



S201 – Paradigmas de Programação

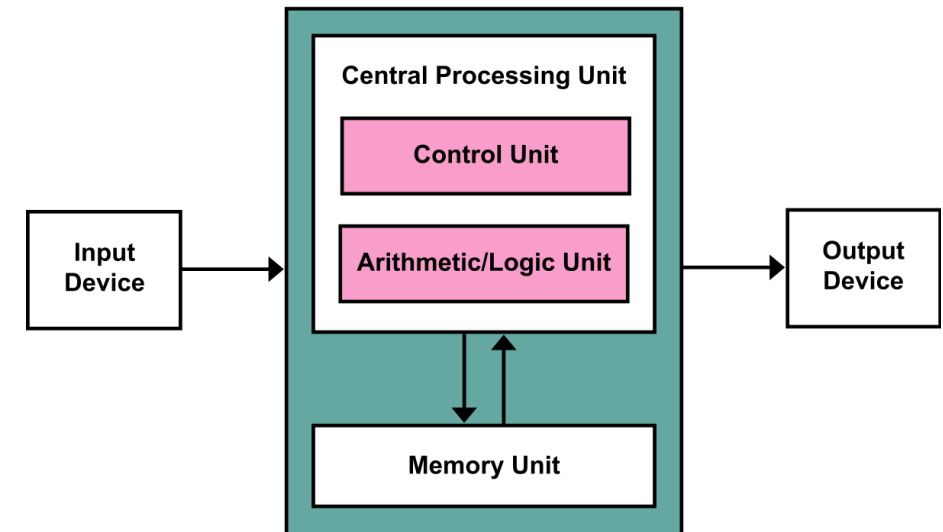
# NOMES, VINCULAÇÕES E ESCOPO

Marcelo Vinícius Cysneiros Aragão

[marcelovca90@inatel.br](mailto:marcelovca90@inatel.br)

# Programação Imperativa

- É o paradigma mais antigo e popular de todos.
- Baseia-se na [arquitetura de Von Neumann](#).
- Os programas definem sequências de **comandos** para o computador que mudam seu **estado** (exemplo: um conjunto de variáveis).




- Comandos são armazenados na memória e executados na ordem em que são encontrados.
- Comandos recuperam dados, realizam cálculos e atribuem o resultado a um local de memória.

# Programação Imperativa

- Elementos centrais do paradigma imperativo:
  - **Instrução de atribuição:** atribui valores a locais de memória e muda o estado atual de um programa.
  - **Variáveis:** referem-se a locais de memória.
  - **Execução passo-a-passo** de comandos.
  - **Controle de fluxo:** laços de repetição para alterar o fluxo de um programa.
  - Exemplo do cálculo do fatorial de um número: \_\_\_\_\_

```
unsigned int n = 5;  
unsigned int result = 1;  
while(n > 1) {  
    result *= n;  
    n--;  
}
```



# VARIÁVEIS

# Variáveis

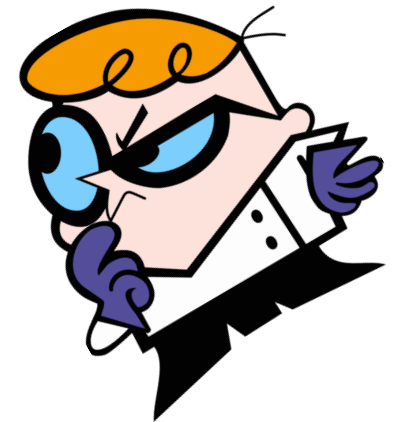
- Uma **variável** de programa é uma **abstração** de uma célula de memória de um computador ou de uma coleção de células.
  - Surgiram durante a mudança das LP de baixo para alto nível.
- Programadores geralmente pensam em variáveis como **nomes para locais de memória**, mas há muito acerca de uma variável do que seu nome, isto é:
  - Nome, endereço, tipo, valor, tempo de vida e escopo.

# Variáveis



- **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:** intervalo no qual a memória permanece alocada para a variável.
- **Escopo:** partes do programa onde a variável é acessível.

# Variáveis: Nome



- Um dos atributos fundamentais das variáveis são os **nomes**.
- Também são associados com subprogramas, parâmetros e outras construções.
- O termo **identificador** é muito usado como sinônimo de nome.
- É uma **cadeia de caracteres** usada para identificar uma entidade no programa.
  - Na maioria das LPs, têm o mesmo **formato**: uma letra seguida por uma cadeia de letras, dígitos e sublinhados (\_).
  - **Sublinhados** foram muito usados de 1970-80, mas hoje isso é menos comum.
  - Nas linguagens baseadas em C, ele foi substituído pelo *camel case*.

# Variáveis: Nome

- **Questões de projeto:**

- Qual deve ser o comprimento máximo permitido?
- Os nomes são sensíveis à capitalização?
- As palavras especiais da LP são palavras reservadas ou palavras-chave?

- **Links relevantes:**

- [https://en.wikipedia.org/wiki/Naming\\_convention\\_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming))
- <http://wiki.c2.com/?CamelCase> e <http://wiki.c2.com/?PascalCase>
- <https://www.theserverside.com/definition/Kebab-case>



# Variáveis: Nome

- Em muitas linguagens (especialmente nas baseadas em C), as letras maiúsculas e minúsculas nos nomes são distintas; ou seja, são sensíveis à **capitalização**.
  - Pode ser um detrimento à **legibilidade**, pois nomes parecidos denotam entidades diferentes.
  - Nesse sentido, a sensibilidade à capitalização viola o princípio de projeto que diz que as construções de linguagem parecidas devem ter significados parecidos.

Contador CONTADOR contador

- Nem todo mundo concorda que a sensibilidade à capitalização é ruim para nomes.
- Problemas da sensibilidade à capitalização são evitados por convenções.

# Variáveis: Nome

## Palavras especiais

- Uma **palavra-chave** é uma palavra de uma linguagem de programação especial apenas em alguns contextos.
  - Em Fortran, a palavra `Integer`, quando encontrada no início de uma sentença e seguida por um nome, é considerada uma palavra-chave que indica que a sentença é **declarativa**.
  - Entretanto, se a palavra `Integer` é seguida por um operador de atribuição, é considerada um **nome de variável**.

```
Integer Apple  
Integer = 4
```

- Compiladores Fortran e pessoas que estejam lendo os programas Fortran devem distinguir entre nomes e palavras especiais pelo contexto.

# Variáveis: Nome

## Palavras especiais

- Uma **palavra reservada** é uma palavra especial de uma linguagem de programação que não pode ser usada como um nome.
  - Como uma escolha de projeto de linguagem, as palavras reservadas são melhores do que as palavras-chave porque a habilidade de redefinir palavras-chave pode ser confusa.
  - Por exemplo, em Fortran, alguém poderia ter escrito as sentenças que declaram a variável de programa chamada `Real` como sendo do tipo `Integer` e a variável chamada `Integer` como sendo do tipo `Real`:

```
Integer Real  
Real Integer
```

- Além da estranha aparência dessas sentenças de declaração, a aparição de `Real` e `Integer` como nomes de variáveis em outros lugares do programa pode causar confusão.

# Variáveis: Endereço

- É o **endereço de memória** associado a uma variável.
  - Uma variável pode ter diferentes endereços de memória durante a execução.
  - Uma variável pode ter diferentes endereços em diferentes lugares de um programa.
    - Mesmo nome de variável em subprogramas distintos
    - Em diferentes momentos da execução do programa. Exemplo: em chamadas recursivas.
  - Pode haver dois nomes para um mesmo endereço (*aliases* ou apelidos)
    - Vantagem: o valor da variável muda com uma atribuição a qualquer um de seus nomes
    - Desvantagem: pode prejudicar a legibilidade.

# Variáveis: Tipo

- Determina a faixa de valores que ela pode armazenar e o conjunto de operações definidas para valores do tipo.
- Exemplo: tipos de dados em Java, com suas respectivas faixa de valores.

Tipos	Primitivo	Valores possíveis		Valor Padrão	Tamanho	Exemplo
		Menor	Maior			
Inteiro	byte	-128	127	0	8 bits	byte ex1 = (byte)1;
	short	-32768	32767	0	16 bits	short ex2 = (short)1;
	int	-2.147.483.648	2.147.483.647	0	32 bits	int ex3 = 1;
	long	-9.223.372.036.854.770.000	9.223.372.036.854.770.000	0	64 bits	long ex4 = 1l;
Ponto Flutuante	float	-1,4024E-37	3.40282347E + 38	0	32 bits	float ex5 = 5.50f;
	double	-4,94E-307	1.79769313486231570E + 308	0	64 bits	double ex6 = 10.20d; ou double ex6 = 10.20;
Caractere	char	0	65535	\0	16 bits	char ex7 = 194; ou char ex8 = 'a';
Booleano	boolean	false	true	false	1 bit	boolean ex9 = true;

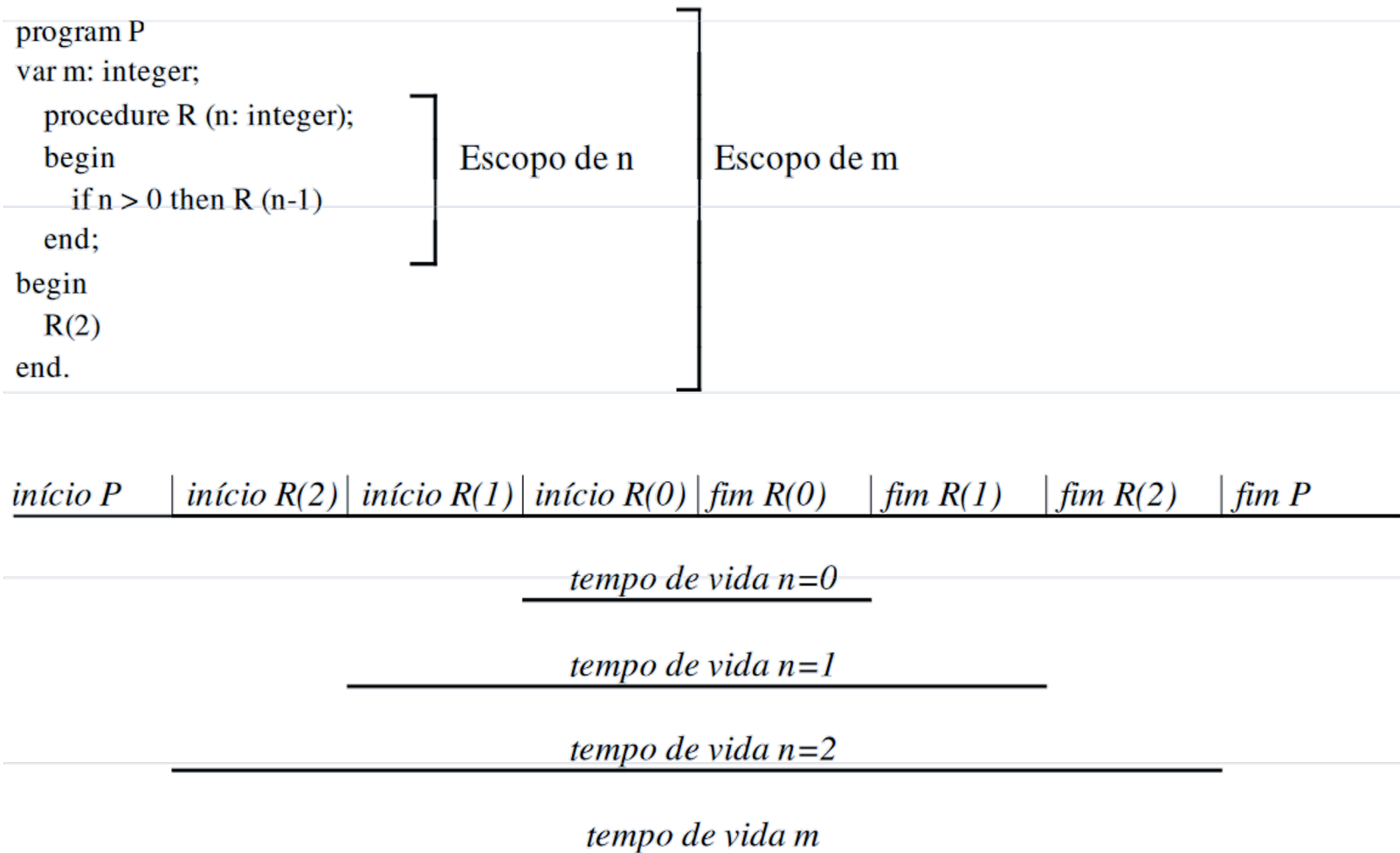
# Variáveis: Valor

- É o **conteúdo** da(s) célula(s) de memória associada(s) a ela.
- Às vezes é chamado de lado direito porque é requerido quando a variável é usada no lado direito de uma sentença de atribuição.
  - Para acessar o lado direito, o lado esquerdo precisa ser determinado.
  - Qual o “valor” da variável  $a$  em  $a := a + 1$ ?
    - Valor-r (right-value ou valor da direita: seu valor)
    - Valor-l (left-value ou valor da esquerda: seu endereço)

# Variáveis: Tempo de vida e Escopo

- **Tempo de vida** é o intervalo entre a criação e a destruição da variável.
  - Variável local: mesmo do bloco onde foi declarada.
  - Variável global: mesmo do programa
- **Escopo** é o trecho do programa em que uma declaração de variável tem efeito.
  - Variável local: bloco onde foi declarada
  - Variável global: todo o programa

# Variáveis: Tempo de vida e Escopo





# Variáveis: Tempo de vida e Escopo

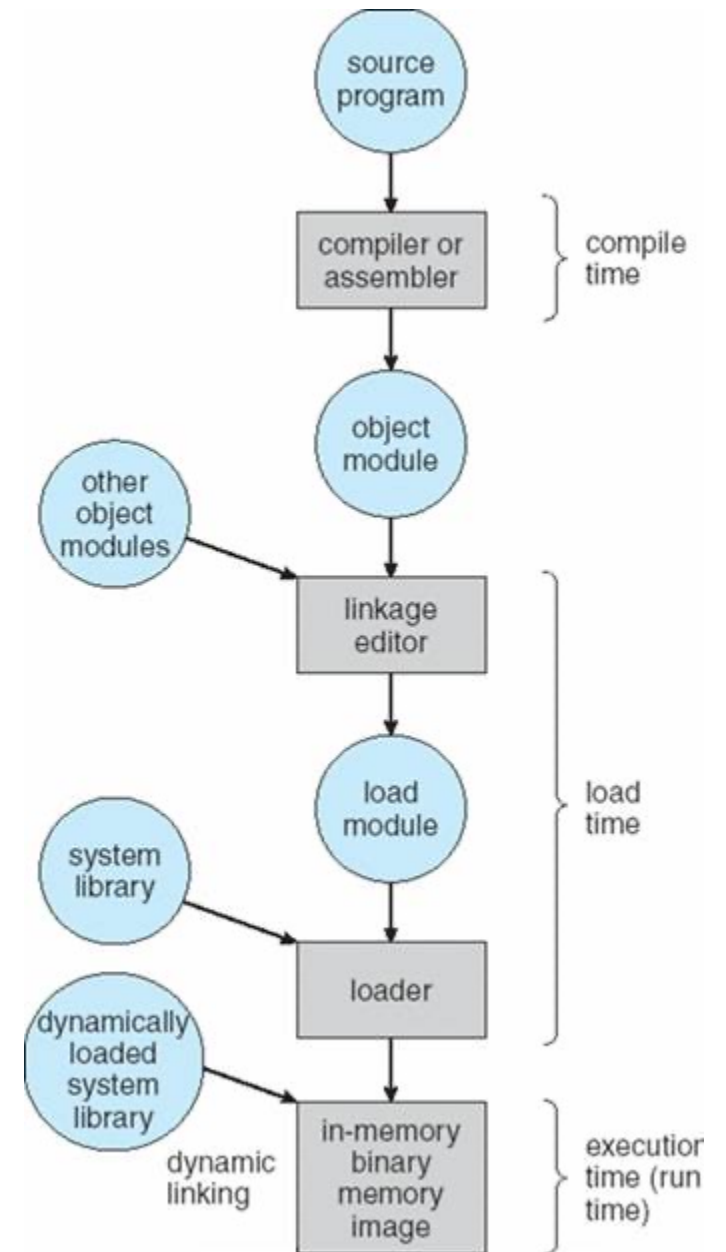
- Variáveis estáticas (static) em C:
  - Variável local com tempo de vida de uma variável global

```
1.  int a = 1;
2.  void f()
3.  {
4.      int b = 1;          // inicializado a cada chamada de f
5.      static int c = a; // inicializado somente uma vez
6.      cout << "a=" << a++ << "b=" << b++ << "c=" << c++ << end;
7.      c = c + 2;
8.  }
9.  int main()
10. {    while (a < 4) f();    }
```

# VINCULAÇÃO

# Vinculação

- Uma **vinculação** é uma associação entre um atributo e uma entidade ou uma operação e um símbolo, por ex.
- O momento no qual uma vinculação ocorre é chamado de **tempo de vinculação**.
- As vinculações podem ocorrer em:
  - tempo de projeto da linguagem;
  - tempo de implementação da linguagem;
  - tempo de compilação;
  - tempo de carga;
  - tempo de ligação;
  - tempo de execução.



# Vinculação

- Considere a seguinte sentença em Java:

```
count = count + 5;
```

- As vinculações e seus tempos de vinculação para as partes dessa sentença são:
  - O tipo de `count` é vinculado em tempo de compilação.
  - O conjunto dos valores possíveis de `count` é vinculado em tempo de projeto do compilador.
  - O significado do símbolo de operador `+` é vinculado em tempo de compilação, quando os tipos dos operandos tiverem sido determinados.
  - A representação interna do literal `5` é vinculada ao tempo de projeto do compilador.
  - O valor de `count` é vinculado em tempo de execução com essa sentença.

# Vinculação

- Tipos de vinculação:
  - Vinculação de atributos a variáveis;
  - Vinculações de tipos (*type binding*);
  - Vinculações de armazenamento (*storage binding*) e tempo de vida (*lifetime*).

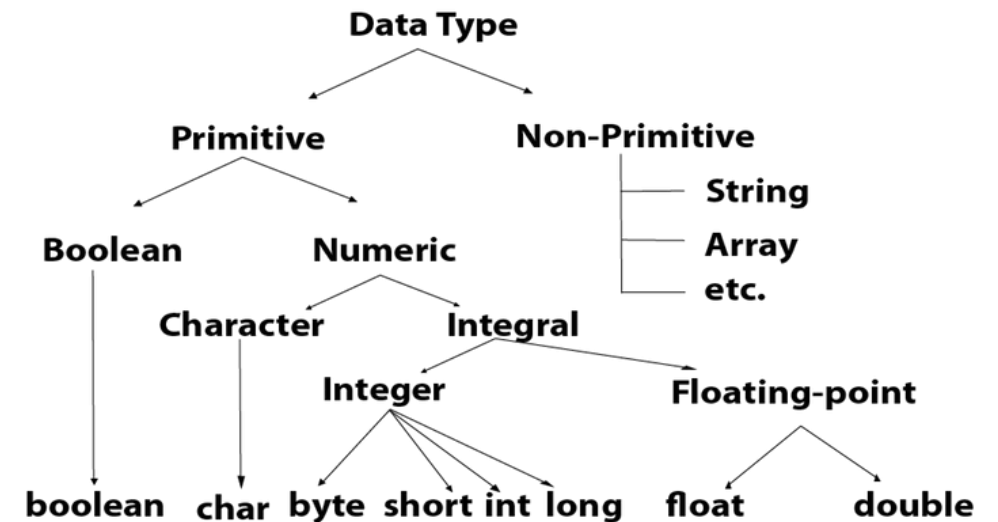


# Vinculação de atributos a variáveis

- Uma vinculação é **estática** se ela ocorre pela primeira vez antes do tempo de execução e permanece intocada ao longo da execução do programa.
- Se a vinculação ocorre pela primeira vez durante o tempo de execução ou pode ser mudada ao longo da execução do programa, é chamada de **dinâmica**.
  - A vinculação física de uma variável a uma célula de armazenamento em um ambiente de memória virtual é complexa, porque a página ou o segmento do espaço de endereçamento no qual a célula reside pode ser movido para dentro ou para fora da memória muitas vezes durante a execução do programa.
  - De certa forma, tais variáveis são vinculadas e desvinculadas repetidamente.

# Vinculação de tipos

- Antes de uma variável poder ser referenciada em um programa, ela deve ser vinculada a um **tipo de dados**.
- Os dois aspectos importantes dessa vinculação são **como** o tipo é especificado e **quando** a vinculação ocorre.
- Os tipos podem ser especificados estaticamente por alguma forma de declaração explícita ou implícita.



*Tipos de dados em Java*

# Vinculação de tipos: estática

- Uma declaração **explícita** é uma sentença em um programa que lista nomes de variáveis e especifica que elas são de um certo tipo.
- Uma declaração **implícita** é uma forma de associar variáveis a tipos por meio de convenções padronizadas, em vez de por sentenças de declaração.
  - Nesse caso, a primeira aparição de um nome de variável constitui sua declaração implícita.
- Tanto declarações explícitas quanto implícitas criam vinculações estáticas a tipos.



# Vinculação de tipos: estática

- A maioria das LPs projetadas desde 1960 requer a declaração explícita de variáveis.
  - Perl, JavaScript, Ruby e ML são algumas exceções.
- Diversas LPs cujos projetos iniciais foram feitos antes do final dos anos 1960 – notavelmente, o Fortran e o BASIC – têm declarações implícitas.
  - Em Fortran, um identificador que aparece em um programa e não é explicitamente declarado é implicitamente declarado de acordo com a seguinte convenção:
    - Se o identificador começar com I, J, K, L, M ou N (maiúsculo ou minúsculo), ele é implicitamente declarado do tipo INTEGER;
    - Caso contrário, é implicitamente declarado do tipo REAL.
    - Link relevante: <http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-decl.htm>

# Vinculação de tipos: estática

- Apesar de serem **convenientes**, as declarações implícitas podem ser **prejudiciais à confiabilidade** pois previnem o processo de compilação de detectar alguns erros de programação e de digitação.
- No Fortran, as variáveis acidentalmente deixadas sem declaração pelo programador recebem **tipos padronizados** e atributos inesperados, que podem causar erros sutis difíceis de ser diagnosticados.
  - Muitos programadores agora incluem a declaração `IMPLICIT NONE` em seus programas.
  - Essa declaração instrui o compilador para **não declarar implicitamente quaisquer variáveis**, evitando os problemas em potencial de acidentalmente termos variáveis não declaradas.
  - Link relevante: <http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-decl.htm>

# Vinculação de tipos: estática

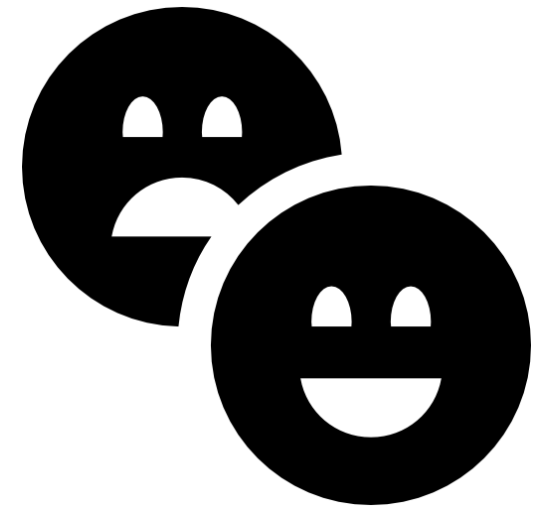
- Alguns dos problemas com declarações implícitas podem ser evitados **obrigando os nomes** para tipos específicos **começarem com caracteres especiais** particulares.
- Por exemplo, em Perl :
  - Qualquer nome que começa com \$ é um escalar, o qual pode armazenar uma cadeia ou um valor numérico.
  - Se um nome começa com @, é um vetor; se começa com %, é uma estrutura de dispersão (hash).
- Nesse caso, os nomes @apple e %apple não são relacionados, porque cada um forma um **espaço de nomes diferente**.
  - Link relevante: <https://perldoc.perl.org/perldata>
- Além disso, um leitor de um programa **sempre sabe o tipo de uma variável quando lê seu nome**.

# Vinculação de tipos: dinâmica

- Com a vinculação de tipos dinâmica, o tipo de uma variável não é especificado por uma sentença de declaração, nem pode ser determinado pelo nome da variável.
  - Em vez disso, a variável é vinculada a um tipo quando é atribuído um valor a ela em uma sentença de atribuição.
  - Quando a sentença de atribuição é executada, a variável que está recebendo um valor atribuído é vinculada ao tipo do valor da expressão no lado direito da atribuição.
- As linguagens nas quais os tipos são vinculados dinamicamente são **drasticamente diferentes** daquelas nas quais os tipos são vinculados estaticamente.
  - A principal da vinculação dinâmica de variáveis a tipos é a maior flexibilidade fornecida ao programador.

# Vinculação de tipos: dinâmica

- Exemplo desta vantagem:
  - Um programa para processar dados numéricos em uma linguagem que usa a vinculação de tipos dinâmica pode ser escrito como um **programa genérico**, ou seja, ele será capaz de tratar dados de quaisquer tipos numéricos.
  - Qualquer tipo de dados informado será aceitável, porque a variável na qual os dados serão armazenados pode ser vinculada ao tipo correto quando o dado for atribuído às variáveis após a entrada.
- Ao contrário, devido à vinculação de tipos estática, não é possível escrever um programa **Java** para processar dados sem conhecer os tipos desses dados.



# Vinculação de tipos: dinâmica

- Em JavaScript e PHP, a vinculação de uma variável a um tipo é dinâmica.

- Por exemplo, um script JavaScript pode conter a seguinte sentença:

```
list = [10.2, 3.5];
```

- Independentemente do tipo anterior da variável chamada `list`, essa atribuição faz com que ela se torne um vetor unidimensional de tamanho 2.
- Se a sentença

```
list = 47;
```

- seguisse a atribuição de exemplo, `list` se tornaria uma variável escalar.

<https://replit.com/@MarceloCysneiro/VinculacaoTiposDinamica>

# Vinculação de tipos: dinâmica

- A seguir, eis algumas **desvantagens** de LPs com vinculação de tipos dinâmica.
- **Confiabilidade baixa**
  - Os programas são menos confiáveis, pois a capacidade **de detecção de erros** do compilador é menor do que a de um compilador para uma linguagem com vinculações de tipo estáticas.
  - Permite valores de quaisquer tipos serem atribuídos a quaisquer variáveis.
  - **Tipos incorretos de lados direitos de atribuições não são detectados como erros**; em vez disso, o tipo do lado esquerdo é trocado para o tipo incorreto.

# Vinculação de tipos: dinâmica

- Por exemplo, suponha que em um programa JavaScript em particular,  $i$  e  $x$  estivessem armazenando valores numéricos escalares, e  $y$  estivesse armazenando um vetor.
- Além disso, suponha que o programa precise da sentença de atribuição

$i = x;$

- mas por causa de um erro de digitação, ele tem a seguinte sentença de atribuição

$i = y;$

- Em JavaScript (ou qualquer outra linguagem que usa vinculação de tipos dinâmica), nenhum erro é detectado nessa sentença pelo interpretador –  $i$  simplesmente se transforma em um vetor.
  - Resultados seguintes de  $i$  esperam que ele seja um escalar, e certamente estarão incorretos.
- Em uma LP com vinculação de tipos estática, como Java, o compilador detectaria o erro na atribuição  $i = y$ , e o programa não seria executado.

<https://replit.com/@MarceloCysneiro/VinculacaoTiposDinamica>



# Vinculação de tipos: dinâmica

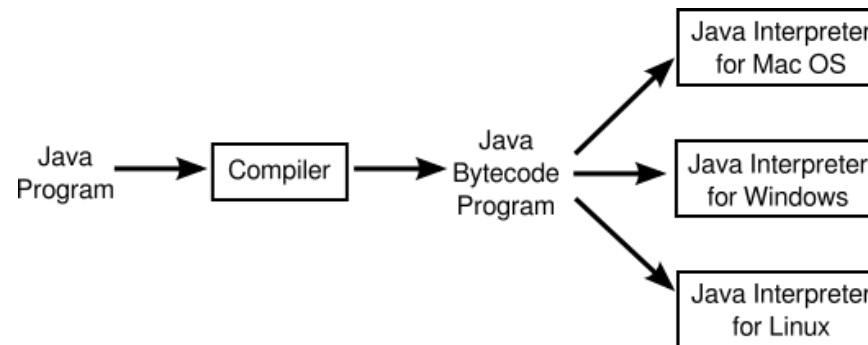
- **Custo elevado**
  - A vinculação de atributos dinâmica é custoso, principalmente em tempo de execução.
  - A **verificação de tipos** deve ser feita em tempo de execução.
  - Além disso, cada variável deve ter um **descritor** em tempo de execução associado a ela de forma a manter o tipo atual.
  - O **armazenamento** usado para o valor de uma variável deve ser de **tamanho variável**, porque valores de tipos diferentes precisam de quantidades distintas de armazenamento.

# Vinculação de tipos: dinâmica

- São usados interpretadores puros, em vez de compiladores.
  - Os computadores não contêm instruções cujos tipos dos operandos não são conhecidos em tempo de compilação.
  - Logo, um compilador não pode construir instruções de máquina para a expressão  $A + B$  se os tipos  $A$  e  $B$  não são conhecidos em tempo de compilação.
  - A interpretação pura tipicamente leva ao menos 10 vezes mais tempo para executar um código de máquina equivalente.

# Vinculação de tipos: dinâmica

- Ao usar um interpretador puro, o tempo para realizar a **vinculação de tipos dinâmica** é ocultado pelo tempo total da interpretação, assim, tal vinculação **parece ser menos cara**.



- Por outro lado, as linguagens com **vinculações de tipo estáticas** são raramente implementadas pela interpretação pura, pois os programas nessas linguagens podem ser facilmente traduzidos para versões em **código de máquina muito eficientes**.

# Vinculação de tipos: exemplo de inferência

- ML é uma LP que oferece suporte para programação funcional e imperativa, cujo mecanismo de **inferência de tipos** permite que os tipos da maioria das expressões sejam determinados **sem que o programador especifique os tipos das variáveis**.
- Eis a sintaxe geral de uma função ML:

**fun** nome\_função(parâmetros formais) = expressão

- O valor da expressão é retornado pela função.

# Vinculação de tipos: exemplo de inferência

- Exemplo 1:

```
fun circumf(r) = 3.14159 * r * r;
```

- A expressão especifica uma função que recebe um ponto flutuante (**real** em ML) como argumento e produz um resultado como ponto flutuante.
- Os tipos são inferidos pelo tipo da constante na expressão.

- Exemplo 2:

```
fun times10(x) = 10 * x;
```

- Nela, o argumento e o valor funcional são inferidos como sendo do tipo **int**.

# Vinculação de tipos: exemplo de inferência

- Exemplo 3:

```
fun square(x) = x * x;
```

- ML determina o tipo tanto do parâmetro quanto do valor de retorno a partir do operador `*` na definição da função.
- Como ele é um operador aritmético, assume-se que o tipo do parâmetro e o tipo da função sejam numéricos (o tipo numérico padrão é `int`).
- Então, infere-se que o tipo do parâmetro e do valor de retorno seja `int`.

# Vinculação de tipos: exemplo de inferência

- Se `square` fosse chamada com um valor de ponto flutuante, isso causaria um erro, porque ML não realiza coerção de valores do tipo **real** para o tipo **int**:

```
square( 2.75 );
```

- Se quiséssemos que `square` aceitasse parâmetros do tipo **real**, ela poderia ser reescrita como:

```
fun square(x) : real = x * x;
```

- Como ML não permite funções sobrecarregadas, essa versão não poderia coexistir com a versão anterior baseada em **int**.

# Vinculação de tipos: exemplo de inferência

- O fato de o valor funcional ser tipado como **real** é suficiente para inferir que o parâmetro também é do tipo **real**.
- Cada uma das definições a seguir também é válida:

```
fun square(x : real) = x * x;
```

```
fun square(x) = (x : real) * x;
```

```
fun square(x) = x * (x : real);
```

- A inferência de tipos também é usada nas linguagens puramente funcionais Miranda e Haskell.



# Vinculações de armazenamento e tempo de vida

- **Alocação** é o processo de **obtenção da célula de memória** à qual uma variável será vinculada a partir de um conjunto de células disponíveis.
- **Liberação** é o processo de **devolver uma célula de memória** que foi desvinculada de uma variável ao conjunto de células disponíveis.

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>

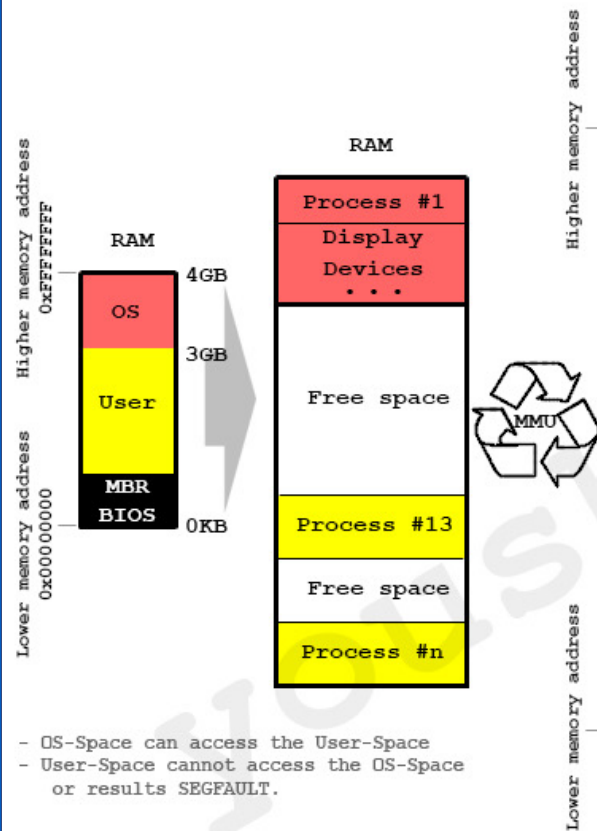
# Vinculações de armazenamento e tempo de vida

- O **tempo de vida** de uma variável é o intervalo durante o qual ela está vinculada a uma posição específica da memória.
- Ele começa quando a variável é **vinculada** a uma célula específica e termina quando ela é **desvinculada** dessa célula.



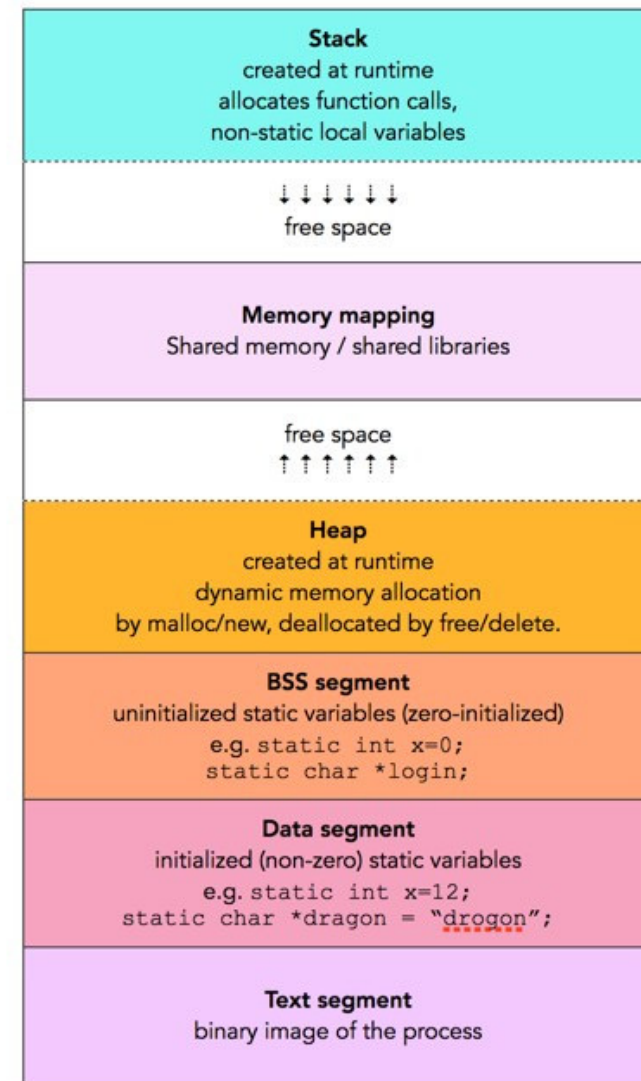
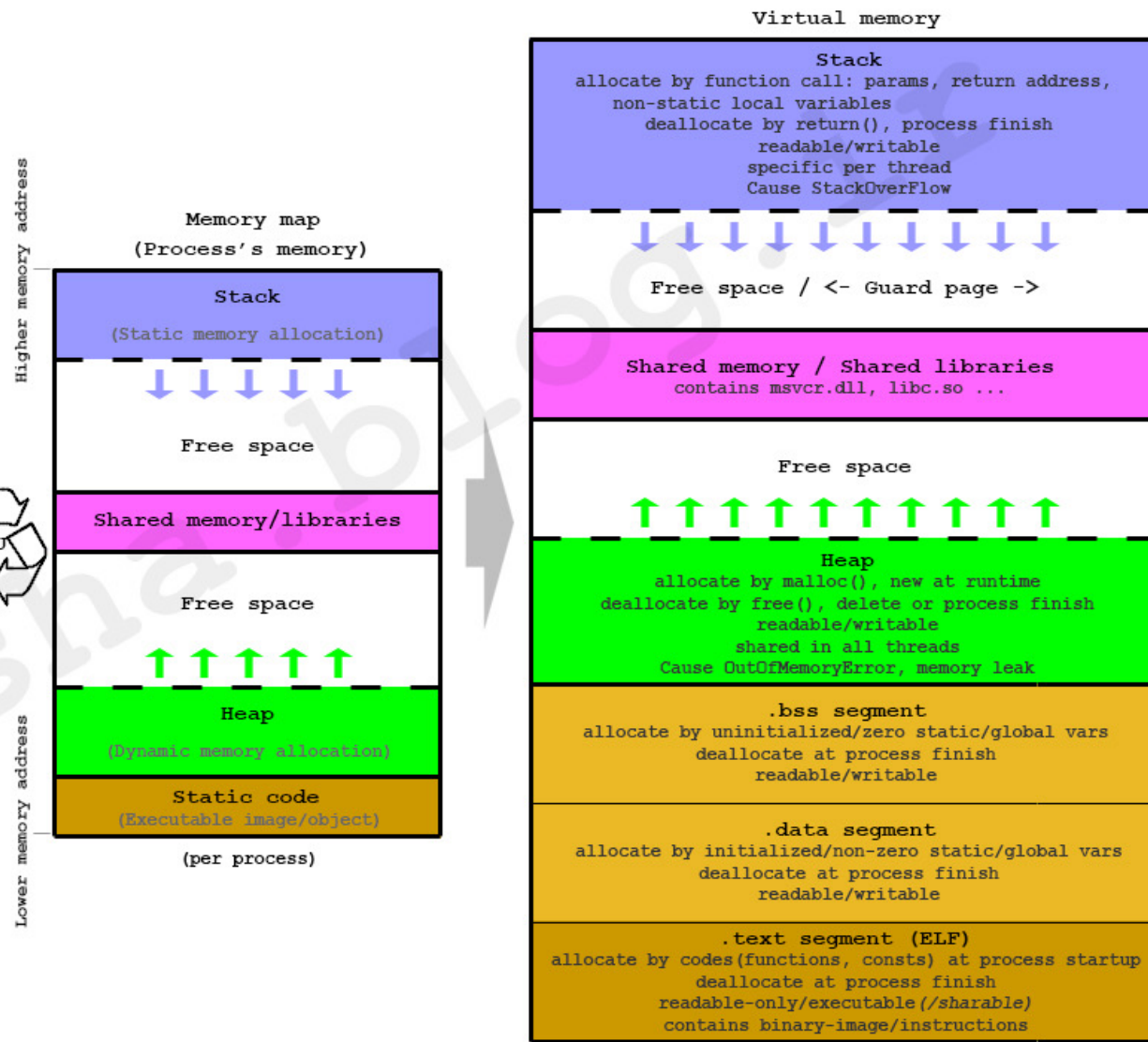
# Vinculações de armazenamento e tempo de vida

- É conveniente separar variáveis escalares (não estruturadas) em quatro categorias, de acordo com seus tempos de vida:
  - Estáticas (*static*);
  - Dinâmicas da pilha (*stack-dynamic*);
  - Dinâmicas do monte explícitas (*explicit heap-dynamic*);
  - Dinâmicas do monte implícitas (*implicit heap-dynamic*).
- <https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap>



- OS-Space can access the User-Space
- User-Space cannot access the OS-Space or results SEGVFAULT.

(c) yousha.blog.ir - Iran



# Vinculações de armazenamento e tempo de vida: variáveis estáticas

- Vinculadas a células de memória **antes do início da execução** de um programa
- Permanecem vinculadas a essas mesmas células **até que a execução termine**.
- Exemplos:
  - Variáveis `static` em funções em C/C++.
- Vantagem(ns):
  - **Eficiência** (endereçamento aberto / *hashing* fechado);
  - Suporte a subprogramas **sensíveis a histórico**.
- Desvantagem(ns):
  - Redução da flexibilidade (não permite o uso de **subprogramas recursivos**).
- <https://craftofcoding.wordpress.com/2017/12/01/where-is-static-memory-stored-in-c/>

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas da pilha (*stack-dynamic*)

- São aquelas cujas **vinculações de armazenamento são criadas quando suas sentenças de declaração são elaboradas**, mas cujos **tipos são estaticamente vinculados**.
  - A **elaboração** de tal declaração se refere à alocação do armazenamento e ao processo de vinculação indicado pela declaração, que ocorre quando a execução alcança o código com o qual a declaração está anexada.
- Logo, a elaboração ocorre apenas em **tempo de execução**.
- Por exemplo, as declarações de **variáveis que aparecem no início de um método Java** **são elaboradas quando o método é chamado** e as variáveis definidas por essas declarações **são liberadas quando o método completa sua execução**.

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas da pilha (*stack-dynamic*)

- Exemplos:
  - Em Java, C++ e C#, as variáveis definidas em métodos são, por padrão, dinâmicas da pilha.
  - Em Ada, todas as variáveis que não são do monte e são definidas em subprogramas são dinâmicas da pilha.

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas da pilha (*stack-dynamic*)

- Vantagem(ns):
  - Permite **recursão** (cada cópia ativa do subprograma recursivo tem sua própria versão das variáveis locais);
  - Subprogramas **compartilham o espaço de memória** para suas variáveis locais.
- Desvantagem(ns):
  - Sobrecarga em **tempo** de execução da alocação e liberação;
  - Acessos mais lentos em função do **endereçamento indireto** necessário;
  - Os subprogramas não podem ser sensíveis ao **histórico de execução**.



# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas do monte explícitas (*explicit heap-dynamic*)

- São células de memória não nomeadas (abstratas) alocadas e liberadas por **instruções explícitas** em tempo de execução pelo programador.
- Só podem ser referenciadas por **ponteiros ou variáveis de referência**.
- Exemplos:
  - Objetos dinâmicos em C++ (via **new** e **delete**);
  - Todos os objetos em Java.

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas do monte explícitas (*explicit heap-dynamic*)

- Vantagem(ns):
  - Úteis para a **construção de estruturas dinâmicas**, como listas ligadas e árvores, que precisam crescer e/ou diminuir durante a execução.
- Desvantagem(ns):
  - **Dificuldade de usar** ponteiros e variáveis de referência corretamente;
  - **Custo de referências** às variáveis;
  - Complexidade da implementação do **gerenciamento de armazenamento**.

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas do monte explícitas (*explicit heap-dynamic*)

- Exemplo em C++:

```
int *intnode;      // Cria um ponteiro
intnode = new int; // Cria a variável dinâmica do monte ..
delete intnode;    // Libera a variável dinâmica do monte
                  // para qual intnode aponta
```

- Aqui, uma variável dinâmica do monte explícita do tipo **int** é criada pelo operador **new**.
- Essa variável pode então ser referenciada por meio do ponteiro `intnode`.
- Posteriormente, a variável é liberada pelo operador **delete**.
- C++ requer o operador de liberação explícita **delete**, porque a linguagem não usa recuperação implícita de armazenamento, como a coleta de lixo ([\*garbage collection\*](#)).

# Vinculações de armazenamento e tempo de vida: variáveis dinâmicas do monte implícitas (*implicit heap-dynamic*)

- Vinculadas ao armazenamento no monte apenas quando são atribuídos valores a elas.
- Exemplo(s):
  - Todas as variáveis em APL;
  - Todas as *strings* e vetores em Perl, JavaScript e PHP.
- Vantagem(ns):
  - Flexibilidade (permite a escrita de código altamente genérico).
- Desvantagem(ns):
  - Ineficiente, pois todos os atributos são dinâmicos (tipos, faixas de índices de vetores, etc);
  - Perda da detecção de erros pelo compilador.

# ESCOPO

# Escopo

- O **escopo** de uma variável é a faixa de sentenças nas quais ela é visível.
- Uma variável é **visível** em uma sentença se puder ser referenciada nessa sentença.
- Uma variável é **local** a uma unidade / um bloco de programa se for declarada lá.
- Variáveis **não locais** de uma unidade ou de um bloco de programa são aquelas visíveis dentro da unidade ou do bloco de programa, mas não declaradas nessa unidade ou nesse bloco.

# Escopo estático

- O ALGOL 60 introduziu o método de vincular nomes a variáveis não locais, chamado de **escopo estático**, copiado por muitas linguagens imperativas subsequentes (e por muitas linguagens não imperativas).
- O escopo estático é chamado assim porque o escopo de uma variável pode ser determinado estaticamente – ou seja, antes da execução.
- Isso permite a um leitor de programas humano (e um compilador) determinar o tipo de cada variável.

# Escopo estático

- Para conectar uma referência de nome a uma variável, você (ou o compilador) deve encontrar a declaração.
- **Processo de busca:** declarações de pesquisa, primeiro localmente, depois em escopos cada vez maiores, até que uma seja encontrada para o nome fornecido
- Escopos estáticos que envolvem um escopo específico são chamados de seus **ancestrais estáticos**; o ancestral estático mais próximo é chamado de **pai estático**.



# Escopo estático

- Veja o seguinte procedimento em Ada ao lado, no qual estão aninhados os procedimentos Sub1 e Sub2:
  - De acordo com o escopo estático, **a referência à variável X em Sub2** é para o X declarado no procedimento Big.
  - Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá.
  - A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada.
  - O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```

# Escopo estático

- Veja o seguinte procedimento em Ada ao lado, no qual estão aninhados os procedimentos Sub1 e Sub2:
  - De acordo com o escopo estático, a referência à variável X em Sub2 é para o X declarado no procedimento Big.
  - Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá.
  - A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada.
  - O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
  begin -- de Sub1
    ...
  end; -- de Sub1
  procedure Sub2 is
  begin -- de Sub2
    ...X...
  end; -- de Sub2
begin -- de Big
  ...
end; -- de Big
```

# Escopo estático

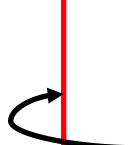
- Veja o seguinte procedimento em Ada ao lado, no qual estão aninhados os procedimentos Sub1 e Sub2:
  - De acordo com o escopo estático, a referência à variável X em Sub2 é para o X declarado no procedimento Big.
  - **Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá.**
  - A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada.
  - O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
  begin -- de Sub1
    ...
  end; -- de Sub1
  procedure Sub2 is
  begin -- de Sub2
    ...X...
  end; -- de Sub2
begin -- de Big
  ...
end; -- de Big
```

# Escopo estático

- Veja o seguinte procedimento em Ada ao lado, no qual estão aninhados os procedimentos Sub1 e Sub2:
  - De acordo com o escopo estático, a referência à variável X em Sub2 é para o X declarado no procedimento Big.
  - Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá.
  - **A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada.**
  - O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.

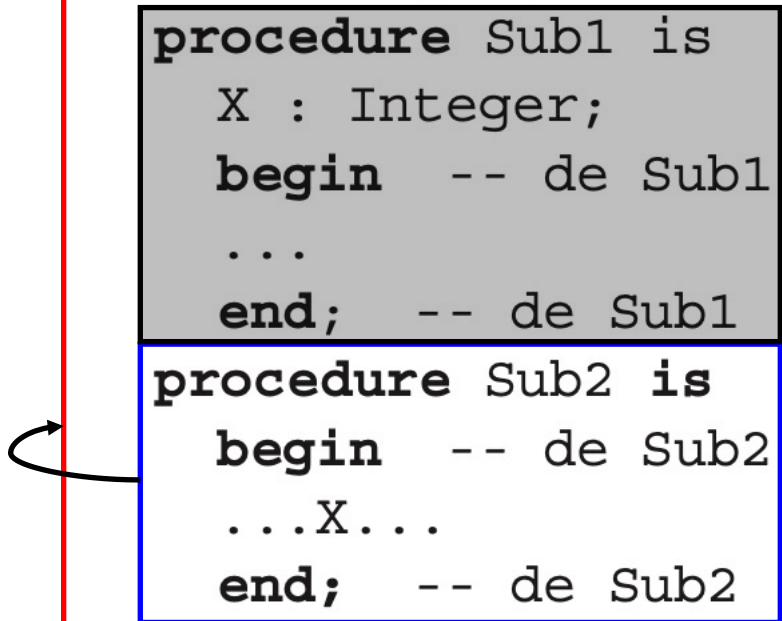
```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
  begin -- de Sub1
    ...
  end; -- de Sub1
  procedure Sub2 is
  begin -- de Sub2
    ...X...
  end; -- de Sub2
begin -- de Big
  ...
end; -- de Big
```



# Escopo estático

- Veja o seguinte procedimento em Ada ao lado, no qual estão aninhados os procedimentos Sub1 e Sub2:
  - De acordo com o escopo estático, a referência à variável X em Sub2 é para o X declarado no procedimento Big.
  - Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá.
  - A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada.
  - **O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.**

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```



# Escopo estático

- Algumas linguagens permitem definições de subprogramas aninhados, que criam escopos estáticos aninhados (ex.: Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F# e Python).
- Exemplo em C (é permitido): <https://replit.com/@MarceloCysneiro/EscopoEstaticoAninhadoC>

```
> clang-7 -pthread -lm -o main main.c
> ./main
i = 17
i = 42
i = 42
i = 42
> 
```

# Escopo estático

- Algumas linguagens permitem definições de subprogramas aninhados, que criam escopos estáticos aninhados (ex.: Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F# e Python).
- Exemplo em Java (não é permitido): <https://replit.com/@MarceloCysneiro/EscopoEstaticoAninhadoJava>

```
➤ javac -classpath ./run_dir/junit-4.12.jar:target/dependency/* -d . Main.java
Main.java:11: error: variable i is already defined in method main(String[])
        int i = 42;
            ^
1 error
compiler exit status 1
➤
```

# Escopo estático

- Em LPs que usam escopo estático, independentemente de ser permitido o uso de subprogramas aninhados ou não, algumas declarações de variáveis podem ser ocultadas de outros segmentos de código.
- Considere mais uma vez o procedimento Big em Ada.
  - A variável X é declarada **tanto em Big quanto em Sub1**, aninhado dentro de Big.
  - Dentro de Sub1, cada referência simples para X é para o X local.
  - Logo, o X externo está oculto de Sub1.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
      ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
      ...X...
    end; -- de Sub2
  begin -- de Big
    ...
  end; -- de Big
```



# Escopo estático

- Em LPs que usam escopo estático, independentemente de ser permitido o uso de subprogramas aninhados ou não, algumas declarações de variáveis podem ser ocultadas de outros segmentos de código.
- Considere mais uma vez o procedimento Big em Ada.
  - A variável X é declarada tanto em Big quanto em Sub1, aninhado dentro de Big.
  - **Dentro de Sub1, cada referência simples para X é para o X local.**
  - Logo, o X externo está oculto de Sub1.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
      ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
      ...X...
    end; -- de Sub2
  begin -- de Big
    ...
  end; -- de Big
```

# Escopo estático

- Em LPs que usam escopo estático, independentemente de ser permitido o uso de subprogramas aninhados ou não, algumas declarações de variáveis podem ser ocultadas de outros segmentos de código.
- Considere mais uma vez o procedimento Big em Ada.
  - A variável X é declarada tanto em Big quanto em Sub1, aninhado dentro de Big.
  - Dentro de Sub1, cada referência simples para X é para o X local.
  - Logo, o X externo está oculto de Sub1.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
      ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
      ...X...
    end; -- de Sub2
  begin -- de Big
    ...
  end; -- de Big
```

# Blocos

- A definição de novos escopos estáticos, introduzida no ALGOL 60, permite uma seção de código ter suas próprias variáveis locais, cujo escopo é minimizado.
  - Tais variáveis são dinâmicas da pilha, de forma que seu armazenamento é alocado quando a seção é alcançada e liberado quando a seção é abandonada.
- Tais seções de código são chamadas de **blocos**; daí a origem da frase **linguagem estruturada em blocos**.



# Blocos

- Em geral, uma declaração de uma variável efetivamente esconde quaisquer declarações de variáveis com o mesmo nome no escopo externo maior.
- Isto é chamado de **sombreamento de variável** (*variable shadowing*).
  - <https://stackoverflow.com/questions/53734399/what-is-variable-shadowing>
- Em C++ tais variáveis globais ocultas podem ser acessadas no interno usando o operador de escopo (`::`).
  - <https://www.geeksforgeeks.org/scope-resolution-operator-in-c/>



# Ordem de declaração

- C99, C++, Java e C# permitem que declarações de variáveis apareçam em qualquer lugar em que uma instrução possa aparecer.
  - Em C99, C++ e Java, o escopo de todas as variáveis locais é desde a declaração até o final do bloco.
  - Em C #, o escopo de qualquer variável declarada em um bloco é o bloco inteiro, independentemente da posição da declaração no bloco
    - No entanto, uma variável ainda deve ser declarada antes de poder ser usada
- Em C++, Java e C#, variáveis podem ser declaradas em instruções `for`
  - O escopo de tais variáveis é restrito à construção `for`.
  - Exemplo: <https://www.geeksforgeeks.org/scope-of-variables-in-c-sharp/>

# Escopo global

- Algumas linguagens permitem uma estrutura de programa que é uma sequência de definição de funções, nas quais as definições de variáveis podem aparecer fora das funções.
- Exemplos:
  - C e C++
  - PHP
  - Python
- Definições fora de funções em um arquivo criam variáveis globais, potencialmente visíveis a essas funções.

# Escopo global

- C e C++ têm tanto **declarações** quanto **definições** de dados globais.
  - Declarações especificam tipos e outros atributos, mas não causam a alocação de armazenamento.
  - As definições especificam atributos e causam a alocação de armazenamento.
- Para um nome global específico, um programa em C pode ter **qualquer número de declarações** compatíveis, mas **apenas uma definição**.
- Uma declaração de uma variável fora das definições de funções especifica que ela é definida em um arquivo diferente.
  - <https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/>

# Escopo global

- Uma variável global em C é implicitamente visível em todas as funções subsequentes no arquivo, **exceto aquelas que incluem uma declaração de uma variável local com o mesmo nome.**
  - Uma variável global definida após uma função pode ser tornada visível na função declarando-a como externa, como:  
**`extern int sum;`**
- Em C99, as definições de variáveis globais sempre têm valores iniciais, mas as declarações de variáveis globais nunca têm.



# Escopo global

- Essa ideia de declarações e definições é usada também para funções em C e C++, onde os **protótipos** declaram nomes e interfaces de funções, mas não fornecem seu código.
- As definições de funções, em contrapartida, são completas.
- Em C++, uma variável global oculta por uma local com o mesmo nome pode ser acessada usando o operador de escopo (::).

# Escopo global

- O caso do PHP
  - Programas em PHP são geralmente embutidos em documentos XHTML.
    - Exemplo: `example_intern.html` e `example_extern.html`
  - Independentemente se estiverem embutidos em XHTML ou em arquivos próprios, os programas em PHP são puramente interpretados.
  - Sentenças podem ser interpoladas com definições de funções.
  - Quando encontradas, as sentenças são interpretadas; as definições de funções são armazenadas para referências futuras.
  - As variáveis em PHP são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição.

# Escopo global

- Exemplo ([calendar.php](#) ou <https://replit.com/@MarceloCysneiro/CalendarPHP>)
  - Qualquer variável implicitamente declarada fora de qualquer função é global; variáveis implicitamente declaradas em funções são variáveis locais.
  - O escopo das variáveis globais se estende de suas declarações até o fim do programa, mas pulam sobre quaisquer definições de funções subsequentes.
  - Logo, **variáveis globais não são implicitamente visíveis em nenhuma função.**
  - Variáveis globais podem ser tornadas visíveis em funções de duas formas:
    - Se a função inclui uma variável local com o mesmo nome da global, esta pode ser acessada por meio do vetor \$GLOBALS, usando o nome da variável como o índice do vetor.
    - Se não existe uma variável local na função com o mesmo nome da global, esta pode se tornar visível com sua inclusão em uma sentença de declaração global.

# Escopo global

- O caso do Python
  - As regras de visibilidade para variáveis globais em Python não são usuais.
  - As variáveis não são normalmente declaradas, como em PHP.
  - Elas são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição.
  - Uma variável global pode ser referenciada em uma função, mas uma variável global pode ter valores atribuídos a ela apenas se tiver sido declarada como global na função.

# Escopo global

- Exemplo (`calendar.py` ou <https://replit.com/@MarceloCysneiro/CalendarPython>)

```
day = 'Monday'
def test():
    print('The global day is:', day)
tester()
```

- A saída desse *script*, como as variáveis globais podem ser referenciadas diretamente nas funções, é:

```
The global day is: Monday
```

# Escopo global

- O *script* a seguir tenta atribuir um novo valor a variável global day:

```
day = 'Monday'
def test():
    print 'The global day is:', day
    day = 'Tuesday'
    print 'The new value of day is:', day
tester()
```

- O *script* cria uma mensagem de erro do tipo `UnboundLocalError`, porque a atribuição a day na segunda linha do corpo da função torna day uma variável local – o que faz a referência a day na primeira linha do corpo da função se tornar uma referência ilegal para a variável local.

# Escopo global

- A atribuição a `day` pode ser para a variável global se `day` for declarada como global no início da função. Isso previne que a atribuição de valores a `day` crie uma variável local, como é mostrado no *script* a seguir:

```
day = 'Monday'
def test():
    global day
    print('The global day is:', day)
    day = 'Tuesday'
    print('The new value of day is:', day)
test()
```

- A saída desse *script* é:  
The global day is: Monday  
The new value of day is: Tuesday

Bônus: [nonlocal](#)

# Avaliação do escopo estático

- Vantagem:
  - Funciona bem em muitas situações.
- Desvantagens:
  - Na maioria dos casos, acesso excessivo é possível.
  - À medida que um programa evolui, a estrutura inicial é destruída e as variáveis locais freqüentemente se tornam globais; subprogramas também tendem a se tornar globais, ao invés de aninhados.



# Escopo dinâmico

- É baseado na sequência de chamadas de subprogramas, não em seu relacionamento espacial uns com os outros.
- Logo, o escopo pode ser determinado apenas em tempo de execução.
- LPs com escopo dinâmico:
  - Primeiras versões de LISP (1958)
  - APL (1962)
  - SNOBOL 4 (1962)
  - Perl\* (1987) e COMMON LISP\* (1984 / 1994)
    - permitem que variáveis sejam declaradas com escopo dinâmico, mas o mecanismo padrão é estático.

# Escopo dinâmico

- Considere mais uma vez o procedimento Big.
- Assuma que as regras de **escopo dinâmico** se aplicam a referências não locais.
- O significado do identificador X referenciado em Sub2 é dinâmico – ele não pode ser determinado em tempo de compilação.
- Ele pode referenciar a qualquer uma das declarações de X, dependendo da sequência de chamadas.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
      ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
      ...X...
    end; -- de Sub2
  begin -- de Big
    ...
  end; -- de Big
```

# Escopo dinâmico

- O significado correto de X pode ser determinado em tempo de execução é iniciar a busca com as **variáveis locais**.
  - Essa é a maneira pela qual o processo começa no **escopo estático**, mas é aqui que a similaridade entre eles termina.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```

# Escopo dinâmico

- O significado correto de X pode ser determinado em tempo de execução é iniciar a busca com as variáveis locais.
  - Essa é a maneira pela qual o processo começa no escopo estático, mas é aqui que a similaridade entre eles termina.
- Quando a busca por declarações locais falha, as declarações do **pai dinâmico (o procedimento que o chamou)** são procuradas.
  - Se uma declaração para X não é encontrada lá, a busca continua no pai dinâmico desse procedimento chamador, e assim por diante, até que uma declaração de X seja encontrada.
  - Se nenhuma for encontrada em nenhum ancestral dinâmico, ocorre um **erro em tempo de execução (runtime error)**.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```

# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- **Primeiro, Big chama Sub1, que chama Sub2.**

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```

# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- **Primeiro, Big chama Sub1, que chama Sub2.**
  - Nesse caso, a busca continua a partir do procedimento local, Sub2, para seu chamador, Sub1, onde uma declaração de X é encontrada.

```
procedure Big is  
  X : Integer;  
  procedure Sub1 is  
    X : Integer;  
    begin -- de Sub1  
    ...  
    end; -- de Sub1  
  procedure Sub2 is  
    begin -- de Sub2  
    ...X...  
    end; -- de Sub2  
  begin -- de Big  
  ...  
  end; -- de Big
```

# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- **Primeiro, Big chama Sub1, que chama Sub2.**
  - Nesse caso, a busca continua a partir do procedimento local, Sub2, para seu chamador, Sub1, onde uma declaração de X é encontrada.
  - Logo, a referência a X em Sub2, nesse caso, é para o X declarado em Sub1.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub2 is
    begin -- de Sub2
    ...X...
    end; -- de Sub2
  begin -- de Big
  ...
  end; -- de Big
```

# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- Primeiro, Big chama Sub1, que chama Sub2.
  - Nesse caso, a busca continua a partir do procedimento local, Sub2, para seu chamador, Sub1, onde uma declaração de X é encontrada.
  - Logo, a referência a X em Sub2, nesse caso, é para o X declarado em Sub1.
- A seguir, Sub2 é chamado diretamente por Big.

```
procedure Big is  
  X : Integer;  
  procedure Sub1 is  
    X : Integer;  
    begin -- de Sub1  
    ...  
    end; -- de Sub1  
  procedure Sub2 is  
    begin -- de Sub2  
    ...X...  
    end; -- de Sub2  
  begin -- de Big  
  ...  
  end; -- de Big
```



# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- Primeiro, Big chama Sub1, que chama Sub2.
  - Nesse caso, a busca continua a partir do procedimento local, Sub2, para seu chamador, Sub1, onde uma declaração de X é encontrada.
  - Logo, a referência a X em Sub2, nesse caso, é para o X declarado em Sub1.
- **A seguir, Sub2 é chamado diretamente por Big.**
  - Nesse caso, o pai dinâmico de Sub2 é Big, e a referência é para o X declarado em Big.

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
  begin -- de Sub1
    ...
  end; -- de Sub1
  procedure Sub2 is
  begin -- de Sub2
    ...X...
  end; -- de Sub2
begin -- de Big
  ...
end; -- de Big
```

# Escopo dinâmico

- Considere duas sequências de chamadas diferentes para Sub2.
- Primeiro, Big chama Sub1, que chama Sub2.
  - Nesse caso, a busca continua a partir do procedimento local, Sub2, para seu chamador, Sub1, onde uma declaração de X é encontrada.
  - Logo, a referência a X em Sub2, nesse caso, é para o X declarado em Sub1.
- A seguir, Sub2 é chamado diretamente por Big.
  - Nesse caso, o pai dinâmico de Sub2 é Big, e a referência é para o X declarado em Big.
- Note que se o escopo estático fosse usado, em qualquer uma das sequências de chamadas discutidas, a referência a X em Sub2 seria o X de Big.

```
procedure Big is  
  X : Integer;  
  procedure Sub1 is  
    X : Integer;  
    begin -- de Sub1  
    ...  
    end; -- de Sub1  
  procedure Sub2 is  
    begin -- de Sub2  
    ...X...  
    end; -- de Sub2  
  begin -- de Big  
  ...  
  end; -- de Big
```

# Avaliação do escopo dinâmico

- O efeito do escopo dinâmico na programação é profundo, tais como:
  - Os atributos corretos das variáveis não locais visíveis a uma sentença de um programa não podem ser determinados estaticamente.
  - Uma referência ao nome de tal variável nem sempre é para a mesma.
  - Uma sentença em um subprograma que contém uma referência para uma variável não local pode se referir a diferentes variáveis não locais durante diferentes execuções do subprograma.
- Diversos problemas podem aparecer por causa do escopo dinâmico.

# Avaliação do escopo dinâmico

- Vantagem:
  - Conveniência (não há necessidade de passar parâmetros de um subprograma para o outro, já que as variáveis do chamador são implicitamente visíveis).
- Desvantagens:
  - Enquanto um subprograma está sendo executado, suas variáveis são visíveis para todos os subprogramas que ele chama;
  - Impossível realizar a verificação de tipos estática;
  - Baixa legibilidade, pois não é possível determinar estaticamente o tipo de uma variável.

# Ambientes de referenciamento

- O **ambiente de referenciamento** de uma sentença é a coleção de todas as variáveis visíveis na sentença.
- Em uma linguagem de escopo estático, ele é composto por:

variáveis declaradas em seu escopo local

+

todas as variáveis de seus escopos ancestrais visíveis

# Ambientes de referenciamento

- Considere o seguinte programa de exemplo.
- Vamos determinar os ambientes de referenciamento para cada um dos **pontos (1, 2, 3 e 4)** destacados.

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- de Sub1
    ... <----- 1
    end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin -- de Sub3
      ... <----- 2
      end; -- de Sub3
    begin -- de Sub2
    ... <----- 3
    end; -- de Sub2
  begin -- de Example
  ... <----- 4
  end. -- de Example
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:

- Ponto:

- 1

- Ambiente de referenciamento:

- X e Y de Sub1, A e B de Example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- de Sub1
    ... <----- 1
    end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin -- de Sub3
      ... <----- 2
      end; -- de Sub3
    begin -- de Sub2
    ... <----- 3
    end; -- de Sub2
  begin -- de Example
  ... <----- 4
  end. -- de Example
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:
  - Ponto:
    - 2
  - Ambiente de referenciamento:
    - X de Sub3, (X de Sub2 está oculto),  
A e B de Example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- de Sub1
    ... <----- 1
  end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin -- de Sub3
      ... <----- 2
      end; -- de Sub3
    begin -- de Sub2
    ... <----- 3
    end; -- de Sub2
  begin -- de Example
  ... <----- 4
end. -- de Example
```



# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:
  - Ponto:
    - 3
  - Ambiente de referenciamento:
    - X de Sub2, A e B de Example

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- de Sub1
    ... <----- 1
  end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin -- de Sub3
      ... <----- 2
    end; -- de Sub3
  begin -- de Sub2
  ... <----- 3
  end; -- de Sub2
begin -- de Example
... <----- 4
end. -- de Example
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:
  - Ponto:
    - 4
  - Ambiente de referenciamento:
    - A e B de *Example*

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin -- de Sub1
    ... <----- 1
    end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin -- de Sub3
      ... <----- 2
      end; -- de Sub3
    begin -- de Sub2
    ... <----- 3
    end; -- de Sub2
  begin -- de Example
  ... <----- 4
  end. -- de Example
```

# Ambientes de referenciamento

- Um subprograma está **ativo** se sua execução já tiver começado, mas não terminado ainda.
- O ambiente de referenciamento de uma sentença em uma linguagem de escopo dinâmico é composto por:

variáveis declaradas localmente

+

variáveis de todos os subprogramas ativos

# Ambientes de referenciamento

- Considere o seguinte programa de exemplo.
- Admita que as únicas chamadas a funções são:
  - main chama sub2, que chama sub1.
- Vamos determinar os ambientes de referenciamento para cada um dos pontos (1, 2 e 3) destacados.

```
void sub1() {  
    int a, b;  
    ... <----- 1  
} /* Fim de sub1 */  
void sub2() {  
    int b, c;  
    ... <----- 2  
    sub1;  
} /* end of sub2 */  
void main() {  
    int c, d;  
    ... <----- 3  
    sub2();  
} /* Fim de main */
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:

- Ponto:

- 1

- Ambiente de referenciamento:

- a e b de sub1, c de sub2, d de main,  
(c de main e b de sub2 estão ocultas)

```
void sub1() {  
    int a, b;  
    ... <----- 1  
} /* Fim de sub1 */  
void sub2() {  
    int b, c;  
    ... <----- 2  
    sub1;  
} /* end of sub2 */  
void main() {  
    int c, d;  
    ... <----- 3  
    sub2();  
} /* Fim de main */
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:

- Ponto:

- 2

- Ambiente de referenciamento:

- b e c de sub2, d de main,  
(c de main está oculta)

```
void sub1() {  
    int a, b;  
    ... <----- 1  
} /* Fim de sub1 */  
void sub2() {  
    int b, c;  
    ... <----- 2  
    sub1;  
} /* end of sub2 */  
void main() {  
    int c, d;  
    ... <----- 3  
    sub2();  
} /* Fim de main */
```

# Ambientes de referenciamento

- Os ambientes de referenciamento dos pontos de programa indicados são:

- Ponto:

- 3

- Ambiente de referenciamento:

- c e d de main

```
void sub1() {  
    int a, b;  
    ... <----- 1  
} /* Fim de sub1 */  
void sub2() {  
    int b, c;  
    ... <----- 2  
    sub1;  
} /* end of sub2 */  
void main() {  
    int c, d;  
    ... <----- 3  
    sub2();  
} /* Fim de main */
```

# Constantes nomeadas

- Uma **constante nomeada** é uma variável vinculada a um valor apenas uma vez.
- Constantes nomeadas são úteis para auxiliar a **legibilidade** e a **confiabilidade** dos programas.
- A legibilidade pode ser melhorada, por exemplo, ao ser usado o nome pi em vez de constante 3.14159.
- <https://stackoverflow.com/questions/2953601/why-use-constants-in-programming>



# Constantes nomeadas

- Outro uso importante de constantes nomeadas é na **parametrização** de um programa.
- Por exemplo, considere um que processa valores de dados um número fixo de vezes, digamos 100.
- Tal programa normalmente usa a constante 100 em diversos locais para declarar as faixas de índices de vetores e para controlar os limites dos laços de repetição.
- <https://stackoverflow.com/questions/47882/what-is-a-magic-number-and-why-is-it-bad>

# Constantes nomeadas

- Considere o seguinte segmento do esqueleto de um programa Java:
- Quando esse programa for modificado para lidar com um número diferente de valores de dados, todas as ocorrências de 100 devem ser encontradas e modificadas.
- Em um grande programa, isso pode ser tedioso e propenso a erros.

```
void example() {  
    int[] intList = new int[100];  
    String[] strList = new String[100];  
    ...  
    for (index = 0; index < 100; index++) {  
        ...  
    }  
    ...  
    for (index = 0; index < 100; index++) {  
        ...  
    }  
    ...  
    average = sum / 100;  
    ...  
}
```

# Constantes nomeadas

- Um método mais fácil e confiável é usando uma constante nomeada como um parâmetro de programa.

- Agora, quando o tamanho precisar ser trocado, apenas uma linha deve ser modificada (a **variável len**), independentemente do número de vezes em que ela é usada no programa.
  - len é uma **abstração** para o número de elementos em alguns vetores e para o número de iterações em alguns laços de repetição.
  - Isso ilustra como constantes nomeadas podem auxiliar na **facilidade de modificação**.

```
void example() {  
    final int len = 100;  
    int[] intList = new int[len];  
    String[] strList = new String[len];  
    ...  
    for (index = 0; index < len; index++) {  
        ...  
    }  
    ...  
    for (index = 0; index < len; index++) {  
        ...  
    }  
    ...  
    average = sum / len;  
    ...  
}
```

# Constantes nomeadas

- O Fortran 95 permite apenas que **expressões constantes** sejam usadas como valores de suas constantes nomeadas.
  - Essas expressões constantes podem conter **constantes nomeadas previamente declaradas, valores constantes e operadores**.
- A razão para a restrição a constantes e expressões constantes em Fortran 95 é ele usar **vinculação estática de valores** às constantes nomeadas.
- Constantes nomeadas em linguagens que usam vinculação estática de valores são algumas vezes chamadas de **constantes de manifesto**.

# Constantes nomeadas

- C++ permite a **vinculação dinâmica de valores** a constantes nomeadas.
- Isso permite **expressões contendo variáveis** serem atribuídas às constantes nas declarações.
- Por exemplo, a sentença C++

```
const int result = 2 * width + 1;
```

- declara `result` como uma constante nomeada do tipo inteiro, cujo valor é informado como o da expressão `2 * width + 1`, onde o valor da variável `width` deve ser visível quando `result` é alocado e vinculado ao valor da expressão.

# Constantes nomeadas

- Java permite a **vinculação dinâmica de valores** a constantes nomeadas.
- Nela, constantes nomeadas são definidas com a palavra reservada **final**.
- O valor inicial pode ser dado na sentença de declaração ou em uma sentença de atribuição subsequente.
- O valor atribuído pode ser especificado com **qualquer expressão**.
- Links relevantes:
  - <https://www.geeksforgeeks.org/final-keyword-java/>
  - <https://stackoverflow.com/questions/15655012/how-does-the-final-keyword-in-java-work-i-can-still-modify-an-object>

# Constantes nomeadas

- C# tem dois tipos de constantes nomeadas: definidas com **const** e definidas com **readonly**.
  - As constantes nomeadas **const**, implicitamente **static**, são **estaticamente vinculadas a valores**; são vinculadas aos valores **em tempo de compilação**, ou seja, esses valores podem ser especificados apenas com literais ou outros membros **const**.
    - <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/const>
  - As constantes nomeadas **readonly**, **dinamicamente vinculadas a valores**, podem ter valores atribuídos a elas **na declaração ou com um construtor estático**.
    - <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/readonly>

# Constantes nomeadas

- Em suma:
  - Se um programa precisa de um objeto de valor constante cujo valor é o **mesmo em cada uso de um programa**, uma constante **const** é usada.
  - Se um programa precisa de um objeto de valor constante cujo valor é determinado apenas quando o objeto é criado e pode ser **diferente para execuções diversas do programa**, uma constante **readonly** é usada.
  - <https://stackoverflow.com/questions/55984/what-is-the-difference-between-const-and-readonly-in-c>



# Constantes nomeadas

- A discussão de valores vinculados a constantes nomeadas naturalmente leva ao tópico de inicialização, pois vincular um valor a uma constante nomeada é o mesmo processo, exceto que é permanente.
- Muitas vezes, é conveniente para as variáveis ter valores antes de o código do (sub)programa onde elas são declaradas começar a executar.
- A vinculação de uma variável a um valor no momento em que ela é vinculada ao armazenamento é chamada de **inicialização**.

# Constantes nomeadas

- Se a variável é estaticamente vinculada ao armazenamento, a vinculação e a inicialização ocorrem antes do tempo de execução.
  - Neste caso, o valor inicial deve ser especificado como um literal ou como uma expressão cujos operandos não literais sejam constantes nomeadas já definidas.
- Se a vinculação for dinâmica, a inicialização é também dinâmica e os valores iniciais podem ser quaisquer expressões.

# Constantes nomeadas

- Na maioria das linguagens, a inicialização é especificada na declaração que cria a variável.
- Por exemplo, em C++, poderíamos ter

```
int sum = 0;
```

```
int* ptrSum = &sum;
```

```
char name[ ] = "George Washington Carver";
```

# RESUMO

# Resumo

A sensibilidade à capitalização e o relacionamento de nomes com palavras especiais, que são palavras reservadas ou palavras-chave, são as questões de projeto para nomes.

Variáveis podem ser caracterizadas por seis atributos: nome, endereço, valor, tipo, tempo de vida e escopo.

Apelidos são duas ou mais variáveis vinculadas ao mesmo endereço de armazenamento. Eles são considerados prejudiciais à confiabilidade, mas são difíceis de serem eliminados completamente de uma linguagem.

A vinculação é a associação de atributos com entidades de programa. O conhecimento dos tempos de vinculação de atributos a entidades é essencial para entender a semântica das linguagens de programação. A vinculação pode ser estática ou dinâmica. Declarações, tanto explícitas quanto implícitas, fornecem uma forma de especificar a vinculação estática de variáveis a tipos. Em geral, a vinculação dinâmica permite uma maior flexibilidade, às custas da legibilidade, eficiência e confiabilidade.

Variáveis escalares podem ser separadas em quatro categorias, considerando seus tempos de vida: estáticas, dinâmicas da pilha, dinâmicas do monte explícitas e dinâmicas do monte implícitas.

O escopo estático é um recurso central do ALGOL 60 e de alguns de seus descendentes. Ele fornece um método simples, confiável e eficiente de permitir visibilidade a variáveis não locais em subprogramas. O escopo dinâmico fornece mais flexibilidade do que o escopo estático, mas à custa da legibilidade, confiabilidade e eficiência.

O ambiente de referenciamento de uma sentença é a coleção de todas as variáveis visíveis para aquela sentença. Constantes nomeadas são simplesmente variáveis vinculadas a valores apenas uma vez.

# 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://www.inf.unibz.it/dis/teaching/PP/ln//pp02\\_oo.pdf](http://www.inf.unibz.it/dis/teaching/PP/ln//pp02_oo.pdf)
- <http://www2.fct.unesp.br/docentes/dmec/olivete/lp/arquivos/Aula6.pdf>