



# Paradigma Imperativo

## Variáveis e Tipos de Dados



A decorative pattern of hexagons in various shades of blue and cyan, some solid and some outlined, arranged in a cluster on the left side of the slide. A small icon of a network or molecule is also visible near the top left.

1

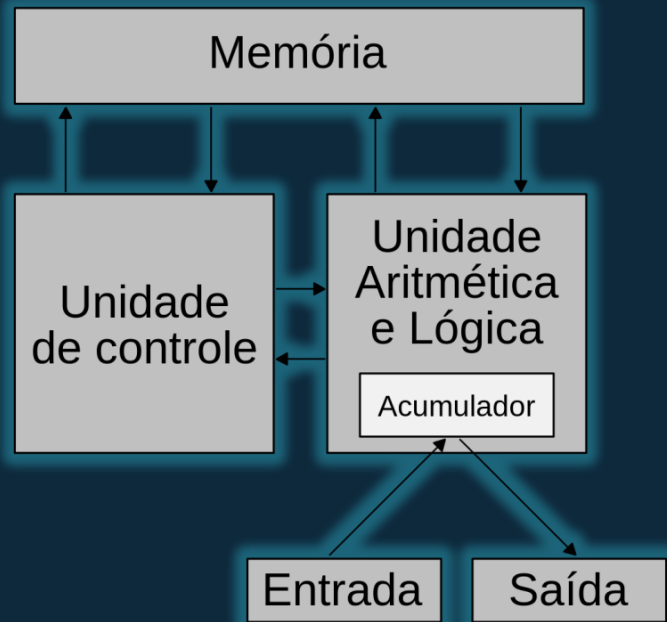
# Variáveis

# Influência

Paradigma imperativo é baseado na arquitetura de Von Neumann.

Regiões da memória são utilizadas para armazenar valores e recebem o nome de variáveis.

O espaço ocupado depende do tipo e especificação da variável utilizada.





# Nomes

Entre as questões envolvidas com nomes de variáveis em LPs, encontram-se:

- comprimento do nome
- caracteres permitidos (ex: \$, \_, #, etc.)
- distinção entre maiúsculas e minúsculas (ex: parseInt)
- existência de palavras reservadas ou palavras chave
- possibilidade de renomeação



# Palavras chave em Fortran

Aqui, a variável chama Integer e recebe o valor 4

Integer X  
Integer = 4  
Qual a diferença?

Aqui define-se uma variável chamada X, do tipo Integer

- ◇ Integer, em Fortran, é uma palavra chave, pois, se encontrada no início de uma sentença, seguida de um nome, então é o tipo da variável X
- ◇ Se seguida de operador de atribuição, é nome de variável





# Atributos

Variável é uma abstração de uma ou mais células de memória de um computador e pode ser caracterizada pelos atributos (descriptor da variável):

- Nome
- Endereço
- Valor
- Tipo
- Tempo de vida
- Escopo





# Nomes: tamanhos

Nas primeiras LPs, variáveis tinham apenas uma letra para o nome. Algumas letras indicavam ponto flutuante (ex: X e Y) e outras indicavam inteiros (ex: I e J). Ex. de comprimentos máximos de nomes:

- FORTRAN I: máximo 6 caracteres
- COBOL: máximo 30 caracteres
- FORTRAN: 90 e ANSI C: máximo 31 caracteres
- Ada, Java e C#: não tem limite e todos são significativos
- C++: não tem limite, mas as implementações sim.





# Vinculação

Vinculação de variáveis diz respeito à atribuição de endereços (e consequentemente valores) e principalmente ao tipo de variável utilizada.

A vinculação pode ocorrer em diferentes momentos: projeto e implementação da linguagem, compilação de código ou em momento de execução, por exemplo







# Tempo de vinculação

```
int count = 1;
```

**Tempo de projeto/implementação da linguagem:** tipos possíveis para `count`, faixa de valores para `int`, significado da operação “=”, representação de 1, etc.

**Tempo de compilação:** tipo atribuído a `count`.

**Tempo de execução:** valor e endereço de `count`.





# Exemplos em Python

- ◇ `my_string = 'Hello, World!'`
- ◇ `myflt = 45.06`
- ◇ `my_bool = 5 > 9`
- ◇ `my_list = ['item_1', 'item_2', 'item_3', 'item_4']`
- ◇ `my_tuple = ('one', 'two', 'three')`
- ◇ `my_dict = {'letter': 'g', 'number': 7, 'symbol': '&'}`





# Formas de vinculação

As variáveis podem ser vinculadas a um tipo por meio de:

- ***Vinculação estática***: ocorre antes do tempo de execução e permanece inalterada durante a execução em si
  - **Declaração explícita**: tipos e nomes das variáveis são declarados explicitamente
  - **Declaração implícita**: tipo da variável não é especificado, sendo definido por convenções da LP ou por inferência de tipos.
- ***Vinculação dinâmica***: vinculação ocorre ou é alterada durante execução do código.





# Vinculação dinâmica

A vinculação dinâmica aumenta a flexibilidade de programação, permitindo a criação de subprogramas genéricos (independente do tipo de dados).

Possui como desvantagens a diminuição da capacidade do compilador detectar erros e custo elevado para implementação, por conta da interpretação de tipo.



# Vinculação dinâmica

Em Python

```
list = [10, 20, 30, 50, 100]
print(list)
list = 25
print(list)
list = {"a": 1, "b": 2}
print(list)
```

A variável `list` recebe uma lista de inteiro, depois, a mesma variável recebe um valor inteiro (25), posteriormente, a mesma variável recebe um dicionário. A cada nova atribuição, a variável muda de tipo.



# Verificação de tipos

**Linguagem fortemente tipificada:** é uma linguagem na qual é possível detectar todos os erros de tipos durante o processo de compilação.

Quando duas variáveis são de **tipos compatíveis**, qualquer uma delas pode ter seu valor atribuído à outra (com alguma perda ou ajuste, obviamente). Na maior parte das LPs os tipos numéricos são compatíveis entre si.





# Inferência de tipos

Os tipos das variáveis são inferidos sem a necessidade do programador especificar o tipo

Pode ocorrer em linguagens com tipagem forte ou fraca.

*Em linguagens de tipagem fraca, como PHP e Javascript, alguns cuidados são necessários para evitar anomalias, como somar um inteiro com uma string...*





# Vinculação em memória

**Variáveis Estáticas:** são vinculadas a células de memória antes do início de execução do programa, e permanecem associadas às mesmas células até o programa terminar.

**Variáveis Pilhas-Dinâmicas:** a associação é efetuada em tempo de execução, em uma estrutura organizada como pilha, sendo utilizada principalmente para variáveis de métodos.








# Vinculação em memória

**Variáveis Heap-Dinâmicas Explícitas:** as variáveis são reservadas e liberadas da memória em tempo de execução por declarações explícitas do programador (ex: ponteiros, objetos em Java).

```
int *intnode;  
intnode = new int;  
...  
delete intnode;
```



Ponteiro para um inteiro, criado e deletado explicitamente, por comandos do programador






# Vinculação em memória

**Variáveis Heap-Dinâmicas Implícitas:** são reservadas no momento em que lhe são atribuídos valores e liberadas por meio de instruções explícitas. Por vezes são nomes que se adaptam a diferentes tipos de variáveis, como variáveis em JavaScript e Python.

```
list = 10
```

```
list = [20,30,40,50]
```



O mesmo nome de variável é utilizado primeiramente para um inteiro e depois para uma lista de valores





# Escopo

Escopo determina os locais em que uma variável é válida e pode ser enxergada/acessada. Em geral, por exemplo, métodos definem variáveis locais, acessíveis apenas dentro dos métodos.





# Escopo

- Blocos definem escopo?
- É possível utilizar um nome já utilizado em um escopo mais abrangente?
- Se a LP possibilita reutilizar nomes, é possível acessar o escopo externo? (ex: C/C++, Pascal e Ada permitem esse acesso).





# Exemplo

```
#include <iostream>
using namespace
std;
int x = 0;
void foo() {
    x++;
}
void bar() {
    int x = 10;
    foo();
}
```

```
int main() {
    bar();
    cout << x << endl;
    return 0;
}
```

**Qual o valor impresso?**





# Exemplo

```
#include <iostream>
using namespace std;
int x = 0;
void foo() {
    x++;
}
void bar() {
    int x = 10;
    foo();
    cout << x << endl;
}
```

```
int main() {
    bar();
    cout << x << endl;
    return 0;
}
```

**E agora?**





# Escopo Global

- ◇ Algumas linguagens permitem a estruturação de programas como uma sequência de funções.
- ◇ As variáveis podem ser definidas dentro ou fora das funções.
- ◇ As definições externas às funções são consideradas globais



# Exemplo em PHP

```
$day = "Monday";  
$month = "January";
```

```
function calendar(){  
    $day = "Tuesday";  
    print "Today is $day<br>";  
    $gday = $GLOBALS['day'];  
    global $month;  
    print "Global today is $gday<br>";  
    print "Global month is $month<br>";  
}  
calendar();
```

Exemplo de escopo de variáveis em PHP. As variáveis \$day e \$month, fora da função calendar são distintas das variáveis com mesmo nome dentro da função.

Acesso à variável \$day de fora da função.

Torna a variável \$month, de fora da função, acessível de dentro da função.





# Exemplo em Python

```
def f():  
    x = 300  
    def inner_f():  
        print(x)  
    inner_f()
```

f()

Variáveis estão acessíveis em subfunções



# Exemplo em Python

```
x = 300
```

```
def myfunc():  
    x = 200  
    print(x)
```

```
myfunc()
```

```
print(x)
```

Variável global

Variável interna à função myfunc



A cluster of hexagons in various shades of blue and cyan, with some having white outlines, arranged in a honeycomb-like pattern in the top-left corner.

# Escopo e tempo de vida

Apesar do senso comum em contrário, escopo e tempo de vida não estão diretamente ligados.





# Exemplo

```
#include <iostream>
using namespace std;

int foo() {
    static int y =
0;
    y++;
    return y;
}
```

```
int main() {
    int x;
    foo();
    foo();
    x = foo();
    cout << x << endl;
    return 0;
}
```

x e y possuem tempo de vida diferentes dos respectivos escopos!





# Ambientes de referenciamento

- ◇ Ambientes de referenciamento de uma sentença é o conjunto de todas as variáveis visíveis na sentença
- ◇ Alguns editores de código trazem esse conjunto no recurso de *autocomplete*



A decorative pattern of hexagons in various shades of blue and cyan, some solid and some outlined, arranged in a cluster on the left side of the slide. A small icon of a network or molecule is also visible near the top left.

2

## Tipos de Dados

A cluster of several hexagons in various shades of blue and cyan, some solid and some outlined, arranged in a geometric pattern in the top-left corner.

# Tipos de dados

Apesar do senso comum em contrário, escopo e tempo de vida não estão diretamente ligados.





# Tipos primitivos: inteiros

Representa valores numéricos, negativos ou positivos, sem casas decimais. Muitas vezes é mero reflexo do hardware.

O tamanho do conjunto que pode ser representado, em geral, depende da máquina em que o programa é executado.

Algumas linguagens o subdividem de acordo com o espaço necessário para o armazenamento de um determinado valor (byte, short, int, long).







# Tipos primitivos: inteiros

Tipo	Tamanho (bits)	Intervalo	
		Início	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





# Ponto Flutuante

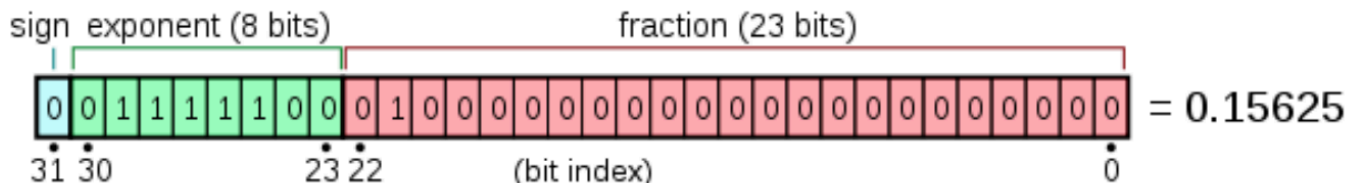
Representa valores numéricos, negativos ou positivos, com casas decimais. Pode também ser chamado incorretamente de “real” (mas não é capaz de permitir a representação de números reais).

Linguagens para uso científico, que requerem maior precisão, suportam, em geral, pelo menos dois tipos de dados em ponto flutuante (*float*, *double*, por exemplo).

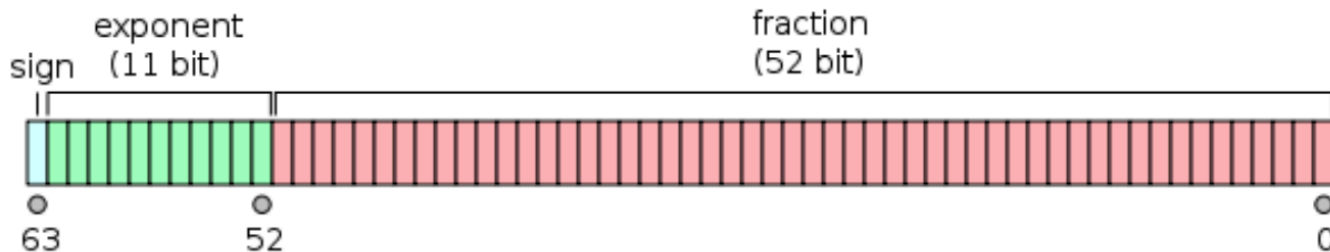


# Ponto Flutuante

Representação de acordo com o padrão IEEE para variáveis desse tipo:



Padrão IEEE 754 (Precisão simples)



Padrão IEEE 754 (Precisão Dupla)





# Números complexos

Suportado como dado primitivo por algumas linguagens, como Pascal, Fortran e Python.

Implementado geralmente como pares ordenados de valores reais. Ex:

```
COMPLEX myVar
```

```
myVar = (1.0, -5.0)
```

myVar recebe o valor  $1 - 5i$

Em Python é possível usar  $1 - 5j$  diretamente.



# Números complexos

🏠 Python Reference (The Right Way)

latest

Introduction

Definitions

Coding Guidelines

Fundamental Data Types

Built-In Functions

Comprehensions and Generator Expression

Container Data Access

Operators

Statements

Other Objects

Double Underscore Methods and Variables

Exceptions

Constants

[Docs](#) » complex

[Edit on GitHub](#)

## complex

Complex numbers are an extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written  $i$  in mathematics or  $j$  in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a  $j$  suffix, e.g.,  $3+1j$ . To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

Complex numbers are stored as a pair of machine-level double precision floating point numbers.

## Constructors

### `complex()`

Returns an expression converted into a complex number.

### literal syntax

Initializes a new instance of the `complex` type.



# Decimal

Suportado por linguagens como COBOL, C# e SQL.

Faixa de valores representados restrita.

Mais precisos, porém com maior gasto de memória.

Representação através de BCD (*Binary-Coded Decimal*).





# Lógico

Representa apenas dois valores: “verdadeiro” ou “falso”. Não é suportado em todas as linguagens de uso geral (ex: C não suporta, C++ sim).

Poderia ser representado com apenas um bit (0 e 1), mas em geral, as linguagens de programação usam 1 byte.

Maior gasto de memória, porém maior legibilidade.





# Caracter

Símbolos e letras armazenados através de representação numérica. Codificação mais conhecida: ASCII (*American Standard Code for Information Interchange*)

Codificações mais atuais: Unicode => representação de qualquer tipo de texto. Pode ser implementado por meio de 16 bits (UCS-2) ou 32 bits (UCS-4). Forma mais utilizada é UTF, que permite utilizar caracteres de tamanhos diferentes (1 a 4 bytes).







# String

São considerados tipos primitivos em algumas linguagens (Fortran, Python), em outras são derivados (vetores) de tipos de caracteres (C, C++).

Questão importante: **como controlar o tamanho?**





# Cadeia de caracteres

Em geral o tamanho de cadeias de caracteres é controlado de três maneiras:

- tamanho estático pré-definido - sem possibilidade de redimensionamento (ex: SQL, Pascal)
- tamanho dinâmico utilizando controle de tamanho - permite redimensionamento, mas exige controle (ex: Pascal)
- tamanho dinâmico utilizando um caracter de término (ex: C/C++, Object Pascal)





# Enumeráveis

Um tipo enumerável é um tipo ordinal definido pelo usuário e que em várias linguagens é associado a um tipo inteiro.

Em geral, é utilizado para suportar constantes enumeráveis, como cores ou dias da semana.

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

**Há melhoras de legibilidade e confiabilidade.**





# Subfaixas

Uma subfaixa, também um tipo enumerável, é uma subsequência contígua de um tipo ordinal. Exemplos em Ada:

```
type Days is (mon, tue, wed, thu, fri, sat, sun);  
subtype Weekdays is Days range mon..fri;  
subtype Index is Integer range 1..100;
```

Há melhoras de legibilidade e, principalmente, confiabilidade.





# Arranjos, Vetores, Matrizes

Um arranjo (vetores/*arrays* ou matrizes) é uma coleção homogênea de elementos, em que cada elemento é identificado pela sua posição relativa ao primeiro elemento do agregado.

Os elementos individuais de um arranjo são de mesmo tipo.

Questões mais importantes: 1) quais os tipos permitidos para índices?  
2) os índices são verificados? 3) existe inicialização dos dados?





# Listas

Uma lista é uma coleção expansível de itens, homogêneos ou não, de acordo com a LP. Ex: em Lisp as listas podem misturar elementos de diferentes tipos, em Haskell, os itens são homogêneos.

São um recurso poderoso para programação funcional e lógica.

Em geral, listas podem ser aninhadas nas LPs que as suportam. Quando são suportadas, em geral a LP não oferece arranjos.





# Hashes

Vetor associativo é uma coleção não ordenada de elementos que são indexados por um conjunto de chaves. São implementados geralmente por meio de tabelas hash.

Em Python é disponibilizado como dicionários e pode ser utilizado para simular variáveis heterogêneas (registros).

Ex:

```
>>> my_dict = {'a' : 'one', 'b' : 'two'}  
>>> my_dict['a']  
'one'
```





# Registros, Estruturas, Classes

Em várias LPs, há mecanismos para definição de tipos de variáveis que são conjuntos heterogêneos de dados. Isso é feito geralmente por meio de registros ou, mais recentemente, classes. Ex:

```
struct ponto {  
    int x;  
    int y;  
};
```







# Ponteiros

Um tipo ponteiro indica variáveis que têm uma faixa de valores que consiste em endereços de memória e um valor especial nulo (*null*, *nil*, *None*, ...), que indica a ausência de referência

- ◆ Fornecem uma forma de gerenciar armazenamento dinâmico
- ◆ Pode ser usado para acessar posição onde o armazenamento é alocado dinamicamente (heap)





# Ponteiros: problemas

- ◇ Ponteiros soltos: indicam posição de memória já liberada (inconsistência)
- ◇ Variáveis da *heap* perdidas: variável alocada que não está mais acessível e não pode ser realocada para novo uso no programa (lixo e vazamento de memória)

A LP consegue identificar e resolver esses problemas?





# Referências

- ◇ Semelhantes aos ponteiros, mas ao invés de apontar para endereço de memória, apontam para um objeto ou valor em memória

```
String str1;
```

```
...
```

```
str1 = "String literal"
```

*str referencia uma instância da classe padrão String, em java*





# Tipos de dados em Python 3

## ◇ Principais

- `str`
- `int`, `float`, `complex`
- `list`, `tuple`, `range`
- `dict`
- `set`
- `bool`
- `bytes`





# Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.  
Capítulos 5 e 6

