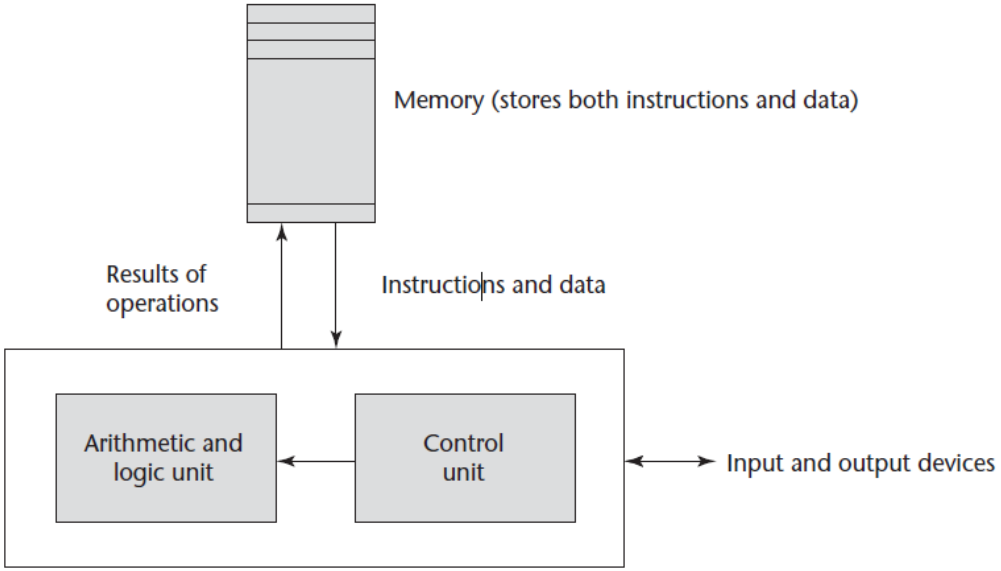
Conceitos de Linguagens de Programação

Aula 07 – Nomes, Vinculações, Escopos e Tipos de Dados

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Introdução

- Linguagens de programação imperativas são abstrações da arquitetura de **Von Neumann**:
  - **Memória:** armazena instruções e dados;
  - **Processador:** fornece informações para modificar o conteúdo da memória.
    - As abstrações para as células de memória da máquina são as **variáveis**;



Nomes

- Um **nome** (ou identificador) é uma cadeia de caracteres usado para identificar alguma entidade em um programa.

- **Questões de projeto:**
  - Qual é o tamanho máximo de um nome?
  - Caracteres especiais podem ser usados em nomes?
  - Os nomes fazem distinção entre minúsculo e maiúsculo?
  - As palavras especiais são palavras reservadas ou palavras-chave?

Nomes: Formato dos Nomes

- Geralmente nomes devem iniciar com letras (a..z) ou _ (underline) e seguidos por letra _ ou dígitos.

- Em algumas linguagens os nomes são case sensitive:
  - Exemplo: iprj, Iprj e IPRJ são nomes distintos;
  - Afeta a legibilidade;

Nomes: Tamanho de Nomes

- Tamanhos para nomes estabelecidos por algumas linguagens de programação:
  - **FORTRAN**: máximo 6;
  - **COBOL**: máximo 30;
  - **FORTRAN 90 e ANSI C**: máximo 31;
  - **Ada**: sem limite*;
  - **C++**: sem limite*;

* Implementadores frequentemente estabelecem limite.

Nomes: Palavras Especiais

- **Palavra-chave:** palavra que é especial apenas em certos contextos
  - Exemplo em FORTRAN:

```
INTEGER REAL
REAL INTEGER
```

- **Palavra reservada:** palavra que não pode ser usada como um nome definido pelo usuário
  - Quanto maior a quantidade de palavra reservada, maior a dificuldade do usuário definir nomes para as variáveis

Variáveis

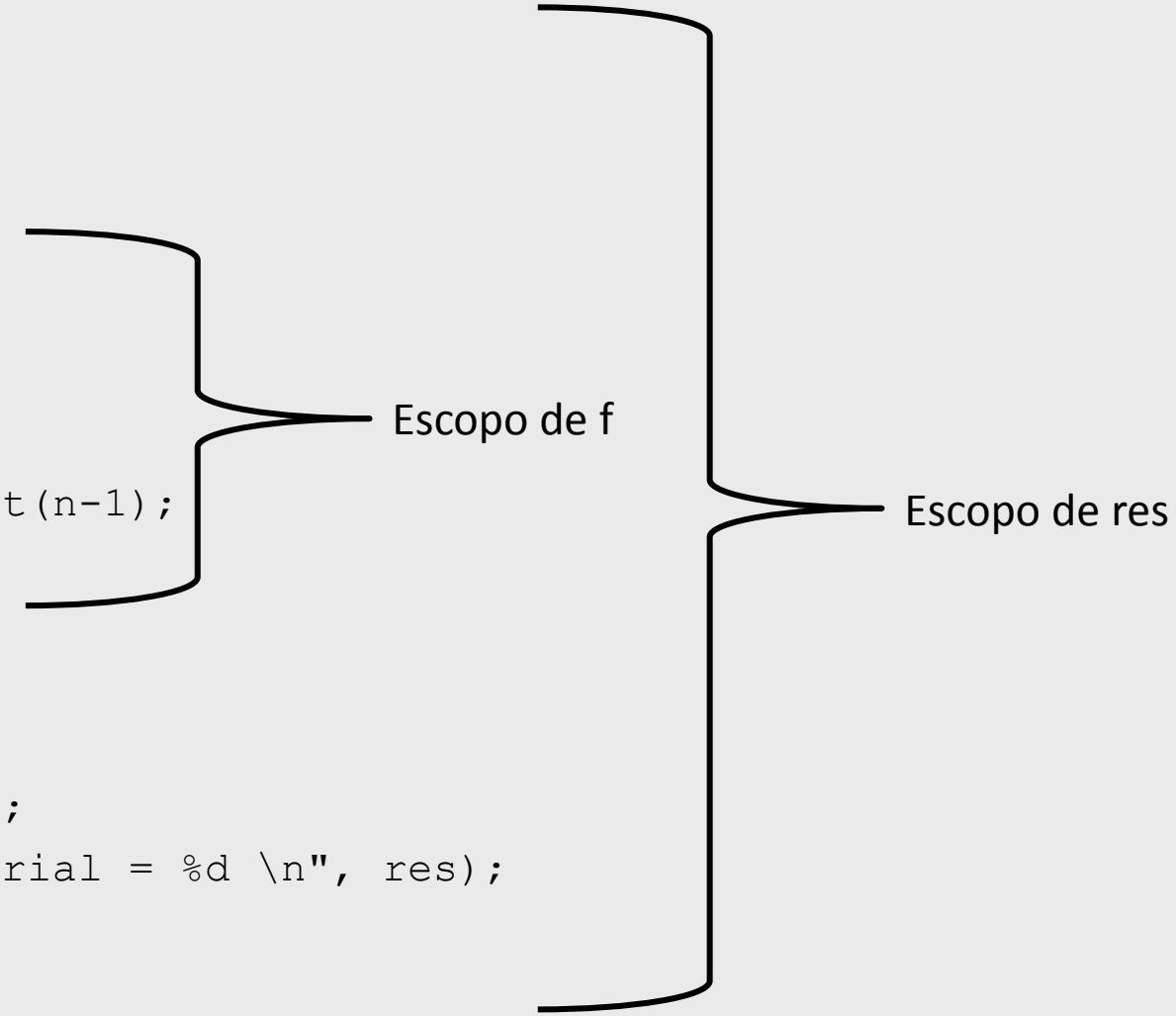
- Variável é uma abstração de uma **célula de memória**;
- Surgiram durante a mudança das linguagens de programação de baixo para alto nível;
- **Pode ser caracterizada por 6 atributos:**
  - **Nome:** identificador;
  - **Endereço:** localização da memória a ela associado;
  - **Tipo:** intervalo de possíveis valores e operações
  - **Valor:** o que está armazenado na variável num determinado momento;
  - **Tempo de vida:** tempo durante o qual a memória permanece alocada para a variável;
  - **Escopo:** partes do programa onde a variável é acessível;

Variáveis – Escopo

```
#include <stdio.h>

int res;

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}
int main(void)
{
  res = fat(3);
  printf("Fatorial = %d \n", res);
  return 0;
}
```

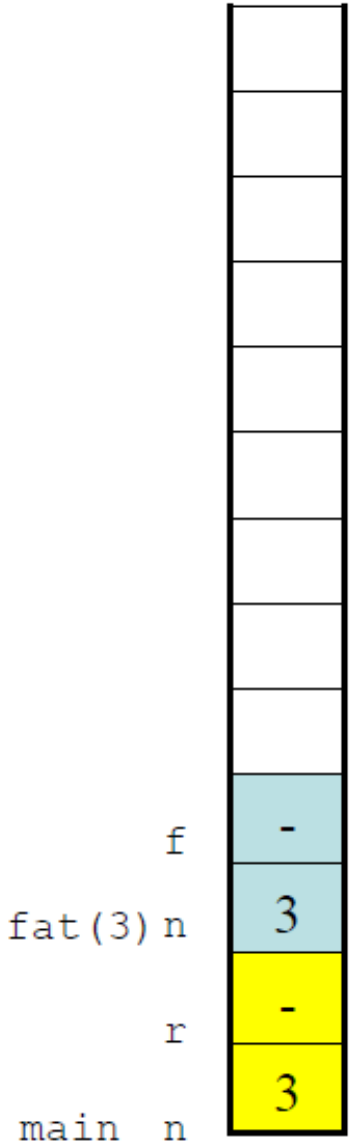


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

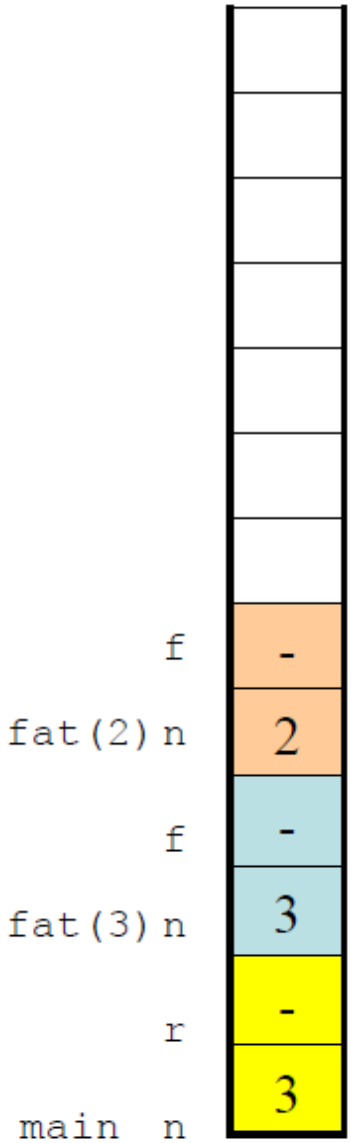


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

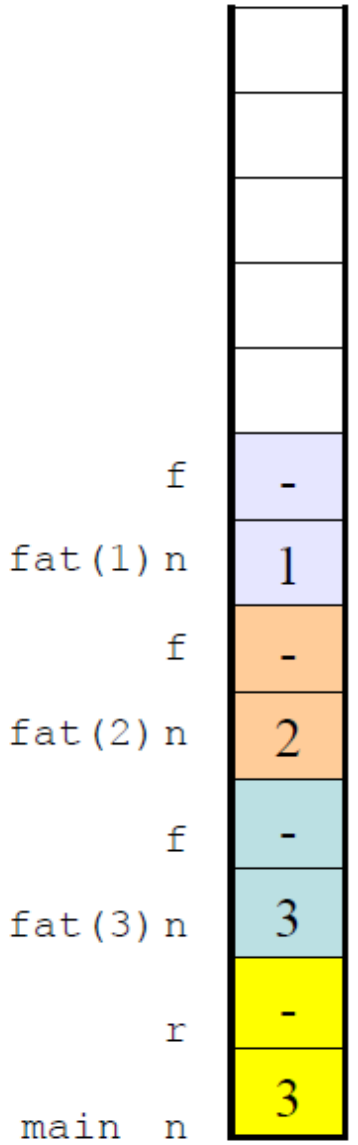


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

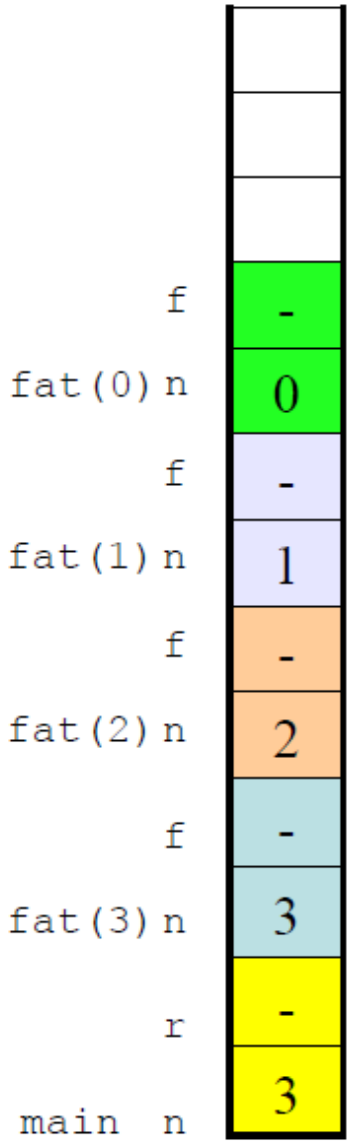


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

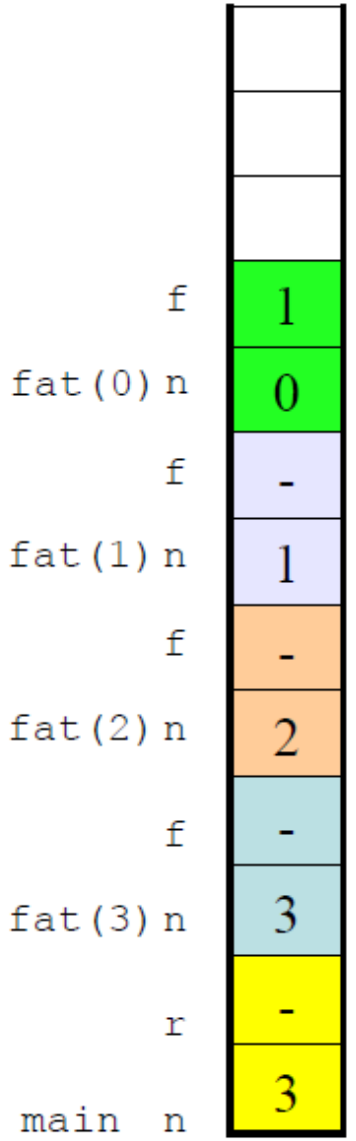


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

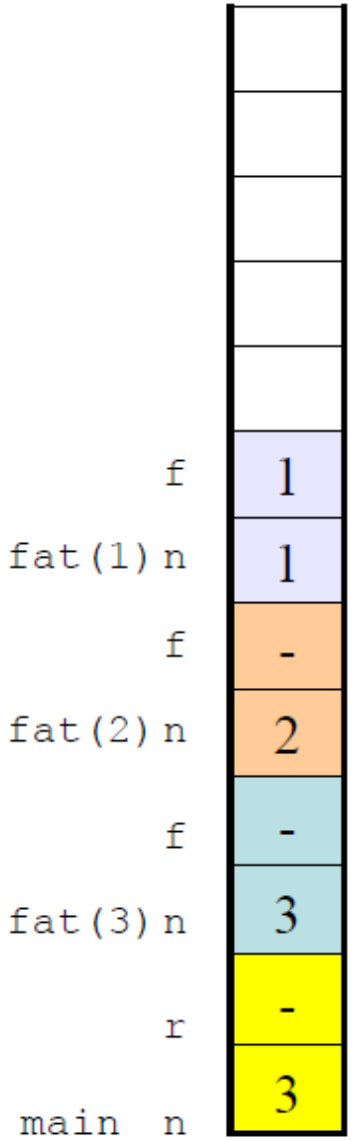


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

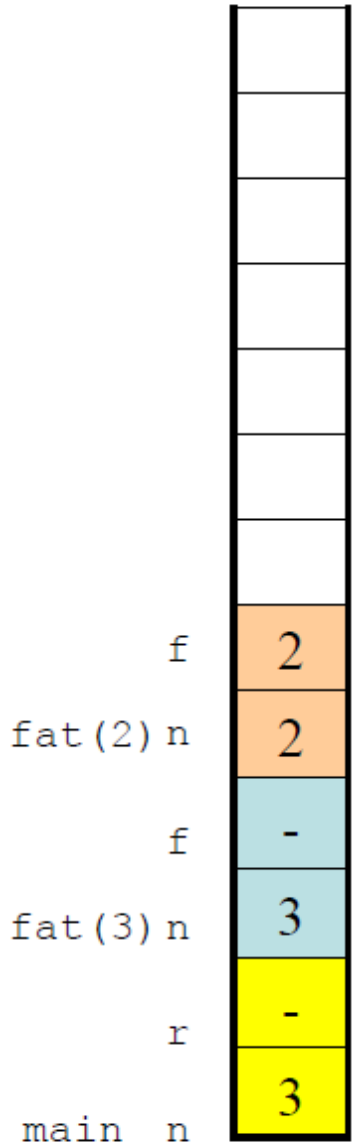


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

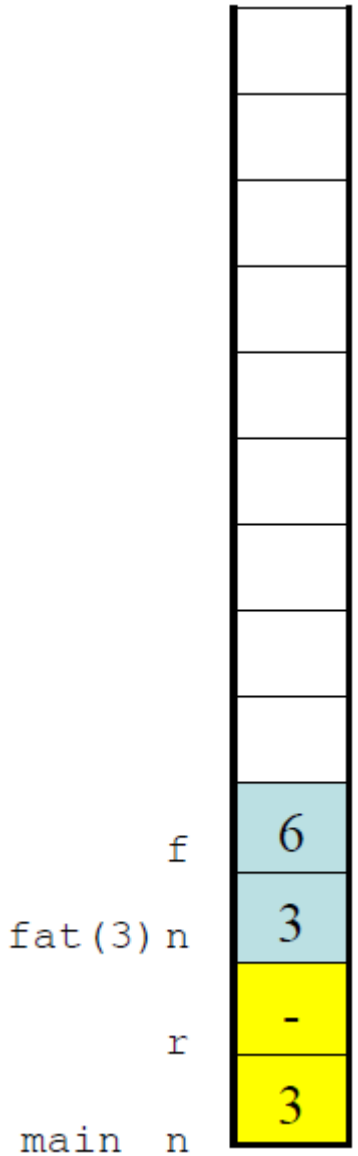


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

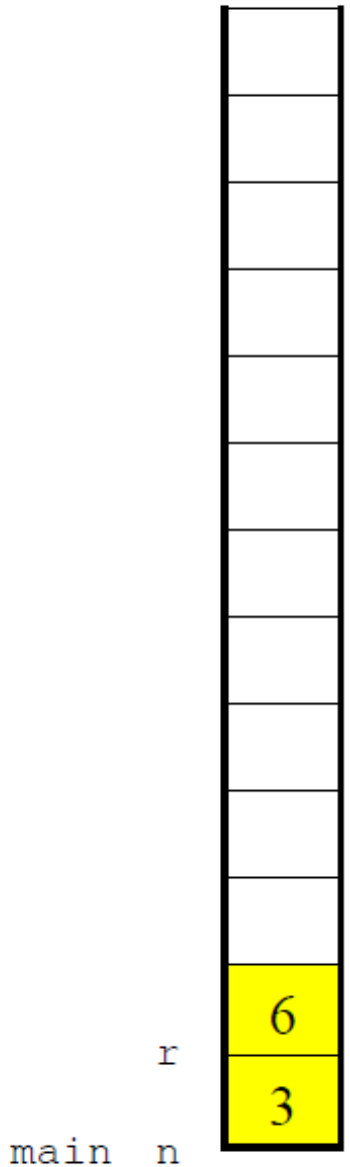


Variáveis – Tempo de Vida

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```



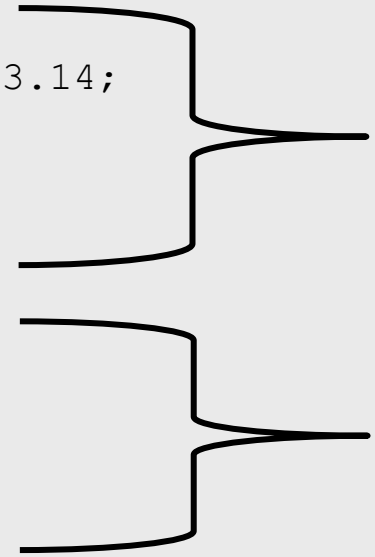
O conceito de Vinculação

- **Vinculação** (binding): associação de um (nome) identificador a sua declaração no programa.
- Exemplo:

```
const int z = 0;
char c;

int func()
{
  const float c = 3.14;
  bool b;
  ...
}

int main()
{
  ...
}
```



c: constante 3.14
b: variável boleana
z: constante 0
func: função

c: variável char
z: constante 0
func: função

O conceito de Vinculação

- **Funcionamento da vinculação:** A = B + C
  1. Obter os endereços de A, B e C;
  2. Obter os dados dos endereços (memória) B e C;
  3. Computar o resultado de B + C;
  4. Armazenar o resultado na locação de A;

- Apesar da abstração de endereços em nomes, as linguagens imperativas mantém os 4 passos como uma unidade de programa padrão.

Vinculação

- **Dado objeto = (L, N, V, T)**, onde:
  - L – locação;
  - N – nome;
  - V – valor;
  - T – tipo;

- A vinculação é a atribuição de valor a um dos quatro componentes acima.

- **As vinculações podem ocorrer em:**
  - **Tempo de compilação**;
  - **Tempo de loading** (quando o programa gerado pelo compilador está sendo alocado à locações específicas na memória);
  - **Tempo de execução**.

Tipos de Vinculações

- **Vinculações de locação:** geralmente ocorrem em tempo de loading, mas também podem ocorrer em tempo de execução (variáveis em procedimentos e alocação dinâmica).

- **Vinculações de nome:** ocorrem tipicamente em tempo de compilação, quando uma declaração é encontrada.

- **Vinculações de valor:** ocorrem tipicamente em tempo de execução.

Tipos de Vinculações

- **Vinculações de tipo:** ocorrem geralmente em tempo de compilação, através de declarações de tipo.

- **Vinculações de tipo dinâmico:** ocorre durante tempo de execução, não havendo declarações de tipo. O tipo do dado objeto é determinado pelo tipo do valor, e portanto, uma mudança de valor implica em nova vinculação.

Tipos de Vinculações

- **Exemplo: x = x + 5**
  - O tipo de x é vinculado em **tempo de compilação**;
  - O conjunto de possíveis valores de x é vinculado em **tempo de loading**;
  - O significado de + é vinculado em **tempo de compilação**, após a determinação dos tipos dos operandos;
  - 5 é vinculado em **tempo de loading**;

Tipos de Vinculações

- **Vinculação Estática:** ocorre antes da execução e permanece inalterado durante a execução do programa;

- O tipo pode ser especificado por uma declaração **explícita** ou **implícita**:
  - Uma declaração explícita é uma sentença declarativa para o tipo da variável:
    - Exemplo: int x;

  - Uma declaração implícita é um mecanismo padrão para a especificação de tipos – acontece na primeira vez que uma variável é encontrada:
    - Exemplo: em Fortran, se uma variável não for explicitamente declarada, usa-se a convenção para declará-la implicitamente: se a variável começar com I, J, K, L, M ou N, recebe tipo integer; caso contrario recebe tipo real.

Tipos de Vinculações

- **Vinculação Dinâmica:** ocorre durante a execução ou pode ser alterado durante a execução do programa;

- O tipo de uma variável não é especificado por uma sentença de declaração, nem pode ser determinado pelo nome da variável;

- A variável é vinculada a um tipo quando é atribuído um valor a ela.
  - Exemplo: PHP

```
$varA = 10; //variável int
$varA = "teste"; //passa a ser uma string
```

Vinculações de Armazenamento

- Um caráter fundamental de uma linguagem de programação é determinando pelo projeto das vinculações de **armazenamento** para suas variáveis.

- É necessário definir como ocorrem essas vinculações:

  - Célula de memória a qual uma variável é vinculada é obtida a partir das células de memória disponíveis → **alocação**

  - Quando a variável não necessita mais desta célula de memória, torna-se necessário devolvê-la ao conjunto de células disponíveis → **liberação**

Tempo de Vida

- O tempo de vida de uma variável é o tempo durante o qual ela está vinculada a uma célula de memória.

- Os tipos de variáveis são classificadas em:
  - Variáveis estáticas;
  - Variáveis dinâmicas na pilha;
  - Variáveis dinâmicas no monte (heap) explícitas;
  - Variáveis dinâmicas no monte (heap) implícitas;

Variáveis Estáticas

- Vinculadas a células de memória antes da execução, permanecendo na mesma célula de memória durante toda a execução do programa.
  - Exemplo: variáveis estáticas em C

- **Vantagem:**
  - Eficiência (endereçamento direto);
  - Não há sobrecarga em tempo de execução para alocação e liberação.

- **Desvantagem:**
  - Falta de flexibilidade (sem recursão);
  - Armazenamento não compartilhado entre variáveis.

Variáveis Estáticas

- Exemplo em C:

```
void func()
{
  static int x = 0;
  x++;
  printf("%d\n", x);
}

int main()
{
  int cont;
  for (cont = 0; cont < 100; cont++)
  {
    func();
  }
}
```

Variáveis Dinâmicas de Pilha

- Vinculações são criadas quando suas sentenças de declaração são efetuadas, mas o tipo é estaticamente vinculado.
  - Exemplo: em C, as declarações de variáveis que aparecem no início de um método são elaboradas quando o método é chamado e as variáveis definidas por essas declarações são liberadas quando o método completa sua execução.

- **Vantagem:**
  - Permitem recursão e otimizam o uso de espaço em memória.

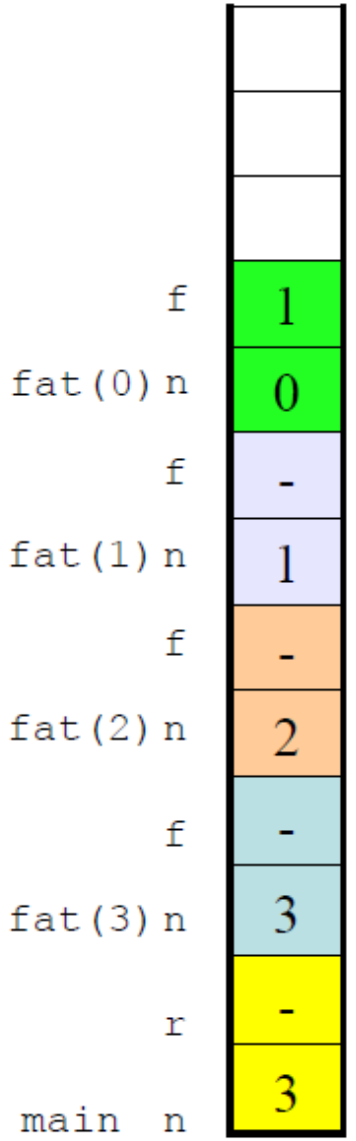
- **Desvantagens:**
  - Sobrecarga de alocação e liberação.

Variáveis Dinâmicas de Pilha

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```



Variáveis Dinâmicas na Heap (explícitas)

- Variáveis dinâmicas do monte (heap) explícitas são células de memória não nomeadas (abstratas), que são alocadas e liberadas por instruções explícitas, especificadas pelo programador, que tem efeito durante a execução.
  - Variáveis referenciadas através de ponteiros.

- **Vantagem:**
  - Armazenamento dinâmico.

- **Desvantagens:**
  - O programador é responsável pelo gerenciamento da memória (não confiável).

Variáveis Dinâmicas na Heap (explícitas)

- Exemplo em C:

```
int *valor;
valor = (int*) malloc(sizeof(int));

...

free(valor);
```

Variáveis Dinâmicas na Heap (implícitas)

- Variáveis dinâmicas do monte implícitas são vinculadas ao armazenamento no monte (heap) apenas quando são atribuídos valores a elas.
  - Exemplo em JavaScript:

```
highs = [74, 84, 86, 90, 71];
```

- **Vantagem:**
  - Flexibilidade.

- **Desvantagens:**
  - Sobrecarga em tempo de execução.

Verificação de Tipos de Variáveis

- Verificação ou Checagem de Tipo é uma atividade de garantia de que operandos e operadores são de tipos compatíveis
  - Tipo compatível é um tipo que é permitido para um operando segundo as regras da linguagem. Pode ser que haja a conversão automática para garantir a compatibilidade.

- Uma linguagem é fortemente tipada se erros de tipo são sempre detectados.
  - A linguagem C e C++ não é fortemente tipificada:
    - Permitem funções cujos parâmetros não são verificados quanto ao tipo.
  - A linguagem java é fortemente tipificada.

Escopo

- O escopo de uma variável é a faixa de instruções na qual a variável é visível.
  - Uma variável é visível em uma instrução se puder ser referenciada nessa instrução.
- **Variáveis locais** vs **variáveis globais**.

- É possível definir escopos através de blocos. Exemplo em C:

```
int a, b;
if (a > b){
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Blocos e Escopo

- Exemplo:

```
void sub()
{
  int count;
  . . .
  while (. . .)
  {
    int count;
    count++;
    . . .
  }
  . . .
}
```

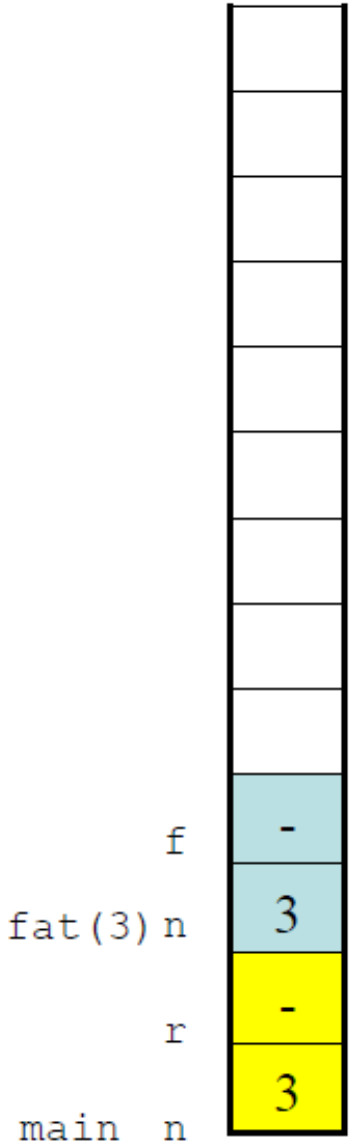
- Válido em C e C++
- Ilegal em Java e C#

Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

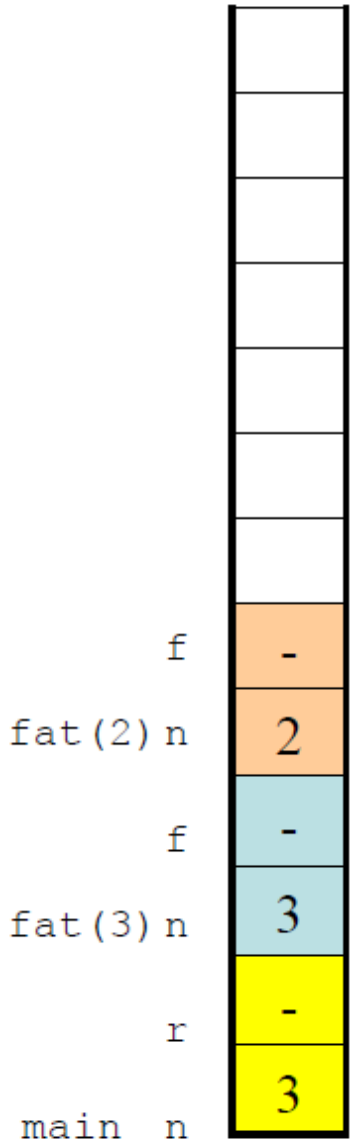


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

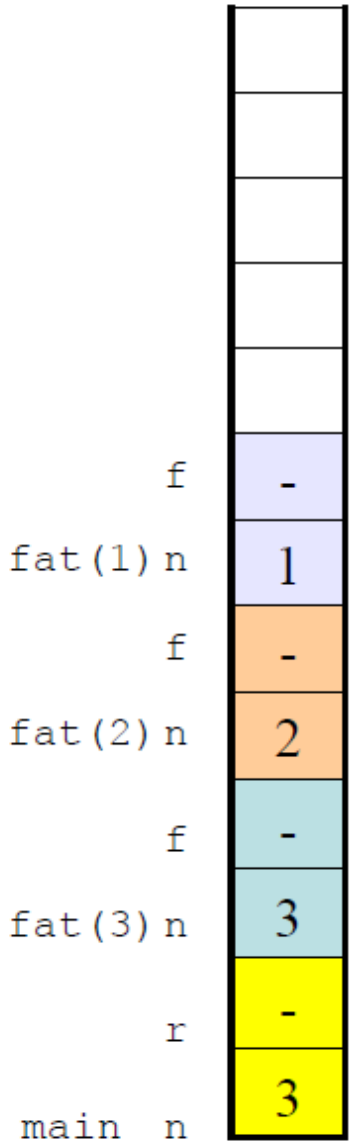


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

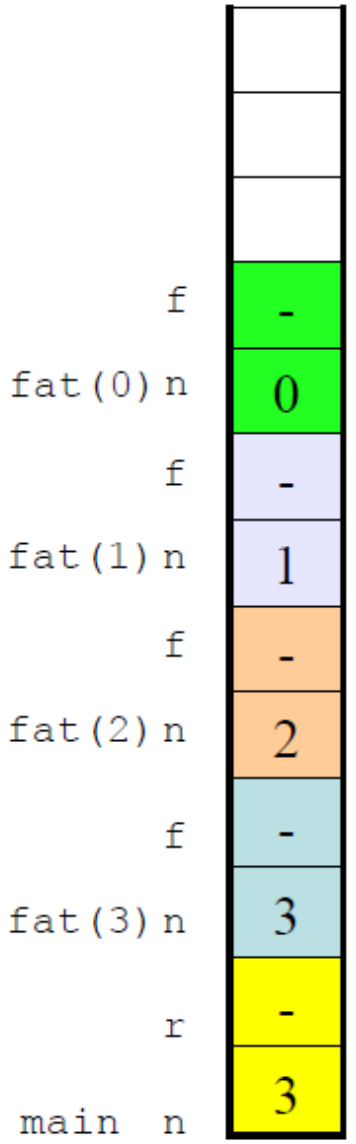


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

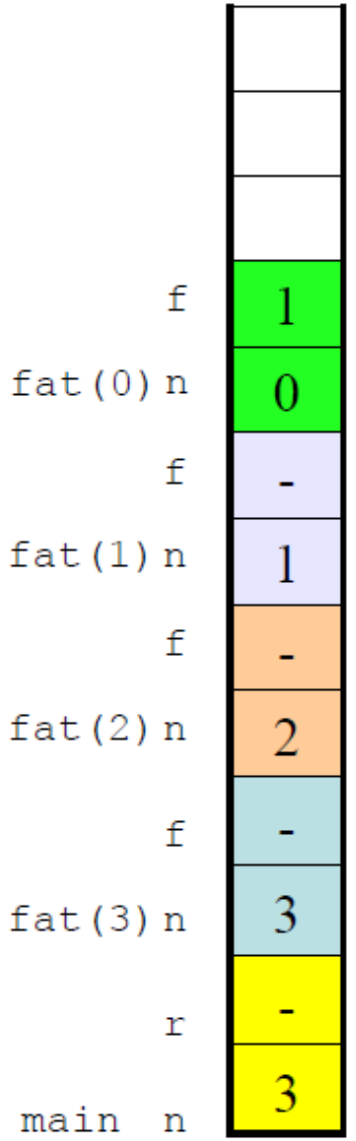


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

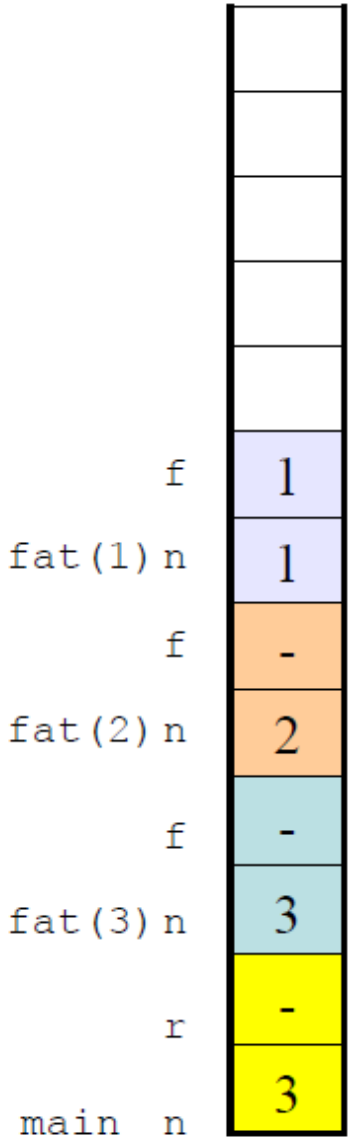


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

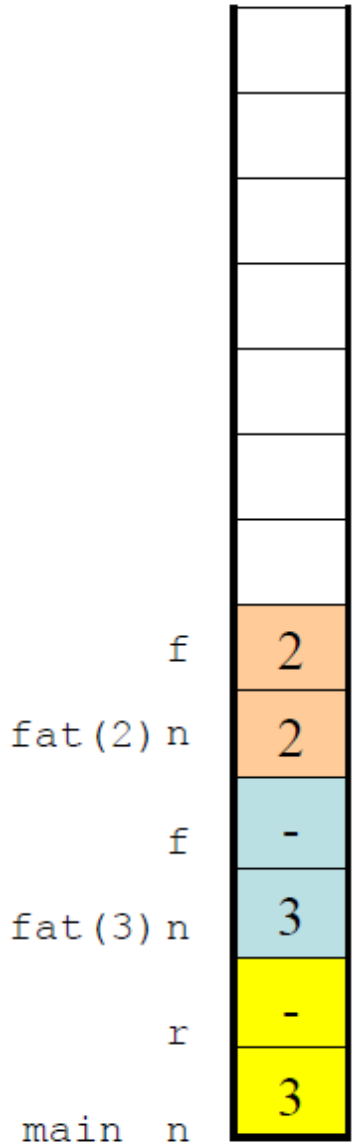


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

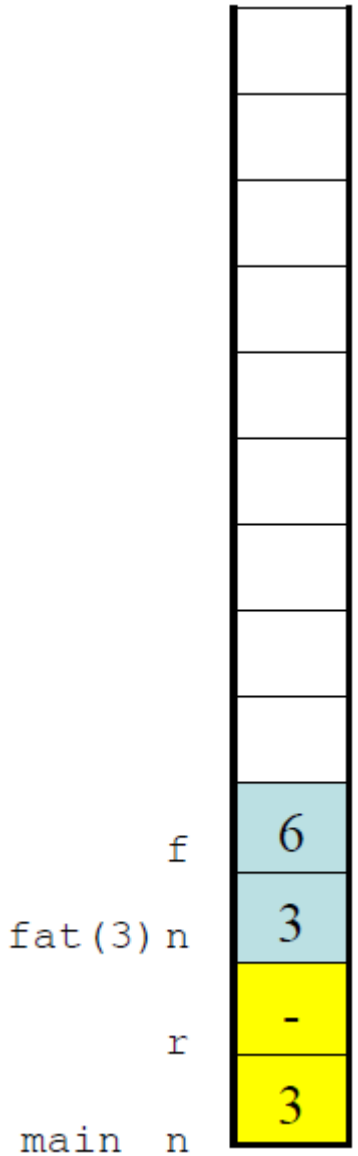


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```

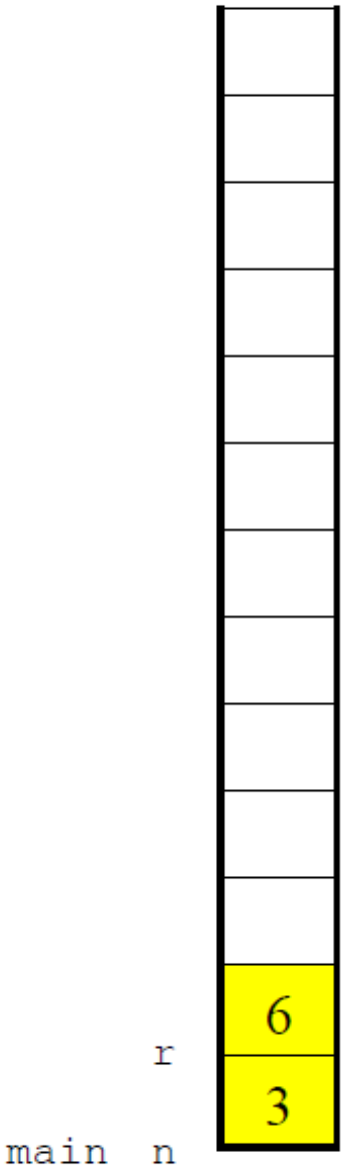


Blocos e Escopo – Implementação

```
#include <stdio.h>

int fat(int n)
{
  int f;
  if (n == 0)
    f = 1;
  else
    f = n * fat(n-1);
  return f;
}

int main(void)
{
  int n = 3, r;
  r = fat(n);
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}
```



Tipos de Dados

- Um **tipo de dado** define uma coleção de dados e um conjunto de operações predefinidas sobre esses dados.
- É importante que uma linguagem de programação forneça uma **coleção apropriada de tipos de dados**.
- **Sistema de tipos:**
  - Define como um tipo é associado a uma expressão;
  - Inclui regras para equivalência e compatibilidade de tipos;
- Entender o sistema de tipos de uma linguagem de programação é um dos aspectos mais importantes para entender a sua **semântica**.

Tipos de Dados Primitivos

- Os **tipos de dados primitivos** são os tipos de dados não-definidos em termos de outros tipos.

- Praticamente todas as linguagens de programação oferecem um conjunto de tipos de dados primitivos.

- Usados com construções de tipo para fornecer os tipos estruturados. Os mais comuns são:
  - Tipos numéricos;
  - Tipos booleanos;
  - Tipos caracteres;

Tipos de Dados Primitivos: Inteiro

- O tipo de dados primitivo numérico mais comum é o inteiro.

- Atualmente, diferentes tamanhos para inteiros são suportados diretamente pelo hardware.
  - Exemplo – Java: byte, short, int, long

Tipo	Tamanho (Bits)	Descrição
byte	8 (1 byte)	Pode assumir valores entre $-2^7 = -128$ e $2^7 = +128$.
short	16 (2 bytes)	Pode assumir valores entre -2^{15} e 2^{15}
int	32 (4 bytes)	Pode assumir valores entre -2^{31} e 2^{31}
long	64 (8 bytes)	Pode assumir valores entre -2^{63} e 2^{63}

Tipos de Dados Primitivos: Ponto Flutuante

- Tipos de dados de ponto flutuante modelam os números reais.

- A maioria das linguagens de programação de fins científicos suportam pelo menos dois tipos de ponto flutuante.
  - Exemplo – Java: float e double

Tipo	Tamanho (Bits)	Descrição
float	32 (4 bytes)	O menor valor positivo representável por esse tipo é 1.40239846e-46 e o maior é 3.40282347e+38.
double	64 (8 bytes)	O menor valor positivo representável é 4.94065645841246544e-324 e o maior é 1.797693134862315 7e+308.

Tipos de Dados Primitivos: Ponto Flutuante

- Valores de ponto flutuante são representados como **frações expoentes** (máquinas mais antigas). Máquinas atuais usam o formato padrão **IEEE Floating-Point Standard 754**.

- Os valores que podem ser representados pelo tipo ponto flutuante é definido em termos de:
  - **Precisão:** exatidão da parte fracionária de um valor (medida pela quantidade de bits);
  - **Faixa:** combinação da faixa de frações e, o mais importante, de expoentes.

Tipos de Dados Primitivos: Booleano

- Tipo mais simples de dado primitivo;
- Faixa de valores: dois elementos, um para “true” e um para “false”;
- Um inteiro poderia ser utilizado para representá-lo, porem dificulta a legibilidade.
  - C não apresenta o tipo booleano: 0 = falso e 1 = verdadeiro

Tipos de Dados Primitivos: Caracteres

- Armazenados como codificações numéricas;
- Tradicionalmente, a codificação mais utilizada era o **ASCII** de 8 bits
  - Faixa de valores entre 0 e 127 para codificar 128 caracteres diferentes.
- Uma alternativa, codificação de 16 bits: **Unicode** (UCS-2)
  - Inclui caracteres da maioria das linguagens naturais
    - Usado em Java;
    - C# e JavaScript também suportam Unicode.

Tipo Cadeia de Caracteres (Strings)

- Valores consistem em sequências de caracteres;

- **Questões de projeto:**
  - É um tipo primitivo ou apenas um tipo especial de vetores de caracteres?
  - As cadeias devem ter tamanho estático ou dinâmico?

Tipo Cadeia de Caracteres (Strings)

- **String de Tamanho Estático:**
  - Tamanho especificado na declaração;
  - "Strings cheias", caso uma string mais curta for atribuída, os caracteres vazios são definidos como brancos (espaços).
- **String de Tamanho Dinâmico Limitado:**
  - Possuem tamanhos variáveis até um máximo declarado e fixo estabelecido pela definição da variável;
  - Tais variáveis podem armazenar qualquer número de caracteres entre zero e o máximo.
- **String de Tamanho Dinâmico:**
  - Possuem tamanhos variáveis sem limites;
  - Tal opção exige a alocação e desalocação dinâmica de armazenamento, mas proporciona máxima flexibilidade.

Tipo Cadeia de Caracteres (Strings)

- **Strings em C:**
  - Não são definidas como tipo primitivo
  - Usam vetores char e uma biblioteca de funções que oferecem operações (string.h)
  - Finalizadas por um caractere especial, nulo, representado com um caractere especial ('\0')
    - Operações são realizadas até encontrar este caractere.
    - Exemplo: char str[] = "teste";
      - "teste\0"
  - Biblioteca de funções:
    - strcpy, strcmp, strlen, strcat...

Tipo Cadeia de Caracteres – Implementação

- **Tamanho estático:**
  - Descritor em tempo de compilação;
- **Tamanho dinâmico limitado:**
  - Podem exigir um descritor em tempo de execução para armazenar tanto o tamanho máximo como o tamanho atual;
- **Tamanho dinâmico:**
  - Exigem um descritor em tempo de execução;
  - Exigem um gerenciamento de armazenagem mais complexo;
  - Alocação e desalocação.

Leitura Complementar

- Sebesta, Robert W. **Conceitos de Linguagens de Programação**. Editora Bookman, 2011.

- **Capítulo 5: Nomes, Vinculações, Verificação de Tipos e Escopos**

- **Capítulo 6: Tipos de Dados**