


CLP – Aula 1

PARADIGMA IMPERATIVO

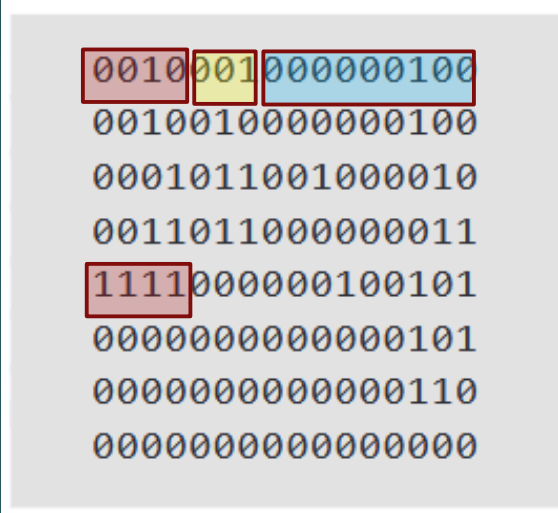
- 
- ▶ Conhecer os princípios das linguagens de programação é fundamental para entender **como as linguagens afetam a resolução de problemas e o pensamento computacional.**

As Origens das Linguagens de Programação

- ▶ Inicialmente, a “programação” envolvia a configuração física ("hardwiring") dos computadores.
- ▶ Um avanço significativo - ideia de **von Neumann em armazenar o programa na memória do computador** como uma sequência de códigos binários, que organizavam as operações básicas do *hardware* para resolver problemas específicos.
- ▶ Isso eliminou a necessidade de reestruturar fisicamente o *hardware* para cada nova tarefa.
- ▶ A **linguagem de máquina** foi a primeira forma de programação, diretamente compreendida pelo computador.
- ▶ Posteriormente, surgiu a **linguagem *assembly***, uma abstração mais legível da linguagem de máquina, utilizando mnemônicos para representar as instruções.
- ▶ No entanto, a linguagem *assembly* ainda carecia das capacidades de abstração mais poderosas de notações matemáticas convencionais.

Von Neumann – componentes do computador permanentemente conectados (final dos 1940s)

- ▶ Na memória são inseridos os dados e as instruções do programa (programa armazenado)
- ▶ O operador insere uma série de códigos binários que organizam as operações básicas de hardware para resolver problemas específicos.
- ▶ Em vez de desligar o computador para reconfigurar seus circuitos, o operador poderia acionar interruptores para inserir esses códigos, expressos em linguagem de máquina, na memória do computador.
- ▶ Nesse momento, os operadores de computador se tornaram os primeiros programadores.



0010001000000100
0010010000000100
0001011001000010
0011011000000011
1111000000100101
0000000000000101
0000000000000110
0000000000000000

Grande Problema – Traduzir abstrações dos problemas matemáticos em “pensamento da máquina”

Assembly

0010001000000100

0010010000000100

0001011001000010

0011011000000011

1111000000100101

0000000000000101

0000000000000110

0000000000000000

```
.ORIG x3000      ; Address (in hexadecimal) of the first instruction
LD R1, FIRST     ; Copy the number in memory location FIRST to register R1
LD R2, SECOND     ; Copy the number in memory location SECOND to register R2
ADD R3, R2, R1    ; Add the numbers in R1 and R2 and place the sum in
                  ; register R3
ST R3, SUM        ; Copy the number in R3 to memory location SUM
HALT              ; Halt the program
FIRST .FILL #5    ; Location FIRST contains decimal 5
SECOND .FILL #6   ; Location SECOND contains decimal 6
SUM .BLKW #1      ; Location SUM (contains 0 by default)
.END              ; End of program
```

Problemas do Assembly

- ▶ Não é boa em abstração
- ▶ Uma abstração é uma notação ou forma de expressar ideias, que as torna reduzidas, simples e fáceis para a mente humana entender.
- ▶ O filósofo/matemático A. N. Whitehead (1911) -> o poder da notação abstrata: *“Alivia o cérebro de todo trabalho desnecessário, uma boa notação o liberta para se concentrar em problemas mais avançados. A civilização avança ao estender o número de operações importantes que podemos executar sem pensar nelas.”*
- ▶ No caso da linguagem assembly, o programador tem que traduzir as ideias abstratas de um domínio de problema para a notação dependente de máquina de um programa.
- ▶ Outro problema -> Portabilidade – cada máquina tem seu próprio dialeto assembly

FORTRAN (FORmula TRANslation)

- ▶ Ao contrário do assembly, linguagens como C, Java e Python, suportam notações mais próximas das abstrações, como expressões algébricas, usadas em matemática e ciências.
- ▶ Por exemplo, o seguinte segmento de código em C é equivalente ao programa em assembly para adicionar dois números mostrado anteriormente:
 - ▶ *int first = 5; int second = 6; int sum = first + second;*
- ▶ O **FORTRAN** - John Backus - início dos 1950 - específico para um computador IBM. Inicialmente muito próximo a arquitetura de máquina - não tinha as instruções de controle estruturadas e estruturas de dados de linguagens de alto nível.
- ▶ Mas evoluiu continuamente – a última revisão é de 2023 e contém características de orientação a objeto.

Abstrações em Linguagens de Programação

- Uma **abstração** é uma notação - uma maneira de expressar ideias que as torna concisas, simples e fáceis de entender.
- **Abstração de Dados** - criação de entidades de dados de nível superior a partir dos bits brutos: inteiros, ponto flutuante e caracteres.
- Um exemplo de abstração de dados em unidade é o mecanismo de **classe** em linguagens orientadas a objetos.
- A “**reusabilidade**” é uma propriedade importante das abstrações de dados.

Abstrações em Linguagens de Programação

- ▶ **Abstração de Unidade:** Coleções de procedimentos que fornecem serviços logicamente relacionados a outras partes de um programa, formando uma unidade independente.
- ▶ **Abstração de Controle:** Refere-se a mecanismos que abstraem o **fluxo de controle**, como
 - ▶ “*if*” e “*for*” em linguagens de alto nível. Estas representam uma melhoria em relação às instruções de desvio (*branch*) da linguagem *assembly*.
 - ▶ **Procedimentos e funções** também são abstrações de controle, agrupando serviços logicamente relacionados.
 - ▶ A **recursão** oferece uma capacidade adicional de abstração para definições de funções. Em linguagens funcionais é muito usada

Abstrações em Linguagens de Programação

- **Abstração de Unidade**

- ▶ Se refere a organização dentro de um programa - agrupamos funcionalidades relacionadas dentro de uma única estrutura.
- ▶ Essa estrutura encapsula dados e operações, permitindo que sejam usados de maneira independente pelo restante do sistema.
- ▶ "Coleções de procedimentos" - "procedimentos" - significa funções ou métodos que executam operações específicas.
- ▶ Em muitas LPs, agrupamos funções relacionadas dentro de módulos, classes ou bibliotecas para manter o código organizado e reutilizável.
- ▶ A Abstração de Unidade ajuda a organizar código de forma modular, reutilizável e independente. Isso facilita a manutenção e melhora a legibilidade do programa.

O que são paradigmas de programação?

- ▶ São **abordagens ou estilos** para a construção de software, cada um com princípios, regras e formas específicas de estruturar código.
- ▶ Diferentes paradigmas oferecem vantagens dependendo do problema a ser resolvido.
- ▶ Os principais paradigmas são:
 - **Paradigma Imperativo** (incluindo Procedural e Orientado a Objetos)
 - **Paradigma Funcional**
 - **Paradigma Lógico**
 - **Paradigma Concorrente e Reativo**
- ▶ Cada paradigma tem suas características, benefícios e desafios, e muitos sistemas modernos combinam múltiplos paradigmas para obter o melhor de cada um.

Paradigmas Computacionais

- ▶ As primeiras LPs imitavam e abstraíam as operações de um computador, principalmente o **modelo de von Neumann**,
 - ▶ com uma unidade central de processamento executando sequencialmente instruções sobre valores armazenados na memória.
- ▶ Linguagens **imperativas** são típicas desse modelo, onde **variáveis** representam locais de memória e a computação ocorre através da **modificação do estado** da memória por meio de atribuições.
- ▶ Há outros paradigmas computacionais derivados da matemática: o **paradigma funcional**, baseado na noção de função do cálculo lambda, e o **paradigma lógico**, baseado na lógica simbólica. Estes serão detalhados mais tarde.

Paradigma **Imperativo**

- ▶ Define um programa como uma sequência de comandos que alteram o estado do sistema. Baseado no conceito de máquinas de estado, onde as **variáveis representam a memória** do sistema e as **instruções modificam essa memória** ao longo do tempo.
- ▶ Subtipos (inclusões ao longo da evolução):
 - ▶ **Programação Procedural** - Baseada no conceito de procedimentos (funções/sub-rotinas) para modularizar o código. Exemplo de linguagens: C, Pascal, Fortran. Princípios: Uso de variáveis globais e locais.
 - ▶ **Programação Estruturada** - Controle do fluxo do programa com laços (for, while) e estruturas condicionais (if, switch). Separação do código em funções para reutilização e melhor legibilidade.

Exemplo imperativo - procedural

```
#include <stdio.h>

void saudacao() {
    printf("Olá, mundo!\n");
}

int main() {
    saudacao();
    return 0;
}
```


Imperativo - **Programação Orientada a Objetos (POO)**

- Extensão do paradigma imperativo, baseada no conceito de **objetos** que encapsulam dados e comportamentos.
- Exemplo de linguagens: Java, C++, Python.
- **Princípios:**
 - **Encapsulamento:** Proteger os dados dentro de objetos.
 - **Herança:** Permitir que classes compartilhem e reutilizem código.
 - **Polimorfismo:** Permitir que métodos tenham diferentes implementações dependendo do contexto.
 - **Abstração:** Modelar conceitos do mundo real em código.

```
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "Latido"

meu_cachorro = Cachorro("Rex")
print(meu_cachorro.fazer_som()) # Saída: Latido
```

Cria uma classe chamada Animal.
O método especial `__init__` é o construtor, que inicializa um novo objeto da classe.
O parâmetro `nome` representa o nome do animal.
`self.nome = nome` -> atribui o valor passado (`nome`) ao atributo `self.nome` do objeto.

Esse método é um método abstrato (não faz nada por enquanto)
.pass é um placeholder que permite que o código compile sem erro. A ideia é que classes filhas (como Cachorro) sobrescrevam esse método para definir sons específicos.

Cria um objeto da classe Cachorro, passando "Rex" como nome. Isso chama o `__init__` da classe Animal (pois Cachorro herdou de Animal).

Como Cachorro sobrescreveu `fazer_som()`, a chamada `meu_cachorro.fazer_som()` retorna "Latido".

Cachorro herda de Animal, ou seja, Cachorro é uma subclasse de Animal. Herdar significa que Cachorro terá todas as características e métodos de Animal, mas pode sobrescrevê-los.

Aqui estamos sobrescrevendo o método `fazer_som()` da classe Animal. Agora, quando chamarmos `fazer_som()` em um objeto da classe Cachorro, ele retornará a string "Latido".

```
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "Latido"

meu_cachorro = Cachorro("Rex")
print(meu_cachorro.fazer_som()) # Saída: Latido
```

Animal é uma classe genérica com um método *fazer_som()* sem implementação.

Cachorro *herda* Animal e sobreescreve *fazer_som()* para retornar "Latido".

Criamos um *objeto* Cachorro chamado meu_cachorro com o nome "Rex".

Quando chamamos *meu_cachorro.fazer_som()*, obtemos "Latido".

Esse é um exemplo clássico de *Herança e Polimorfismo*, dois conceitos essenciais da Programação Orientada a Objetos

Resumo – Paradigma Imperativo

- ▶ A evolução das linguagens imperativas está relacionada à necessidade de criar abstrações que permitam aos programadores expressar soluções de maneira mais intuitiva, alinhando-se ao modelo sequencial de execução dos computadores, para produzir soluções eficientes e compreensíveis.
- ▶ Ressaltamos:
 - ▶ A necessidade de **abstrações** – As linguagens evoluem para facilitar a programação.
 - ▶ A afinidade com o modelo computacional – O paradigma imperativo **reflete a forma como os computadores executam instruções**.
 - ▶ A busca por eficiência e clareza – Tornar os programas mais fáceis de entender e mais eficientes na execução.

Alguns fatos atuais que mostram a necessidade de usar outros paradigmas


- ▶ Estado mutável significa que os dados de um programa podem ser alterados depois de terem sido criados. Isso é comum em paradigmas imperativos e orientados a objetos, onde variáveis são usadas para armazenar e modificar informações ao longo da execução do programa.
- ▶ A popularização de arquiteturas paralelas -> necessidade de multiprocessamento/ multi-threading -> variáveis acessadas por muitos processos/threads.
- ▶ Nesse ambiente concorrente (vários threads/processos acessando uma variável ao mesmo tempo) podemos ter **condições de corrida**. Dois threads podem tentar modificar a variável ao mesmo tempo, causando erros imprevisíveis.
- ▶ É necessário utilizar mecanismos de sincronização tornando o código mais complexo.

Exemplo Real: Twitter Migrando para Scala

- ▶ O Twitter inicialmente usava Ruby on Rails, que tem forte orientação a objetos.
 - ▶ Ruby tem uma **implementação single-threaded do interpretador**, dificultando o aproveitamento de múltiplos núcleos de CPU.
- ▶ Com o crescimento, enfrentaram problemas de concorrência e escalabilidade, pois a orientação a objetos gerava muito estado mutável, dificultando a execução eficiente em vários servidores.
- ▶ Solução? Scala -> Scala é **funcional**. Suporta **imutabilidade** e concorrência eficiente. Reduziu a complexidade no manuseio de múltiplas requisições simultâneas.

Outros exemplos de migração de paradigmas

- ▶ **Facebook**: Inicialmente em PHP (imperativo/OO), mas desenvolveram o Hack (linguagem híbrida com suporte a programação funcional).
- ▶ **WhatsApp**: Usava C++ (imperativo), mas adotou Erlang (funcional) para melhor suporte a concorrência.
- ▶ **Google**: Criou o Go, que tem suporte a concorrência funcional via goroutines, para substituir C++ em algumas áreas.

- 
- ▶ Estado mutável pode ser útil para o funcionamento de programas na arquitetura von neumann, mas traz desafios para paralelismo e concorrência.
 - ▶ Linguagens funcionais evitam esse problema garantindo imutabilidade (não há a ideia de atualização de variáveis – novos valores são criados) e ausência de efeitos colaterais.
 - ▶ Empresas como Twitter, Facebook e WhatsApp adotam linguagens funcionais para maior escalabilidade e eficiência.