftofcoding.wordpress.com/2018/03/02/coding-ada-pointers-the-extreme-basics/>
- Além disso, são diferentes de variáveis escalares porque são mais **usados para referenciar alguma outra variável**, em vez de serem usados para armazenar dados de algum tipo.
- Essas duas categorias de variáveis são chamadas de **tipos de referência** e **tipos de valor**, respectivamente.

Ponteiros e Referências

- Ambas as formas de usos de ponteiros facilitam a escrita de programas.
 - Suponha que seja necessário **implementar uma estrutura dinâmica como uma árvore binária** em uma linguagem como o Fortran 77, que não tem ponteiros.
 - Isso requereria que o programador fornecesse e mantivesse uma **coleção de nós de árvore disponíveis**, os quais provavelmente seriam implementados em matrizes paralelas.
 - Além disso, pela **falta de armazenamento dinâmico** em Fortran 77, o programador precisaria adivinhar o número máximo de nós necessários.
 - Essa é uma **maneira deselegante e passível de erros** de trabalhar com árvores binárias.

Ponteiros e Referências: Questões de projeto

- Qualis são o escopo e tempo de vida de uma variável do tipo ponteiro?
- Qual é 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?

Ponteiros e Referências: Operações de ponteiros

- Linguagens que fornecem um tipo ponteiro normalmente incluem duas **operações de ponteiros** fundamentais:
 - **Atribuição:** modifica o valor de uma variável de ponteiro para um endereço útil.
 - **Desreferenciamento:** produz o valor armazenado no local representado pelo valor do ponteiro.
 - Pode ser explícito ou implícito
 - C++ usa uma operação explícita com o operador *
- j = *ptr atribui a j o valor localizado em ptr

Ponteiros e Referências: Operações de ponteiros

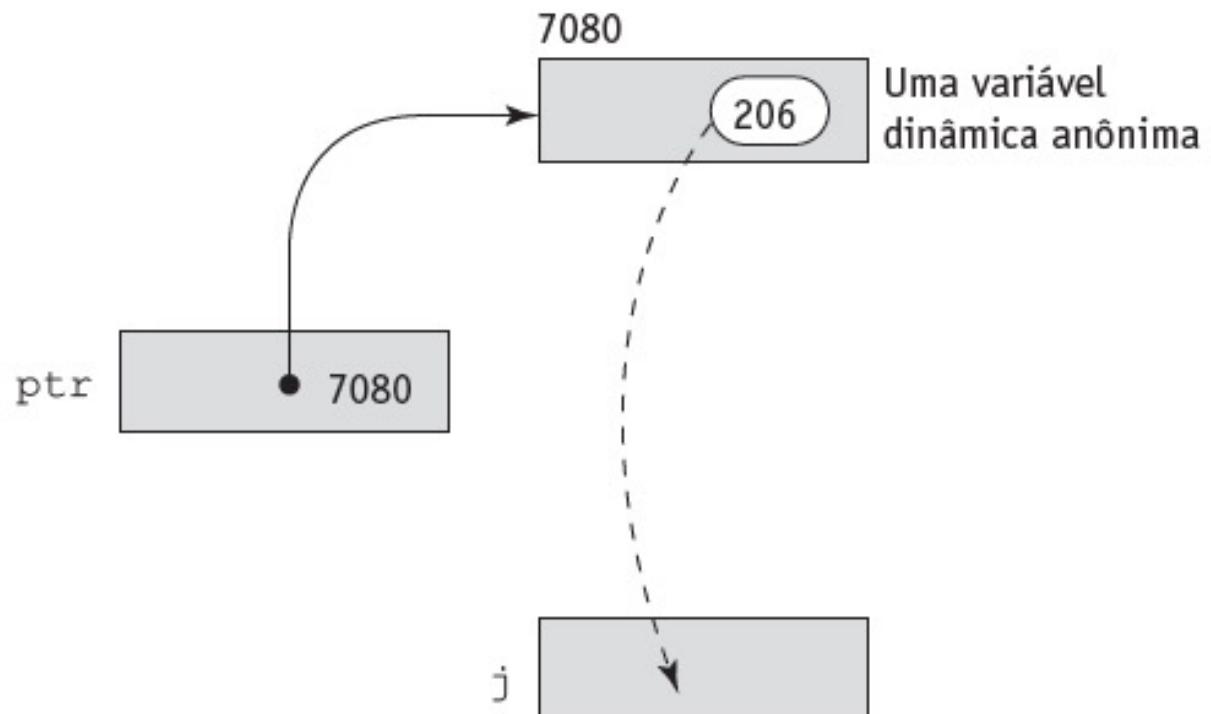


FIGURA 6.8

A operação de atribuição `j = *ptr.`

Ponteiros e Referências: Problemas com ponteiros

- A primeira linguagem de programação de alto nível a incluir variáveis do tipo ponteiro foi **PL/I**, na qual os ponteiros podiam ser usados para referenciar tanto variáveis dinâmicas do monte quanto outras variáveis de programa.
 - Eles eram **muito flexíveis**, mas seu uso podia levar a diversos tipos de **erros de programação**.
 - <https://www.ibm.com/docs/en/epfz/5.3?topic=expressions-pointer-operations>
 - Alguns dos problemas dos ponteiros em PL/I também estão presentes nas LPs subsequentes.
- Algumas LPs, como Java, substituíram completamente os ponteiros por **tipos de referência** que, com a **liberação implícita**, minimizam os principais problemas.
- Um tipo de referência é apenas um **ponteiro com operações restritas**.

Ponteiros e Referências: Problemas com ponteiros

1. Ponteiros soltos ou referências soltas (*dangling pointers*)
 - Contém o endereço de uma variável dinâmica do monte já liberada.
 - A posição sendo apontada pode ter sido realocada para alguma **variável nova**
 - Se a nova variável não for do mesmo tipo da antiga, as **verificações de tipos** dos usos do ponteiro solto são inválidas.
 - Se o ponteiro solto é usado para modificar a variável dinâmica do monte, o **valor da nova variável será destruído**.
 - Adicionalmente, é possível que a posição agora esteja sendo temporariamente usada pelo **sistema de gerenciamento de armazenamento** .

Ponteiros e Referências: Problemas com ponteiros

1. (continuação)

- A seguinte sequência de operações cria um ponteiro solto em muitas linguagens:
 1. Uma nova variável dinâmica do monte é criada e o ponteiro `p1` é configurado para apontar para ela.
 2. O ponteiro `p2` é atribuído como o valor de `p1`.
 3. A variável apontada por `p1` é explicitamente liberada (possivelmente configurando `p1` para `nil`), mas `p2` não é modificado pela operação.
 - `p2` é agora um ponteiro solto.
- Se a operação de liberação não modificar `p1`, tanto `p1` quanto `p2` seriam soltos.

Ponteiros e Referências: Problemas com ponteiros

- Por exemplo, em C++ teríamos:

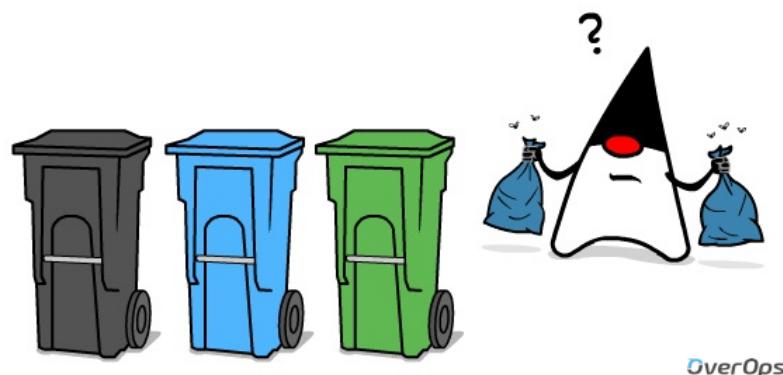
```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Agora, arrayPtr1 é solto, porque o armazenamento no monte
// para o qual ele estava apontando foi liberado.
```

- Em C++, tanto arrayPtr1 quanto arrayPtr2 são agora **ponteiros soltos**, porque o operador C++ **delete** não tem efeito no valor de seu ponteiro passado como operando.
- Em C++, é comum (e seguro) seguir um operador **delete** com uma atribuição de zero, que representa **null**, para o ponteiro cujo valor apontado tenha sido liberado.
- Note que a **liberação explícita** de variáveis dinâmicas é a causa dos ponteiros soltos.

Ponteiros e Referências: Problemas com ponteiros

2. Variáveis dinâmicas do monte perdidas

- Uma **variável dinâmica do monte perdida** é uma alocada que não está mais acessível para os programas de usuário.
- São frequentemente chamadas de **lixo**, pois não são úteis para seus propósitos originais e não podem ser realocadas para algum novo uso no programa.



OverOps

Ponteiros e Referências: Problemas com ponteiros

2. (continuação)

- Variáveis dinâmicas do monte perdidas são em geral criadas pela seguinte sequência de operações:
 - p1 é configurado para apontar para uma variável dinâmica do monte recém-criada.
 - p1 é posteriormente configurado para apontar para outra variável recém-criada.
- A primeira variável dinâmica do monte é agora **inacessível, ou perdida**.
- Isso às vezes é chamado de **vazamento de memória** e, independentemente da linguagem usar liberação implícita ou explícita, é um problema.

Ponteiros e Referências: Ponteiros em C e C++

- São extremamente flexível, mas devem ser usados com cuidado.
- Os ponteiros podem apontar para qualquer variável, independentemente de quando ou onde ela foi alocada.
- Usado para gerenciamento e endereçamento de armazenamento dinâmico.
- A aritmética de ponteiro é possível.
- Desreferenciação explícita e operadores de endereço.
- Tipo não precisam ser fixados (**void ***).

void * pode apontar para qualquer tipo e pode ser *type checked* (não pode ser desreferenciado)

Ponteiros e Referências: Ponteiros em C e C++

```
int list [10];  
int *ptr;  
ptr = list;
```

```
// *(ptr + 1) é equivalente a list[1]  
// *(ptr + index) é equivalente a list[index]  
// ptr[index] é equivalente a list[index]
```

Está claro, a partir dessas sentenças, que as operações de ponteiros incluem **a mesma escala usada em operações de indexação**.

Ponteiros e Referências: Ponteiros em C e C++

- Ponteiros em C e C++ podem apontar para funções.
- Esse recurso é usado para passar funções como parâmetros para outras funções.
 - Exemplo: <https://www.cplusplus.com/reference/cstdlib/qsort/>
- C e C++ incluem ponteiros do tipo **void ***, que podem apontar para valores de quaisquer tipos.
- Eles são, para todos os efeitos, **ponteiros genéricos**.
- Entretanto, a verificação de tipos não é um problema com ponteiros **void ***, porque essas linguagens não permitem desreferenciá-los.
- Um uso comum de ponteiros **void *** é como os tipos de parâmetros de funções que operam em memória.

Ponteiros e Referências: Tipos de referência

- Uma variável de **tipo de referência** é similar a um ponteiro, com uma diferença fundamental:
 - Um ponteiro se refere a um endereço em memória, enquanto uma referência, a **um objeto ou a um valor em memória**.
- C++ inclui uma forma especial de tipo de referência usada primariamente para os **parâmetros** formais em definições de funções.
 - Vantagens das passagens tanto por valor quanto por referência.
 - <https://www.javatpoint.com/call-by-value-and-call-by-reference-in-c>
 - <https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/>

Ponteiros e Referências: Tipos de referência

- Em C++, variáveis de tipo de referência são especificadas em definições ao preceder seus nomes com o sinal de e comercial (&). Por exemplo,

```
int result = 0;  
int &ref_result = result; ...  
ref_result = 100;
```

- Nesse segmento de código, result e ref_result são apelidos.

Ponteiros e Referências: Tipos de referência

- Em Java, variáveis de referência são estendidas da forma de C++ para uma que as permitem substituírem os ponteiros inteiramente.
- Buscando segurança aumentada em relação ao C++, os projetistas de Java **removeram ponteiros no estilo de C++** definitivamente.
- Diferentemente das variáveis de referência de C++, em Java elas podem ter atribuídas a elas diferentes **instâncias de classes**; ou seja, não são constantes.
- Todas as instâncias de classes em Java são referenciadas por **variáveis de referência**.

Ponteiros e Referências: Tipos de referência

- No código a seguir, String é uma classe padrão de Java:

```
String str1;  
...  
str1 = "This is a Java literal string";
```

- str1 é definida como uma referência a uma instância (ou objeto) da classe String.
- Ela é inicialmente configurada como null.
- A atribuição subsequente configura str1 para referenciar o objeto String,
“This is a Java literal string”.
- Como as instâncias de classe em Java são **implicitamente liberadas** (não existe um operador explícito de liberação), **não podem existir referências soltas em Java**.

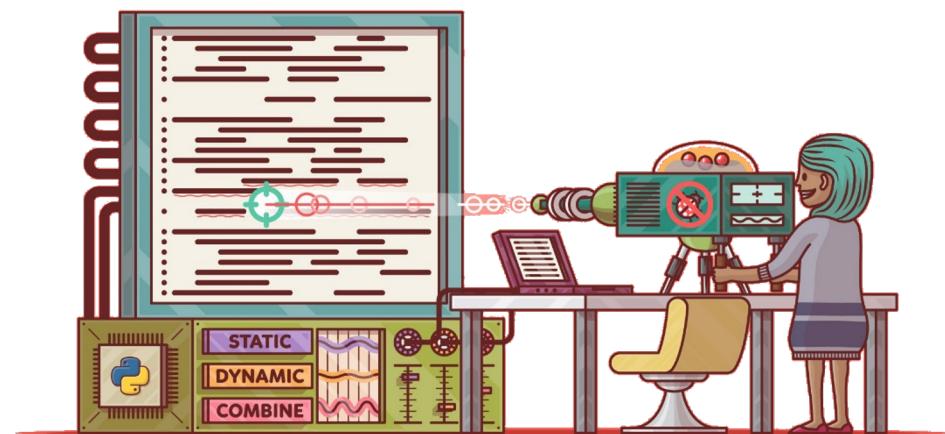
Ponteiros e Referências: Tipos de referência

- C# inclui tanto as referências de Java quanto os ponteiros de C++.
- Entretanto, o uso de ponteiros é fortemente desencorajado.
- Na verdade, quaisquer programas que usem ponteiros devem incluir o modificador **unsafe**.
 - <https://www.c-sharpcorner.com/UploadFile/f0b2ed/understanding-unsafe-code-in-C-Sharp/>
 - Note que, apesar dos objetos apontados por referências serem implicitamente liberados, isso não é verdade para objetos apontados por ponteiros.
- Os ponteiros foram incluídos em C# principalmente para permitir que os programas em C# interoperassem com código C e C++.

VERIFICAÇÃO DE TIPOS

Verificação de Tipos

- Na verificação de tipos, o conceito de operandos e operadores é generalizado para incluir subprogramas e sentenças de atribuição.
- Subprogramas serão pensados como operadores cujos operandos são seus parâmetros.
- O símbolo de atribuição será considerado como um operador binário, com sua variável-alvo e sua expressão sendo os operandos.



Real Python

Verificação de Tipos

- A **verificação de tipos** é a atividade de garantir que os operandos de um operador são de tipos compatíveis.
- Um tipo **compatível** ou é legal para o operador, ou é permitido a ele, dentro das regras da LP, ser implicitamente convertido pelo código gerado pelo compilador (ou pelo interpretador) para um tipo legal.
 - Essa conversão automática é chamada de **coerção**.
- Por exemplo, se uma variável **int** e uma variável **float** são adicionadas em Java, o valor da variável inteira sofre uma coerção para **float** e uma adição de ponto flutuante é realizada.

Verificação de Tipos

- Exemplos de conversão de tipos:
 - Java: <http://faculty.salina.k-state.edu/tmertz/Java/041datatypeoperators/07typecoercionandconversion.pdf>
 - C: https://www.tutorialspoint.com/cprogramming/c_type_casting.htm
- **Erro de tipo** é a aplicação de um operador a um operando de um tipo inapropriado.
 - Por exemplo, na versão original do C, se um valor **int** fosse passado para uma função que esperava um valor **float**, um erro de tipo ocorreria (pois os compiladores dessa LP não verificavam os tipos dos parâmetros).

Verificação de Tipos

- Se todas as vinculações de variáveis a tipos são estáticas na linguagem, a verificação de tipos pode ser feita praticamente sempre de maneira **estática** (ou seja, em tempo de compilação).
- A vinculação dinâmica de tipos requer a verificação de tipos em tempo de execução, chamada de **verificação de tipos dinâmica**.
- Algumas linguagens, como JavaScript e PHP, por causa de sua vinculação de tipos dinâmica, permitem apenas a verificação de tipos dinâmica.

Verificação de Tipos

- É **melhor** detectar erros em tempo de compilação do que em tempo de execução, pois a correção feita mais cedo é geralmente **menos custosa**.
- A penalidade para a verificação estática é uma **flexibilidade reduzida** para o programador.
- Menos **atalhos e truques** são permitidos.
- Tais técnicas, entretanto, são normalmente consideradas **não apropriadas**.



Verificação de Tipos

- A **verificação de tipos é complicada** quando uma LP permite que uma célula de memória armazene valores de **tipos diferentes em momentos diferentes da execução**.
 - Tais células de memória podem ser criadas com registros variantes em Ada, **Equivalence** em Fortran, e **uniões** de C e C++.
- Aqui, a verificação de tipos deve ser dinâmica e requer que o sistema **mantenha o tipo do valor atual** de tais células de memória.
- Então, apesar de todas as variáveis serem estaticamente vinculadas a tipos em linguagens como C++, **nem todos os erros de tipos podem ser detectados por verificação estática de tipos**.

TIPAGEM FORTE

Tipagem Forte

- Uma das ideias em projeto de LPs que se tornou proeminente na chamada revolução da programação estruturada da década de 1970 foi a **tipagem forte**, muito reconhecida como uma característica de linguagem altamente valiosa.
- Infelizmente, ela muitas vezes é **definida de maneira pouco rígida** ou mesmo usada na literatura em computação sem ser definida.
 - <https://stackoverflow.com/questions/2351190/static-dynamic-vs-strong-weak>
 - <https://stackoverflow.com/questions/2690544/what-is-the-difference-between-a-strongly-typed-language-and-a-statically-typed>

Tipagem Forte

- Uma linguagem de programação é **fortemente tipada** se erros de tipos são sempre detectados.
- Isso requer que os **tipos de todos os operandos possam ser determinados**, em tempo de compilação ou em tempo de execução.
- A importância da tipagem forte está na habilidade de **detectar usos incorretos de variáveis** que resultam em erros de tipo.
- Uma linguagem fortemente tipada **também permite a detecção, em tempo de execução**, de usos de valores de tipo incorretos em variáveis que podem armazenar valores de mais de um tipo.

Tipagem Forte

- O **Fortran 95** não é **fortemente tipado** porque o uso de **Equivalence** entre variáveis de tipos diferentes permite que uma variável de um tipo se refira a um valor de outro tipo, sem o sistema ser capaz de **verificar o tipo** do valor quando uma das variáveis equivalentes é **referenciada** ou tem valores a ela **atribuídos**.
 - <https://craftofcoding.wordpress.com/2020/01/14/fortran-re-engineering-equivalence-statements-i/>
- Na verdade, a verificação de tipos em variáveis com o uso de **Equivalence** eliminaria grande parte de sua utilidade.

Tipagem Forte

- Ada é quase fortemente tipada, pois permite aos programadores usarem brechas nas regras de verificação de tipos ao requererem especificamente que a verificação de tipos seja suspensa para uma conversão em particular.
 - Essa suspensão temporária de verificação de tipos pode ser feita apenas quando uma instanciação da função genérica `Unchecked_Conversion` é chamada.
 - Alguém obtém um valor de seu tipo de parâmetro e retorna a cadeia de bits que é o valor atual do parâmetro.
 - Nenhuma conversão real é realizada; é meramente uma forma de extrair o valor de uma variável de um tipo e usá-la como se fosse de um tipo diferente.
 - https://www.adaic.org/resources/add_content/docs/95style/html/sec_5/5-9-1.html
 - Tais funções podem ser instanciadas para qualquer par de subtipos.

Tipagem Forte

- Esse tipo de conversão é algumas vezes chamado de **conversão implícita sem conversão (*nonconverting cast*)**.
- Conversões não verificadas podem ser úteis para as operações de **alocação e de liberação** definidas pelo usuário, na qual endereços são manipulados como inteiros, mas devem ser usados como ponteiros.
 - <https://craftofcoding.wordpress.com/2018/03/02/coding-ada-pointers-the-extreme-basics/>
- Como nenhuma verificação é feita em **Unchecked_Conversion**, é **responsabilidade do programador** garantir que o uso de um valor obtido a partir dela tenha algum significado.

Tipagem Forte

- C e C++ não são linguagens **fortemente tipadas** porque ambas incluem tipos união não verificados em relação a tipos.
- ML é **fortemente tipada**, apesar de tipos de alguns parâmetros de funções poderem não ser conhecidos em tempo de compilação.
- Java e C#, apesar de serem baseadas em C++, são **fortemente tipadas no mesmo sentido de Ada**.
 - Os tipos podem ser convertidos explicitamente, resultando em um erro de tipos.
 - Entretanto, não existem maneiras implícitas pelas quais os erros de tipos possam passar despercebidos.

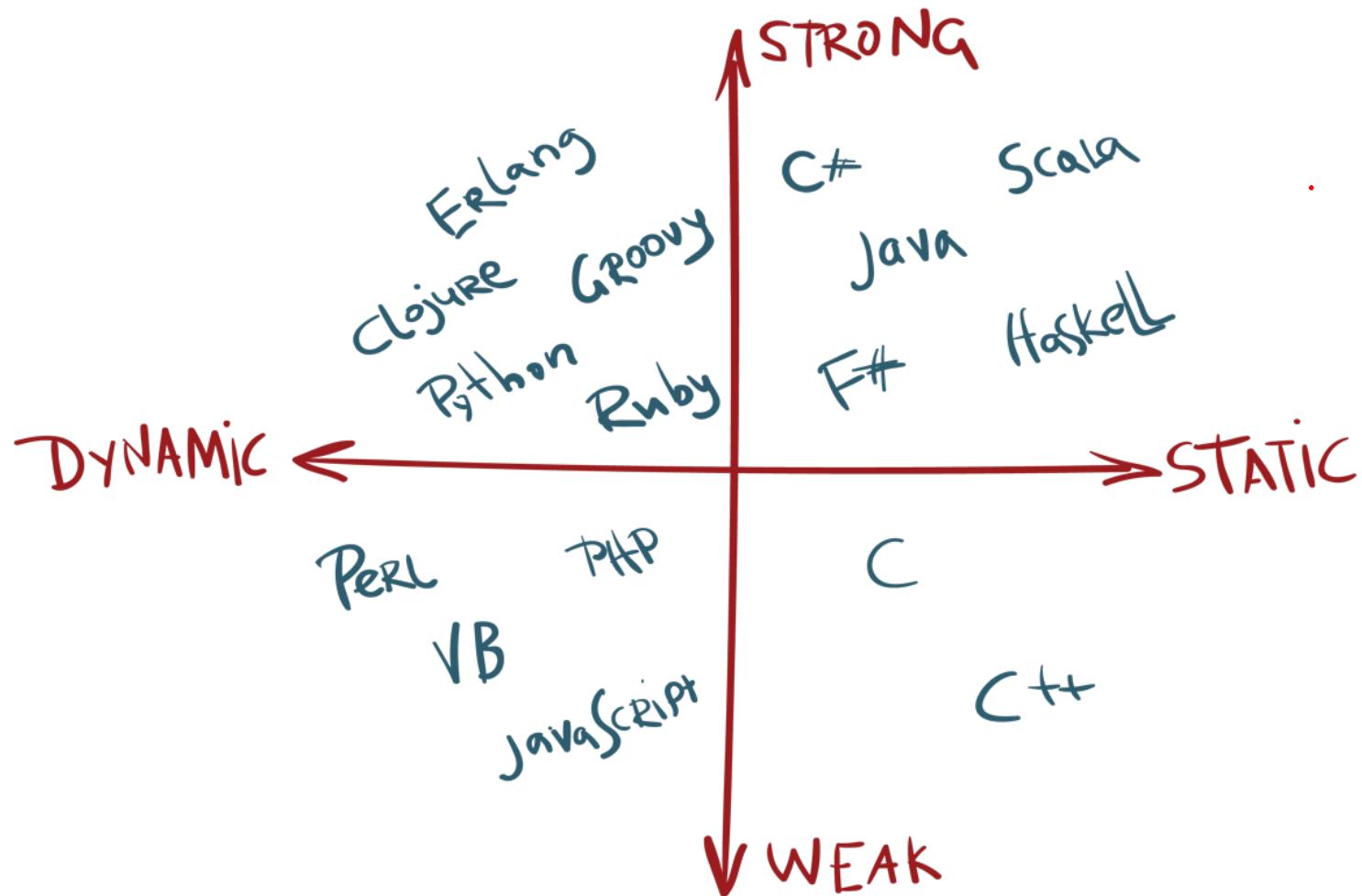
Tipagem Forte

- As **regras de coerção** de uma linguagem têm efeito importante no valor da verificação de tipos.
 - Por exemplo, as expressões são fortemente tipadas em Java.
 - Entretanto, um operador aritmético com um operando de ponto-flutuante e um operador inteiro é legal.
 - O valor do operando inteiro sofre uma coerção para ponto-flutuante, e uma operação de ponto-flutuante é realizada.
 - Isso é o que normalmente o programador pretende.
- Entretanto, a coerção também resulta em uma perda de um dos benefícios da tipagem forte – a **detecção de erros**.

Tipagem Forte

- Exemplo:
 - Suponha que um programa tem as variáveis inteiras (**int**) a e b e a variável de ponto-flutuante (**float**) chamada d.
 - Agora, se um programador queria digitar a + b, mas equivocadamente digitou a + d, o erro não seria detectado pelo compilador.
 - O valor de a simplesmente sofreria uma coerção para float.
 - Então, **o valor da tipagem forte é enfraquecido pela coerção**.
- Linguagens com muitas coerções, como Fortran, C e C++, são menos confiáveis do que linguagens com poucas coerções, como Ada.
 - Java e C# têm cerca de metade das coerções de tipo em atribuições que C++, então sua detecção de erros é melhor, mas ainda assim não está perto de ser tão efetiva quanto a de Ada.

Tipagem Forte



EQUIVALÊNCIA DE TIPOS

Equivalência de Tipos

- A ideia da compatibilidade de tipos foi definida quando ocorreu a introdução da questão da verificação de tipos.
- **As regras de compatibilidade ditam os tipos de operandos aceitáveis para cada um dos operadores e especificam os possíveis tipos de erros da linguagem.**
- As regras são chamadas “de compatibilidade” porque, em alguns casos, o tipo de um operando pode ser convertido implicitamente pelo compilador ou pelo sistema de tempo de execução para torná-lo aceitável ao operador.

Equivalência de Tipos

- As regras de compatibilidade de tipos **são simples e rígidas para os escalares pré-definidos.**
- Entretanto, nos casos dos **tipos estruturados, como matrizes e registros e alguns tipos definidos pelo usuário, as regras são mais complexas.**
 - A coerção desses tipos é rara, então a questão não é a compatibilidade, mas a equivalência.
 - Ou seja, **dois tipos são equivalentes se um operando de um em uma expressão é substituído por um de outro, sem coerção.**
- A equivalência de tipos é uma forma estrita da compatibilidade – **compatibilidade sem coerção.**

Equivalência de Tipos

- O **projeto das regras de equivalência de tipos** de uma linguagem é importante, porque **influencia o projeto dos tipos de dados** e das operações fornecidas para seus valores.
- Com os tipos discutidos aqui, existem poucas operações pré-definidas.
- Talvez o resultado mais importante de duas variáveis sendo de tipos equivalentes é que **uma pode ter seu valor atribuído para a outra**.
- Existem duas abordagens para definir equivalência de tipos:
 - Por nome
 - Por estrutura

Equivalência de Tipos

- A **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.
- A **equivalência de tipos por estrutura** significa que duas variáveis têm tipos equivalentes se seus tipos têm estruturas idênticas. Existem algumas variações dessas abordagens, e muitas linguagens usam combinações de ambas.

Equivalência de Tipos

Por nome

- A equivalência de tipos por nome é fácil de implementar, mas é mais restritiva.
- Em uma interpretação estrita, uma variável cujo tipo é uma subfaixa dos inteiros não seria equivalente a uma variável do tipo inteiro.
- Por exemplo, suponha que Ada usasse equivalência estrita de tipos por nome e considere o seguinte código:

```
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

Por nome

- Outro problema com a equivalência de tipos por nome surge quando um tipo estruturado ou um definido pelo usuário é passado entre subprogramas por meio de parâmetros.
 - **Tal tipo deve ser definido apenas uma vez, globalmente.**
 - Um subprograma não pode informar o tipo de tais parâmetros formais em termos locais. Esse era o caso com a versão original do Pascal.
- **Para usar a equivalência por nome, todos os tipos devem ter nomes.**
 - A maioria das LPs permite aos usuários definirem tipos anônimos – sem nomes.
 - Para que uma linguagem use equivalência de tipos por nome, o compilador deve nomeá-los (com nomes internos) implicitamente.

Equivalência de Tipos

Por estrutura

- A equivalência de tipos por estrutura é mais flexível do que a equivalência por nome, mas é mais difícil de ser implementada.
- Sob a equivalência por nome, apenas os nomes dos dois tipos precisam ser comparados para determiná-la.
- Sob a equivalência de tipos por estrutura, entretanto, as **estruturas inteiras dos dois tipos devem ser comparadas**.
- Essa comparação nem sempre é simples (considere uma estrutura de dados que refere ao seu próprio tipo, como uma lista encadeada).

Equivalência de Tipos

Por estrutura

- Outras questões também podem aparecer.
 - Por exemplo, dois tipos de registro (ou **struct**) são equivalentes se têm a **mesma estrutura, mas nomes de campos diferentes?**
 - Dois tipos de matrizes de uma dimensão em um programa Fortran ou Ada são equivalentes se têm o **mesmo tipo de elemento, mas faixas de índices de 0..10 e 1..11?**
 - Dois tipos de enumeração são equivalentes se têm o **mesmo número de componentes, também literais com nomes diferentes?**
- Outra dificuldade com a equivalência de tipos por estrutura é que ela **não permite diferenciar tipos com a mesma estrutura.**

Equivalência de Tipos

Por estrutura

- Por exemplo, considere as seguintes declarações em Ada:

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

- Os tipos das variáveis desses dois tipos são considerados **equivalentes sob a equivalência de tipos por estrutura**, permitindo que sejam misturados em expressões, o que é claramente indesejável nesse caso, considerando a diferença indicada pelos nomes dos tipos.
- De um modo geral, **tipos com nomes diferentes** provavelmente são abstrações de diferentes categorias de valores de problemas e **não devem ser considerados equivalentes**.

Equivalência de Tipos

Por estrutura

- Ada usa uma forma restritiva de equivalência de tipos por nome, mas fornece duas construções de tipos – **subtipos e tipos derivados** – que evitam os problemas associados com a equivalência por nome.
- Um **tipo derivado** é um novo tipo baseado em algum previamente definido com o qual ele não é equivalente, apesar de terem estrutura idêntica.
- Os tipos derivados herdam todas as propriedades de seus ancestrais.

Equivalência de Tipos

Por estrutura

- Considere o seguinte exemplo:

```
type Celsius is new Float;
```

```
type Fahrenheit is new Float;
```

- Os tipos de variáveis desses dois tipos derivados **não são equivalentes**, apesar de suas estruturas serem idênticas.
- Além disso, variáveis de ambos os tipos não são equivalentes aos de nenhum outro tipo de ponto-flutuante.
- Literais são uma exceção a essa regra. Um literal como 3.0 tem o tipo real e é equivalente a qualquer tipo de ponto flutuante.
- Tipos derivados podem incluir também restrições de faixa nos tipos ancestrais, enquanto ainda assim herda todas as operações do ancestral.

Equivalência de Tipos

Por estrutura

- Um **subtipo** em Ada é uma versão possivelmente reduzida em relação à faixa de um tipo existente.
- Um subtipo tem equivalência com seu ancestral.
- Por exemplo, considere a declaração:

```
subtype Small_type is Integer range 0..99;
```

- O tipo **Small_type** é equivalente ao **Integer**.

RESUMO

Resumo

Os tipos de dados de uma linguagem são uma grande parte do que determina seu estilo e sua utilidade. Com as estruturas de controle, eles formam o coração de uma linguagem.

Os **tipos de dados primitivos** da maioria das linguagens imperativas incluem os tipos numéricos, de caracteres e booleanos. Os tipos numéricos, em geral, são diretamente suportados por hardware.

Os **tipos de enumeração e de subfaixa definidos pelo usuário** são convenientes e melhoram a legibilidade e a confiabilidade dos programas.

Os **registros** são agora incluídos na maioria das linguagens. Campos de registros são especificados de diversas maneiras. No caso do COBOL, podem ser referenciados sem nomear todos os registros que os envolvem, apesar disso ser confuso de implementar e de prejudicar a legibilidade. Em diversas linguagens que suportam a programação orientada a objetos, os registros são suportados com objetos.

Uniões são estruturas que podem armazenar valores de tipo diferentes em momentos diferentes. Uniões discriminadas incluem uma etiqueta para gravar o valor de tipo atual. Uma união livre é uma união que não tem essa etiqueta. A maioria das linguagens com uniões não tem projetos seguros para elas, com exceção da linguagem Ada.

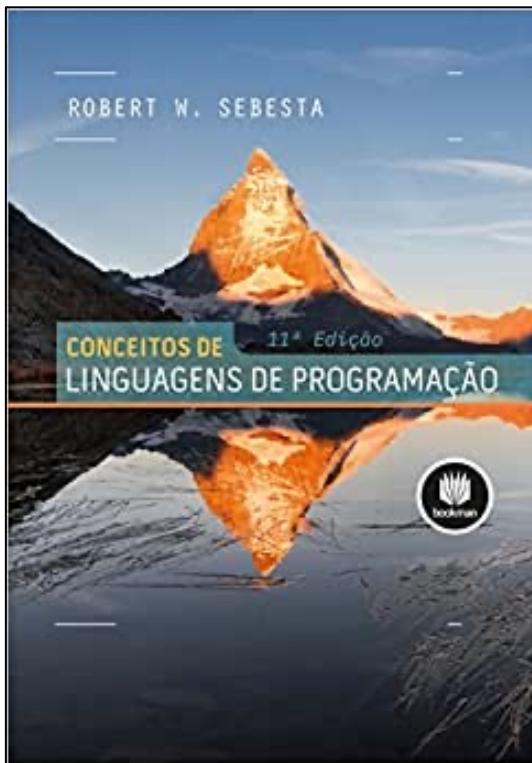
Ponteiros são usados para lidar com a flexibilidade e para controlar o gerenciamento de armazenamento dinâmico. Os ponteiros apresentam alguns perigos inerentes: ponteiros soltos são difíceis de ser evitados, e podem ocorrer vazamentos de memória.

Tipos de referência, como os de Java e C#, fornecem gerenciamento do monte sem os perigos de ponteiros.

A **tipagem forte** é o conceito de requerer que todos os erros de tipos sejam detectados. O valor da tipagem forte é um aumento na confiabilidade.

As **regras de equivalência de tipos** determinam quais operações são legais dentre os tipos estruturados de uma linguagem. As equivalências de tipos por nome e por estrutura são as duas abordagens fundamentais para definir equivalência de tipos.

Referência Bibliográfica



- SEBESTA, Robert W.; SANTOS, José Carlos Barbosa dos; TORTELLO, João Eduardo Nóbrega, Conceitos de linguagens de programação. 11 ed. Porto Alegre, RS: Editora Bookman, 2018, 758 p. ISBN 978-85-8260-468-7.
- <http://www2.fct.unesp.br/docentes/dmec/olivete/lp/arquivos/Aula7.pdf>