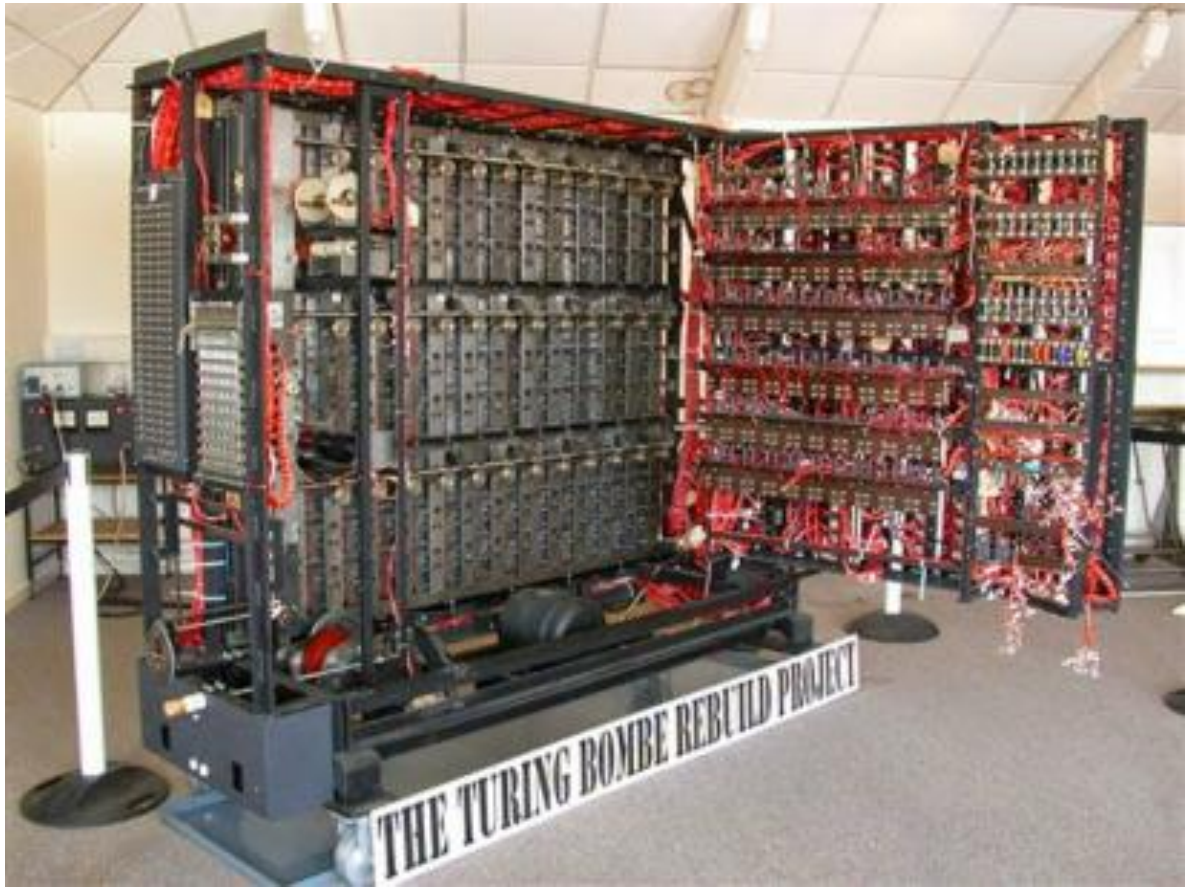
Paradigma Imperativo
Variáveis e tipos

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

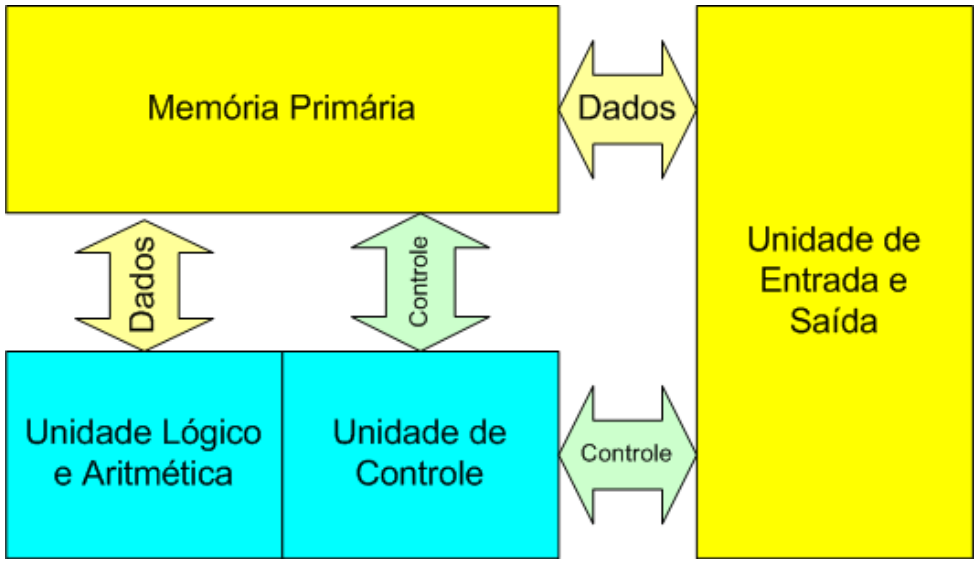
Paradigma imperativo

O modelo fundamental do paradigma imperativo a **Máquina de Turing**, que nada mais é que uma abstração matemática que corresponde ao conjunto de funções computáveis.



Paradigma imperativo

- A **Máquina de Turing** foi aprimorada por **John Von Neumann** a uma arquitetura de computadores que fundamenta os computadores construídos até hoje.



- A máquina de Von Neumman, consequentemente o paradigma Imperativo, é baseada na execução sequencial de comandos e no armazenamento de dados na **memória do computador** (**variáveis**).

Variável – Capítulo 4 Tucker

- Uma **variável** é uma **ligação** de um **nome** (identificador) com um **endereço de memória**. Além disso, uma variável possui um **tipo**, um **valor** e um **tempo de vida**.

- Um **tipo** é um conjunto de **valores** e um conjunto de **operações** sobre esses valores.

- Os tipos que todas as linguagens imperativas suportam são os **básicos** ou **primitivos** (inteiros, reais, booleanos e caracteres) e **compostos** (ponteiros, cadeias de caracteres, matrizes e estruturas/registros).

Tipagem estática e dinâmica – Capítulo 5 Tucker

- Em C um único tipo é associado a uma variável quando essa for declarada, permanecendo associada por toda sua vida em **tempo de execução**. Isso permite que o tipo de valor de uma expressão seja determinado em **tempo de compilação**, ou seja, temos uma **tipagem estática**.

- O Python permite que o tipo de uma variável seja redefinido cada vez que um novo valor for atribuído a ela, em **tempo de execução**, temos assim uma **tipagem dinâmica**.

- O seria uma linguagem não tipada ? Poderíamos dizer que o Python é uma linguagem não tipada ?

Tipo ponteiro – Capítulo 5 Tucker

- Um **ponteiro** é um valor que representa uma referência ou um endereço de memória e são usados nas linguagens C, C++, Ada e Perl.
- A linguagem C fornece duas operações com ponteiros: o operador **endereço** *de* (unário &) recebe uma variável como argumento e retorna o endereço dessa variável, veja o exemplo:

```
int main (void) {
   int i = 1234;
   printf (" valor de i       = %d\n", i);
   printf ("long int &i       = %ld\n", (long int) &i);
   printf ("hexadadecimal &i = %p\n", &i);
   return 0;
}
```

Tipo ponteiro – Capítulo 5 Tucker

- O operador de **desreferenciação** (unário*) recebe uma referência e acessa o valor dessa referência (conteúdo). O operador (*) também é usada para declarar uma variável do ponteiro.
- Para declarar uma variável do tipo ponteiro que armazena a referência de uma variável do tipo `int` basta usar **`int *p;`**

```
int main (void) {
   int i; int *p;
   i = 1234; p = &i;
   printf ("*p = %d\n", *p);
   printf (" p = %ld\n", (long int) p);
   printf (" p = %p\n", (void *) p);
   printf ("&p = %p\n", (void *) &p);
   return 0;
}
```

Tipos definidos pelo programador – Capítulo 5 Tucker

- Uma **estrutura** ou um **registro** é um conjunto de elementos de tipos diferentes com nomes distintos denominados **campos**.

- A utilização das **estrutura/registro** se originou nas linguagens **Pascal** e **C**, em **Python** as estruturas são como listas e a **linguagem Java** omite as estruturas completamente, utilizando uma classe para essa abstração, e a **linguagem C++** possui tanto estruturas quanto classes.

Tipos definidos pelo programador – Capítulo 5 Tucker

- Definindo um tipo na linguagem C.

  Você conseguiria definir um tipo (estrutura/registro) com três campos que possam ser usados para armazenar **datas** (dia, mês e ano) ?

Tipos definidos pelo programador – Capítulo 5 Tucker

- Tipo data

```
typedef struct {
   int dia;
   int mes;
   int ano;
}data;
```

- Para criar uma variável do tipo data e atribuir valor temos:

```
data hoje;
hoje.dia = 07;
hoje.mes = 02;
hoje.ano = 2020;
```

Estruturas e ponteiros – Capítulo 5 Tucker

- Cada registro tem um endereço na memória do computador, podemos usar uma variável do tipo ponteiro para armazenar o endereço de um registro, por exemplo:

```
data *p; // p é um ponteiro para registros do tipo data
data hoje;
p = &hoje; // agora p aponta para hoje
(*p).dia = 31; // mesmo efeito que x.dia = 31
               // visto anteriormente
```

- Cuidado! a expressão `*p.dia` equivale a `*(p.dia)` e tem significado muito diferente de `(*p).dia`.
- A expressão `p->dia` é uma abreviatura muito útil da expressão `(*p).dia` , ou seja, `p->dia = 31` tem o mesmo valor que `(*p).dia = 31`

Tempo de vida e escopo de variável – Capítulo 4 Tucker

- Considere o exemplo da função abaixo, Qual sua saída do programa, tente explicar ?

```
 1 int soma=0;
 2 int calcula( int vetor[], int tam){
 3    int soma=0,i;
 4    for( i=0; i < tam; i++){
 5      soma = soma + vetor[i];
 6    }
 7    return soma;
 8 }
 9 int main (void) {
10    int v[]={1,2,3,4};
11    printf("soma1=%d\n",calcula(v,4));
12    printf("soma=%d\n",soma);
14    ....
```

Escopo de uma variável – Capítulo 4 Tucker

- O **escopo de uma variável** permite que os programadores reusem o mesmo identificador para definir variáveis com o mesmo nome em escopo diferentes, o identificador é sempre associado à declaração mais recente da variável, com base no histórico de execução do programa.

- No exemplo a variável **`soma`** é declarada logo no início do programa função (linha 1) e dentro do corpo da função (linha 3), a declaração dentro do laço sobrepõe a declaração externa (escopo mais interno), ou seja, a instrução **`soma = soma + vetor[i];`** é realizada utilizando a variável **`soma`** com a declaração mais recente (linha 3).

Tempo de vida de uma variável – Capítulo 4 Tucker

- O **tempo de vida de uma variável** se refere ao intervalo de tempo durante qual a variável fica alocada na memória, o que pode ocorrer **estática** ou **dinamicamente** .
- As variáveis com **tempo de vida estático** (variáveis globais) são alocadas a memória no início da execução do programa e permanecem "vivas" até que o programa termine.
- As variáveis com **tempo de vida dinâmico** são alocadas a partir da pilha de tempo de execução, por exemplo as variáveis declaradas dentro de funções são alocadas quando a função é chamada e são liberas quando a função completa sua execução.

Tempo de vida e escopo de variável – Capítulo 4 Tucker

- No exemplo a variável **`soma`** (linha 1) tem **tempo de vida estático**, ou seja, global e permanece alocada durante todo o tempo de execução do programa. Dentro da função **`calcula()`** é declarada dinamicamente (na pilha) outra variável **`soma`** que ficará viva somente durante a execução da função **`calcula()`**.

Exercícios

1) Para linguagem C, C++ e Java, liste todos os **tipos básicos** suportados e seus tamanhos em bytes.

2) Explique o seria uma linguagem fortemente tipada, de exemplos de linguagens fortemente tipadas e não fortemente tipadas.

3) Explique porque a expressão `*p.dia` tem significado muito diferente de `(*p).dia`.

4) Leia a seção 5.4.3 do livro de Sebesta e responda qual é vantagem e desvantagens de se utilizar variáveis com tempo de vida estático.

Exercícios

5) Escreva um programa que leia um inteiro representando um tempo medido em minutos e calcula o número equivalente de horas e minutos armazenando o cálculo em um tipo registro (por exemplo, 131 minutos equivalem a 2 horas e 11 minutos). Para esse exercício use o tipo **`tempo`** definido abaixo para armazenar a resposta do programa, ao final o seu programa imprime o resultado armazenado na variável do tipo **`tempo`**.

```
typedef struct{
     int horas;
     int minutos;
}tempo;
```

Exercícios

6) Escreva um programa que leia um valor inteiro correspondente à idade de uma pessoa em dias e informe-a em anos, meses e dias armazenado em variável registro do tipo **`data`** definida abaixo:

```
typdef struct{
     int dia;
     int mes;
     int ano;
}data;
```

Obs.: apenas para facilitar o cálculo, considere todo ano com 365 dias e todo mês com 30 dias. Nos casos de teste nunca haverá uma situação que permite 12 meses e alguns dias, como 360, 363 ou 364. Este é apenas um exercício com objetivo de testar seu entendimento do tipo registro.

Para saber mais

- Leia os capítulo 4 seções 4.2, 4.3 e 4.9 e capítulo 5 seções 5.2, 5.3 e 5.4 do livro TUCKER, A. B.; NOONAN, R. E. **Linguagens de programação: Princípios e Paradigmas.**

- Leia os capítulo 5 as seções 5.4 e 5.6 do livro SEBESTA, R.W. **Conceitos de Linguagens de programação.**

Fim