Paradigmas de Linguagens de Programação

Nomes, vinculações e escopos

Introdução

- Linguagens imperativas são abstrações da arquitetura de von Neumann
- Arquitetura de von Neumann
  - Memória
  - Processador
- As abstrações para as células de memória da máquina em uma linguagem são as variáveis
  - Próxima ao hardware: variáveis inteiras
  - Mais distante: matrizes tridimensionais (mapeadas em software)
- Variáveis caracterizadas por propriedades/atributos
  - Para projetar um tipo, é necessário considerar:
    - escopo, tempo de vida, verificação de tipos, inicialização e compatibilidade de tipos

Nomes

- Um **nome** é uma cadeia de caracteres usada para identificar alguma entidade em um programa.
- Atributo fundamental das variáveis
- **Identificador** é frequentemente usado como sinônimo de nome
- Questões de projeto:
  - Os nomes são sensíveis a maiúsculas/minúsculas?
  - Palavras especiais da linguagem são palavras reservadas ou palavras-chave?

Nomes

- Nomes: Formato e Tamanho
- Primeiras linguagens → nomes de um único caractere (influência matemática)
- FORTRAN I → máximo 6 caracteres
- COBOL → até 30 caracteres
- FORTRAN 90 e C89 → até 31 caracteres (nomes externos)
- C99 → até 63 caracteres (internos)
- C, Ada, Java, C# → sem limite fixo, todos caracteres são significativos
- C++ → sem limite na linguagem, mas implementações podem impor

Nomes

- Décadas de 1970 e 1980: uso do **underscore (_)** para separar palavras
  - my_stack
- Convenção CamelCase (notação camelo) em linguagens baseadas em C
  - Primeira palavra minúscula, demais iniciam em maiúscula
  - myStack

Nomes

- Linguagens baseadas em C tem nomes **sensíveis a maiúsculas/minúsculas**
  - rosa, Rosa e ROSA são diferentes
- Desvantagem: legibilidade prejudicada (nomes parecidos significam coisas distintas)
- Problema em linguagens com muitos nomes pré-definidos em *CamelCase*
  - Java e C#: IndexOutOfBoundsException
- Algumas linguagens **não são sensíveis** (Pascal, Ada)

Nomes

- **Palavras reservadas** melhoram a legibilidade e delimitam sentenças ou instruções
- Não podem ser usadas como identificadores (ex: while, if)
- Palavras-chave: têm significado especial mas podem ser redefinidas em alguns contextos (ex: FORTRAN)
  - Real VarName
  - Real = 3.4 (*Real* is a variable)
- Excesso de palavras reservadas geram colisões
  - COBOL tem mais de 300 palavras reservadas
  - LENGTH, BOTTOM, DESTINATION e COUNT

Nomes

- PHP
  - Todos os nomes de variáveis começam com um cifrão
  - $nome, $idade
- Perl
- Caractere inicial indica o tipo da variável :
  - $ → escalar (um valor)
  - @ → array
  - % → hash (tabela associativa)
- Ruby
  - Prefixos especiais indicam o escopo da variável:
    - @variavel → variável de instância
    - @@variavel → variável de classe

Variáveis

- Uma variável é uma **abstração de uma célula de memória**
- Variáveis podem ser caracterizadas por **seis atributos**:
  - Nome
  - Endereço
  - Tipo
  - Valor
  - Tempo de vida (*lifetime*)
  - Escopo

Variáveis: Atributos

- Nome
  - Nem todas as variáveis possuem nome (variáveis temporárias do compilador)

Variáveis: Atributos

- Endereço
  - Endereço de memória com o qual a variável está associada
  - Uma mesma variável pode ter **diferentes endereços em momentos distintos** da execução (ex: variáveis locais de subprogramas)
  - Diferentes nomes podem se referir ao **mesmo endereço de memória**
    - **apelidos** (aliases)
    - Criados por: ponteiros, variáveis de referência (C++, Java), *unions* (C e C++)
    - Problema: prejudicam legibilidade, pois o programador precisa lembrar que são a mesma célula

Variáveis: Atributos

- Tipo
  - Define o conjunto de valores possíveis
  - Em Java, o tipo int representa valores de –2147483648 a 2147483647
  - Também define quais operações são válidas
    - adição, subtração, multiplicação, divisão, módulo

Variáveis: Atributos

- Valor
  - Conteúdo da célula de memória associada
  - Valor de uma variável é lido/escrito por meio de seu endereço
  - Valor de uma variável é visto como uma **célula abstrata**
    - independente do número de bytes físicos ocupados
  - l-value (*locator value*):
    - Endereço associado à variável
    - Lado esquerdo de uma atribuição x = 10;
  - r-value (*read value*):
    - Conteúdo armazenado na célula de memória
    - Lado direito de uma atribuição y = x;

Vinculação

- Associação entre um atributo e uma entidade
  - variável ↔ tipo
  - operação ↔ símbolo
- Tempo de vinculação = momento em que ocorre a associação

Vinculação

- Tempos de vinculação:
  - Projeto da linguagem: símbolo * vinculado à multiplicação
  - Implementação da linguagem: tipo float vinculado a uma representação binária
  - Compilação: variável vinculada a um tipo em C ou Java
  - Tempo de carga: variável estática em C/C++ vinculada a uma célula de memória
  - Tempo de execução: variável local não estática vinculada a uma célula de memória

Vinculação

- **Estática**
  - Acontece antes do tempo de execução
  - Permanece inalterada durante todo o programa
  - Mais previsível, mais eficiente
- **Dinâmica**
  - Acontece em tempo de execução
  - Pode mudar durante a execução do programa
  - Mais flexível, mas menos eficiente e mais difícil de verificar

Vinculação

- Exemplo de sentença de atribuição em C++

  count = count + 5;

- O tipo de *count* é vinculado em tempo de compilação
- O conjunto de valores possíveis de *count* é vinculado em tempo de projeto da linguagem
- O significado do operador **+** é vinculado em tempo de compilação, a partir dos tipos dos operandos
- A representação interna do literal 5 é vinculada em tempo de projeto do compilador
- O valor de count é vinculado em tempo de execução

Vinculação

- **Vinculação de Tipos**
  - Antes de ser usada, uma variável deve ser vinculada a um tipo de dado
- Questões importantes:
  - **Como** o tipo é especificado
  - **Quando** ocorre a vinculação
- Se for estática, pode ocorrer de duas formas:
  - **Declaração explícita**
  - **Declaração implícita**

Vinculação de tipos estática

- **Declaração Explícita**
  - Programador especifica o tipo da variável
  - Exemplo em Java:
    - *int total;*
    - *float media;*

Vinculação de tipos estática

- **Declaração Implícita**
  - Tipo é associado pela primeira aparição do nome ou por convenções da linguagem
- Exemplo em Perl:
  - Nome começando com $ → escalar
  - Nome começando com @ → array
  - *$apple* != *@apple* != *%apple*
- Exemplo em C#:
  - *var sum = 0;* // int
  - *var total = 0.0;* // float
  - *var name = "Fred";* // string

Vinculação de tipos estática

- Declarações explícitas reduzem erros e aumentam legibilidade
- Declarações implícitas:
  - Mais convenientes, mas podem levar a erros sutis
  - Exemplo: Perl permite inferência de tipo a partir do prefixo do identificador
- Em linguagens como C#, mesmo com var, o tipo é estático (não muda após inferido)

Vinculação de tipos dinâmica

- O tipo de uma variável é definido quando um valor é atribuído a ela
- O tipo pode mudar diversas vezes durante a execução do programa
- Muito comum em linguagens como Python, Ruby, JavaScript, PHP, Lisp
- Exemplo em JavaScript:

```
list = [10.2, 3.5];  // list é array
list = 47;           // list vira escalar (int)
```

Vinculação de tipos dinâmica

- Em Python:

```
x = 10      # int
x = "texto" # agora string
```

- Em Ruby:
  - Todas variáveis são referências a objetos
    - @ → variável de instância
    - @@ → variável de classe
- Em C# (2010+):
  - Palavra-chave dynamic permite vinculação dinâmica

```
dynamic any = 5;
any = "agora é string";
```

Vinculação de tipos dinâmica

- Flexibilidade:
  - Permite escrever programas mais genéricos
  - um mesmo programa pode processar diferentes tipos de dados numéricos sem declarar explicitamente cada tipo
- Facilidade de programação rápida (prototipagem)
- Usada em linguagens de script e linguagens funcionais dinâmicas

Vinculação de tipos dinâmica

- Menor eficiência:
  - Vinculação dinâmica é feita em tempo de execução, exigindo verificações adicionais
  - Programas tendem a ser mais lentos (interpretação pura pode ser 10x mais lenta que compilação estática)
- Maior chance de erros:
  - erros de digitação em nomes de variáveis podem criar novas variáveis em vez de falhar
  - Difícil detecção de incompatibilidades de tipo pelo compilador
- Maior custo de implementação:
  - Requer interpretadores ou compiladores híbridos com verificação em tempo de execução

Variáveis: Atributos

- **Armazenamento e Tempo de Vida**
- **Alocação**: obter uma célula de memória de um conjunto disponível
- **Liberação**: devolver uma célula ao conjunto de memória disponível
- **Tempo de vida**: período em que a variável está vinculada a uma posição específica da memória
- Categorias principais:
  - Estáticas
  - Dinâmicas de pilha
  - Dinâmicas de monte explícitas
  - Dinâmicas de monte implícitas

Tempo de vida

- **Variáveis Estáticas**
  - Vinculadas a células de memória **durante toda a execução do programa**
  - Exemplo: variáveis static em C e C++
  - **Vantagens**:
    - Eficiência (endereçamento direto, sem alocação/liberação em tempo de execução)
    - Histórico preservado entre chamadas de subprograma
  - **Desvantagens**:
    - Redução da flexibilidade
    - Não permitem recursividade pura
    - Dificuldade em compartilhar armazenamento entre variáveis

Tempo de vida

- **Variáveis Dinâmicas de Pilha**
  - Alocadas quando a sentença de declaração é executada
  - Liberação ocorre automaticamente no fim do bloco/função
  - Exemplo: variáveis locais em C, C++, Java, C#
  - **Vantagens**:
    - Flexibilidade (suportam recursão)
    - Conservam espaço de memória
  - **Desvantagens**:
    - Sobrecarga de alocação/liberação em tempo de execução
    - Endereçamento indireto mais lento
    - Subprogramas não podem ser sensíveis ao histórico de execução

Tempo de vida

- **Variáveis Dinâmicas de Monte Explícitas**
  - Alocadas e liberadas por instruções explícitas em tempo de execução
  - Exemplo 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
```

Tempo de vida

- **Variáveis Dinâmicas de Monte Explícitas**
  - **Vantagens**:
    - Suporte a estruturas dinâmicas (listas, árvores, grafos)
  - **Desvantagens**:
    - Gerenciamento complexo (uso de ponteiros/referências)
    - Custo de alocação/liberação
    - Possibilidade de vazamentos de memória

Tempo de vida

- **Variáveis Dinâmicas de Monte Implícitas**
  - Vinculadas a células de memória no heap via atribuição, de forma **implícita**
  - Atributos totalmente dinâmicos
  - Exemplo:
  - Exemplo JavaScript:

    var list = [74, 84, 69, 71];
  - **Vantagens**:
    - Máxima flexibilidade (código altamente genérico)
  - **Desvantagens**:
    - Ineficiência (todos atributos dinâmicos)
    - Sobrecarga de execução
    - Perda de detecção de erros pelo compilador

Tempo de vida

- A distinção entre **alocação/liberação** é fundamental para o gerenciamento de memória.
- O **tempo de vida** de variáveis locais dinâmicas de pilha está vinculado à ativação do subprograma.
- Em linguagens com garbage collector (Java, C#), a liberação de variáveis de monte explícitas é **implícita**.
- Variáveis dinâmicas implícitas trazem **máxima flexibilidade**, mas são também as mais custosas em tempo de execução.

Python: O Documentário

Escopo

- O **escopo de uma variável** é a faixa de sentenças nas quais ela é **visível** (pode ser referenciada ou receber atribuição).
- As **regras de escopo** de uma linguagem determinam como os nomes são associados às variáveis (ou expressões, em linguagens funcionais).
- Uma variável é **local** a um bloco ou unidade de programa se for declarada nele.
- Variáveis **não locais** são visíveis em um bloco/unidade, mas não foram declaradas nele.
- O entendimento claro de escopo é essencial para escrever e ler programas corretamente.

Escopo Estático

- **ALGOL 60** introduziu o método de vincular nomes a variáveis não locais, chamado de **escopo estático** (ou **escopo léxico**).
- O escopo de uma variável pode ser determinado **antes da execução**, apenas examinando o código-fonte.
- Permite que programadores e compiladores saibam a qual declaração cada variável está associada sem precisar rodar o programa.
- Usado por muitas linguagens imperativas posteriores e também em várias não imperativas.
- Duas categorias de linguagens com escopo estático:
  - **Com subprogramas aninhados** (Ada, JavaScript, Python, Lisp, Scheme, Fortran 2003+, F#).
  - **Sem subprogramas aninhados** (C, C++ e outras baseadas em C).

Escopo Estático

- **Pai estático**:
  - O bloco/subprograma em que um subprograma foi **declarado**.
  - Exemplo: se sub1 é declarado dentro de big, então o **pai estático** de *sub1* é *big*.
- **Ancestrais estáticos**:
  - O pai estático, o pai do pai, e assim por diante até o bloco mais externo.
  - Formam a **cadeia de escopos** que será usada na busca por variáveis.
- **Processo de busca** (em escopo estático):
  - Procura no bloco atual.
  - Se não encontrar, sobe para o pai estático.
  - Continua subindo pelos ancestrais até encontrar ou até chegar ao nível mais externo.
  - Se não encontrar, ocorre **erro de variável não declarada**.

```
<script>
    function big() {
        function sub1() {
            var x = 7;
            sub2();
        }
        function sub2() {
            var y = x;
        }
        var x = 3;
        sub1();
        }
    big();
</script>
```

Escopo Estático

- Em *sub2*, a busca por x começa localmente: não encontra.
- Sobe para o **pai estático** de *sub2*, que é *big*.
- Em *big*, existe x = 3.
- Resultado: y = 3.
- O x = 7 de sub1 é **ignorado**, porque sub1 **não é ancestral estático** de sub2.

```
<script>
    function big() {
        function sub1() {
            var x = 7;
            sub2();
        }
        function sub2() {
            var y = x;
        }
        var x = 3;
        sub1();
        }
    big();
</script>
```

Escopo Estático – Ocultamento (shadowing)

- Uma variável declarada em um escopo interno pode **ocultar** uma variável externa com o mesmo nome.
- No exemplo, *big* declara x = 3 e *sub1* declara outro x = 7.
- Dentro de *sub1*, toda referência a x aponta para o valor local (7), enquanto o x externo (3) fica oculto.

```
<script>
    function big() {
        function sub1() {
            var x = 7;
            sub2();
        }
        function sub2() {
            var y = x;
        }
        var x = 3;
        sub1();
        }
    big();
</script>
```

Escopo Estático – Ocultamento (shadowing)

- O x declarado dentro do while **oculta** o x externo.
- Cada referência a x dentro do bloco while usa o float x, e não o int x.
- **Java** não permite tal ocultamento

```c
#include <stdio.h>

int main() {
    int i = 0;
    int x = 10;
    while (i++ < 3) {
        float x = 3.231;
        printf("x = %f\n", x * i);
    }
}
```

Escopo Estático – Ocultamento (shadowing)

- **Java** não permite tal ocultamento

```
class Teste {
    public static void main(String[] args) {
        int i = 0;
        int x = 10;
        while (i++ < 3) {
            float x = 3.231f;
            System.out.println("x = " + (x * i));
        }
    }
}
```

java_scope.java:6: error: variable x is already defined in method main(String[])
**   float x = 3.231f;**

Blocos

- Introduzido no **ALGOL 60**
  - permitiu criar **novos escopos** no meio do código.
- Cada **bloco { }** pode ter suas próprias variáveis locais.
- Esses escopos se aninham e funcionam como **subprogramas menores**.

```
if (list[i] < list[j]) {
        int temp;
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
        }

printf("%d", temp);
```

Blocos

- Introduzido no **ALGOL 60**
  - permitiu criar **novos escopos** no meio do código.
- Cada **bloco { }** pode ter suas próprias variáveis locais.
- Esses escopos se aninham e funcionam como **subprogramas menores**.

```
if (list[i] < list[j]) {
        int temp;
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
        }

printf("%d", temp);
```

error: use of undeclared identifier 'temp'

Blocos

- Variáveis declaradas dentro de um bloco podem **ocultar** variáveis externas.
- A busca por variáveis segue a hierarquia de blocos (do menor para o maior).
- Os projetistas de Java e C# acreditavam que o reúso de nomes em blocos aninhados era muito propenso a erros para ser permitido.

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

Inválido em Java e C#

Blocos

- Nas linguagens funcionais, blocos são representados pela construção **let**.
- Estrutura geral:
  - Parte 1: associa nomes a valores (**declarações**).
  - Parte 2: expressão que usa esses nomes (**escopo local**).
- Diferente das linguagens imperativas, o let sempre retorna **um valor**, não é uma sentença.
- Em **Scheme:**

```
(LET (
   (nome_1 expressão_1)
   . . .
   (nome_n expressão_n))
   expressão
)
```

```
(LET (
   (top (+ a b))
   (bottom (- c d)))
   (/ top bottom)
)
```

Blocos

- Calcula (a + b) / (c - d).
- Os nomes top e bottom só existem **dentro do bloco let**.
- Em **Scheme:**

```
(LET (
   (top (+ a b))
   (bottom (- c d)))
   (/ top bottom)
)
```

Blocos

- Calcula (a + b) / (c - d).
- Os nomes top e bottom só existem **dentro do bloco let**.
- Em **ML:**

```
let
  val   nome_1 = expressão_1
  . . .
  val   nome_n = expressão_n
in
  expressão
end;
```

```
let
  val  top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

Blocos

- Em **F#**, a forma geral é: let nome = expressão
- Escopo:
  - Vai da definição até o fim da expressão/função.
  - Pode ser limitado por **indentação** (cada indentação cria novo escopo).

- O n2 e n3 só existem **dentro do bloco identado**.
- Fora desse bloco, apenas n1 continua acessível.
- O use de n3 no último let causa um erro

```
let n1 =
    let  n2 = 7
    let  n3 = n2 + 3
    n3;;
let n4 = n3 + n1;;
```

Blocos

- **C89**:
  - Todas as declarações de variáveis em uma função deviam aparecer **no início da função** (antes das sentenças).
  - Exceto em blocos aninhados.
- **C99, C++, Java, JavaScript, C#**:
  - Permitem declarações em **qualquer lugar do bloco**.
  - Escopo da variável vai **da declaração até o final do bloco** em que aparece.

```
void f() {
    int a = 10;
    a++;
    int b = 20; // válido em C99, inválido em C89
}
```

Blocos

- **C#**:
  - Escopo = bloco inteiro, mas variável só pode ser usada **após a declaração**.
  - Não permite redeclaração com mesmo nome em blocos aninhados.

```
{
        int x;
        { int x; }
}
```

```
{
        { int x; }
        int x;
}
```

Inválido em C# e Java

Blocos

- **JavaScript**:
  - var: escopo da função inteira (mesmo declarada no meio).
    - Uso antes da declaração: undefined.
  - Hoje prefere-se let e const: escopo de bloco.

```
console.log(x); // undefined
var x = 10;
```

- **C++, Java e C#**:
  - Permitem declaração no **for**.
  - Em versões posteriores de C++, como em Java e C#, o escopo de count vai da sentença for até o final de seu corpo (a chave de fechamento).

```
void fun() {
        . . .
        for (int count = 0; count < 10; count++){
                . . .
        }
        . . .
}
```

Escopo Global

- Variáveis globais: declaradas **fora das funções**, visíveis em múltiplas funções do arquivo.
- **C e C++**:
  - **Variáveis globais**: declaradas fora das funções, visíveis em múltiplas funções do arquivo.
  - **Declaração vs Definição**:
    - **Declaração:** informa tipo/nome (sem alocar memória).
    - **Definição:** cria a variável e aloca memória.
  - **extern**: permite declarar que uma global existe em outro arquivo (definida em outro lugar).
  - **static**: variável global visível apenas dentro do arquivo (mesma duração de global, mas visibilidade restrita).
  - Vinculação é **estática**: endereço da global é resolvido em tempo de compilação/ligação.

```
// declaração (em arquivo .h)
extern int sum;
// definição (em um .c)
int sum = 0;
// global restrita ao arquivo
static int y = 5;
```

Escopo Global

- **C99**: globais recebem valor inicial **zero por padrão** se não inicializadas.
  - Exemplo: *int g;* automaticamente inicializada com 0.
  - Diferente das locais, que ficam com valor **indeterminado**.
- **C++**: permite acessar a global mesmo se houver local com mesmo nome usando ::.

```
int x = 10; // global
void f() {
        int x = 20; // oculta global
        printf("%d", ::x); // acessa global (10)
}
```

Escopo Global

- **PHP**:
  - Variáveis globais podem ser usadas em funções com global ou $GLOBALS.
- **JavaScript (var)**:
  - Comportamento parecido ao PHP.
  - Globais podem ser acessadas, mas não se pode acessar a global se houver uma local com o mesmo nome.

local day is Tuesday
global day is Monday
global month is January

```php
<?php
$day = "Monday";
$month = "January";
    function calendar() {
        $day = "Tuesday";
        global $month;
        print "local day is $day";
        $gday = $GLOBALS['day'];
        print "global day is $gday <br \>";
        print "global month is $month";
    }

calendar()
?>
```

Escopo Global

- **Python**:
  - Variáveis globais podem ser acessadas em funções.

Python 2.x

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

tester()
```

The global day is: Monday

Python 3.x

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

tester()
```

The global day is: Monday

Escopo Global

- **Python**:
  - Para modificar global em Python
    - usar **global**.

```
day = "Monday"

def tester():
        print("The global day is:", day)
        day = "Tuesday"
        print("The global day is:", day)

tester()
```

UnboundLocalError: local variable 'day' referenced before assignment

Escopo Global

- **Python**:
  - Para modificar global em Python
    - usar **global**.

```
day = "Monday"

def tester():
    global day
    print("The global day is:", day)
    day = "Tuesday"
    print("The global day is:", day)

tester()
```

The global day is: Monday
The global day is: Tuesday

Escopo Global

- **Python**:
  - Em Python, para modificar variáveis da função externa, usa-se **nonlocal**.
  - Sem **nonlocal**, a função interna criaria uma nova variável local.

```
def outer():
    x = "outer"
    def inner():
        nonlocal x
        x += " modified in inner"
    inner()
    print(x)

outer()
```

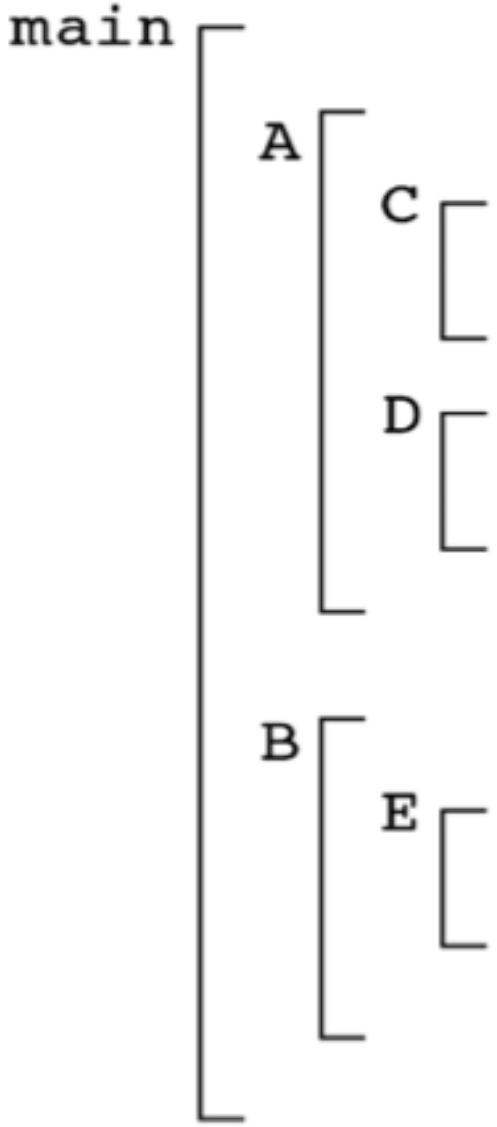
outer modified in inner

Escopo Estático

- **Vantagem**: simples, previsível e amplamente utilizado.
- **Problemas**:
  - Pode conceder acesso mais amplo do que o necessário.
  - Dificulta a evolução de programas, pois mudanças de estrutura quebram restrições.
  - Incentiva o uso excessivo de variáveis globais.
  - Projetos podem se tornar artificiais e difíceis de manter.
- **Alternativa moderna**:
  - Uso de **encapsulamento** (classes, pacotes, módulos) para controlar o acesso a variáveis e subprogramas.

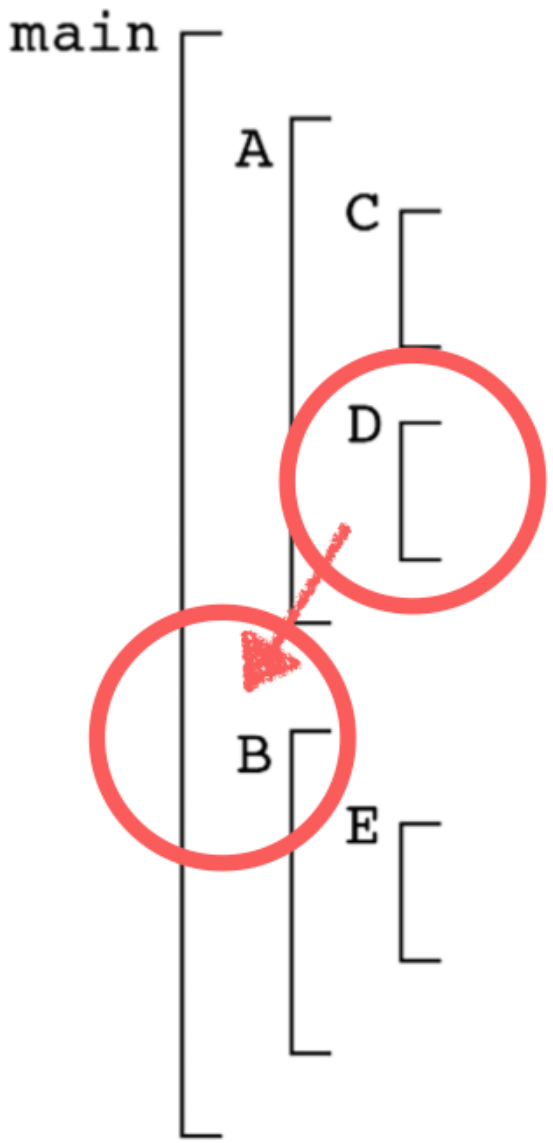
Escopo Estático

- Assumindo que MAIN chama A e B
- A chama C e D
- B chama A eE



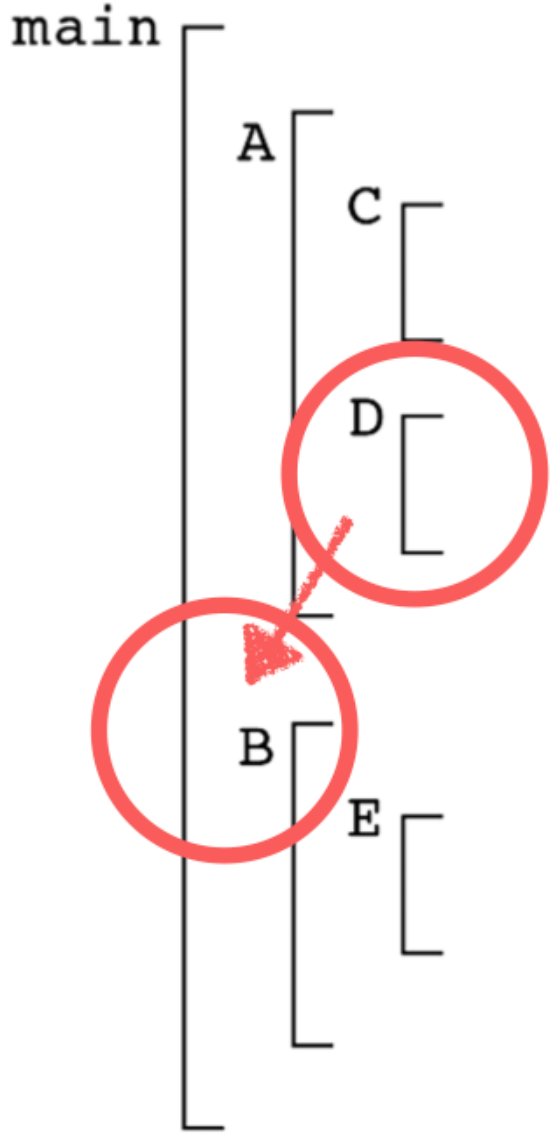
Escopo Estático

- Suponha que a especificação mudou e D agora precisa acessar dados em B.



Escopo Estático

- Possíveis soluções:
  - Colocar D dentro de B
  - Mas D deixa de acessar variáveis de A.
  - Mover os dados de B que D precisa para MAIN
  - Mas assim todas as funções passam a acessar esses dados.
- Conclusão: O escopo estático muitas vezes incentiva o uso excessivo de variáveis globais.



Escopo Dinâmico

- Vantagem principal: conveniência.
- O subprograma chamado executa no **contexto do chamador**.
- Permite reutilizar subprogramas sem passar muitas variáveis como parâmetros.
- Exemplo: parâmetros implícitos podem ser acessados diretamente pelo subprograma.

Escopo Dinâmico

- **Dificulta a leitura**: é muito difício prever a origem de uma variável apenas lendo o código.
- **Menor confiabilidade**: dependência da ordem de chamadas.
- **Tipos não podem ser checados estaticamente:** erros só aparecem em tempo de execução.
- Em geral, **linguagens modernas abandonaram o escopo dinâmico**, substituindo-o por:
  - Escopo estático (padrão atual).
  - Encapsulamento e passagem explícita de parâmetros.

Escopo Dinâmico

- Determinado em **tempo de execução**, não em tempo de compilação.
- Baseado na **sequência de chamadas de subprogramas**, não na posição do código.
- Uma referência a variável não local procura a declaração:
- Primeiro nas variáveis locais da função atual.
- Depois nos subprogramas **que chamaram** a função, seguindo a pilha de chamadas.
- Usado em linguagens antigas (APL, SNOBOL, primeiras versões de Lisp) e também em Perl (apesar do padrão ser estático).

Nota Histórica - Perl

- Criada em 1987 por Larry Wall, Perl tornou-se muito popular nos anos 1990.
- Foi chamada de **"duct tape that holds the Internet together"**, pois dominava o desenvolvimento de scripts CGI (Common Gateway Interface) para páginas web dinâmicas.
- Sua flexibilidade e suporte a diferentes paradigmas (procedural, OO, funcional) ajudaram a difundir ideias que influenciaram linguagens posteriores.
- Antes do Python ganhar força, Perl era a principal escolha para automação de sistemas, manipulação de texto e scripts web.
- Até hoje possui frameworks modernos como **Catalyst**

Escopo Dinâmico

- Se sub2 for chamado por sub1:
  - A referência a x em sub2 encontra o x = 7 de sub1.
- Se sub2 for chamado diretamente por big:
  - A referência a x em sub2 encontra o x = 3 de big.
- Conclusão: no escopo dinâmico, o valor acessado depende da ordem de chamadas e não da posição no código.

```
function big() {
  function sub1() {
    var x = 7;
  }
  function sub2() {
    var y = x;
    var z = 3;
  }
  var x = 3;
}
```

Escopo e Tempo de Vida

- **Escopo**: região do código onde um nome é visível.
  - variável local a um bloco/função só é referenciável dentro daquele bloco/função.
- **Tempo de vida**: intervalo de tempo de execução em que a variável existe na memória.
- Eles podem coincidir, mas não são a mesma coisa:
  - Em C/C++, uma variável local static tem **escopo local** (visível só na função), mas tempo de **vida global** (existe por toda a execução do programa).
  - Em Java/C/C++, uma variável local (sem static) tem **escopo local** e **tempo de vida** apenas durante a **ativação** da função (frame na pilha).

Escopo e Tempo de Vida

- Escopo de sum: apenas o corpo de compute.
- sum não é visível dentro de printheader.
- Tempo de vida de sum: começa quando compute é ativada e permanece válido enquanto compute estiver ativa
- sum continua existindo na pilha enquanto printheader roda, embora não seja visível dentro de printheader.
- sum existe (tempo de vida) durante printheader, mas não pode ser acessado (escopo) em printheader.

```
void printheader() {
  . . .
 }  /* fim de printheader */
void compute() {
  int sum;
  . . .
  printheader();
 }  /* fim de compute */
```

Ambientes de Referenciamento

- **Definição**: o ambiente de referenciamento de uma sentença é o **conjunto de todas as variáveis visíveis** naquele ponto do programa.
- **Escopo estático**: inclui as variáveis declaradas no **escopo local** + todas as dos **escopos ancestrais visíveis**.
- Em linguagens de escopo estático, esse ambiente é determinado **na compilação**, garantindo que referências a variáveis não locais sejam resolvidas corretamente.
- **Escopo dinâmico**: o ambiente de referenciamento de uma sentença é formado pelas variáveis locais + todas as variáveis visíveis em **subprogramas ativos** (isto é, ainda não finalizados).

Ambientes de Referenciamento

- Em linguagens como **Python**, cada definição de função cria um novo escopo, e o ambiente de referenciamento é formado pelas variáveis locais e visíveis em funções externas (exceto as que foram ocultadas por declarações mais internas).

```
g = 3; # Uma global

def sub1():
    a = 5; # Cria uma local
    b = 7; # Cria outra local
    . . . <------------------------------ 1
  def  sub2():
  global g;  # A global g agora pode ser atribuída aqui
    c = 9;  # Cria uma nova local
    . . . <------------------------------ 2
    def  sub3():
      nonlocal  c:  # Torna c não local visível aqui
      g = 11;  # Cria uma nova local
      . . . <------------------------------ 3
```

Ambientes de Referenciamento

```
g = 3; # Uma global

def sub1():
    a = 5; # Cria uma local
    b = 7; # Cria outra local
    . . . <------------------------------ 1
  def  sub2():
  global g;  # A global g agora pode ser atribuída aqui
    c = 9;  # Cria uma nova local
    . . . <------------------------------ 2
    def  sub3():
      nonlocal  c:  # Torna c não local visível aqui
      g = 11;  # Cria uma nova local
      . . . <------------------------------ 3
```

Ponto	*Ambiente de referenciamento*
1	local `a` e `b` (de `sub1`), global `g` para referência, mas não para atribuição
2	local `c` (de `sub2`), global `g` tanto para referência como para atribuição
3	não local `c` (de `sub2`), local `g` (de `sub3`)

Constantes Nomeadas

- **Constante nomeada**: variável vinculada a um valor uma única vez.
- Melhora a **legibilidade**
  - usar *pi* em vez de 3.14159265).
- Facilita a **confiabilidade** e **manutenção** do programa.
- A associação de valores às constantes nomeadas pode ser **estática** ou **dinâmica**.
- Exemplos por linguagem:
  - **FORTRAN 95**: expressões de valor constante.
  - **Ada, C++, Java**: permitem expressões de qualquer tipo.
  - **C#**: possui dois tipos de constantes nomeadas: const (estática) e readonly (dinâmica).

Constantes Nomeadas

- Parametrização de programas
  - evita repetir valores fixos em múltiplos pontos do código.

```
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;
```

```
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

- **C++**
  - Permite vinculação dinâmica de valores a constantes nomeadas.
  - Expressões contendo variáveis podem ser atribuídas a constantes.

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

Paradigmas de Linguagens de Programação

Nomes, vinculações e escopos