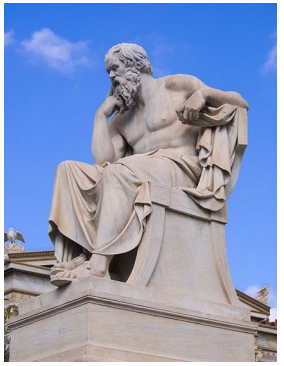
Linguagens de Programação

5. Nomes, vinculações e escopos

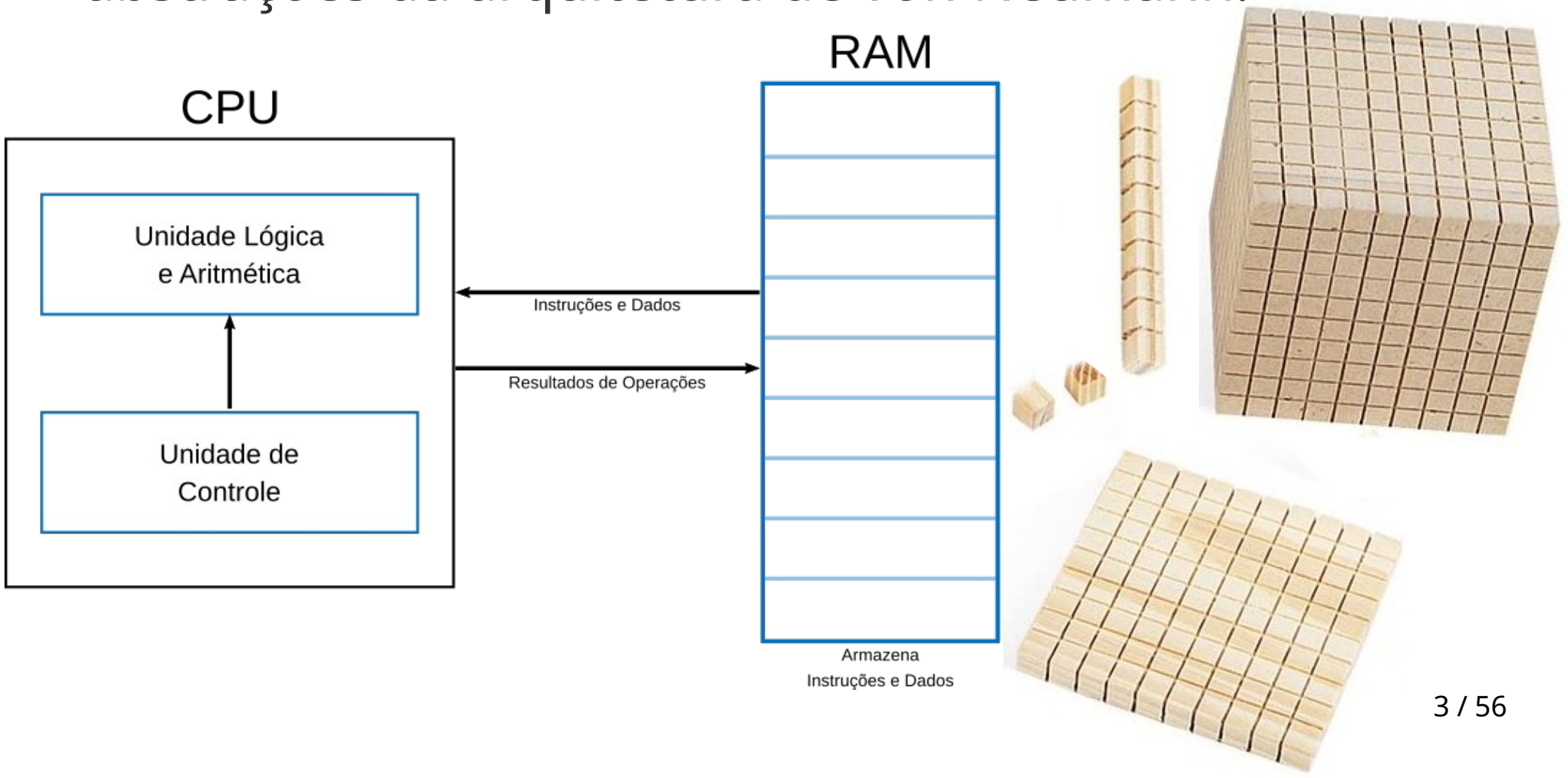
O que aprenderemos?



- Problemas semânticos fundamentais das variáveis (pode chamar de problemas filosóficos se preferir):
  - O que é um nome/identificador?
  - De que é feita uma variável? Quais seus atributos?
  - O que é um apelido?
  - O que é vinculação? E tempo de vinculação?
  - Todas as variáveis são “iguais”, têm a mesma alma?
  - Onde uma variável existe? Qual seu tempo de vida?
  - O que é inicializar uma variável?

1. Por quê estudar variáveis?

- **Linguagens imperativas**: são, em graus variados, abstrações da arquitetura de von Neumann.



1. Por quê estudar variáveis?

- **Linguagens funcionais puras**: “variáveis” não variam!

  ”A variable in Haskell gets defined only once and cannot change. The variables in Haskell seem almost *invariable*, but they work like variables in mathematics. In a math classroom, you never see a variable change its value within a single problem. In precise terms, Haskell variables are *immutable*.”

  https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

2. Variáveis

- Uma variável **é uma abstração para uma célula ou mais células da memória do computador**. NÃO É só um nome para um endereço de memória (aceitável).
- É caracterizada por **6 atributos**:
  - Nome
  - Endereço
  - Tipo
  - Valor
  - Tempo de vida
  - Escopo

2.1. Nomes das variáveis

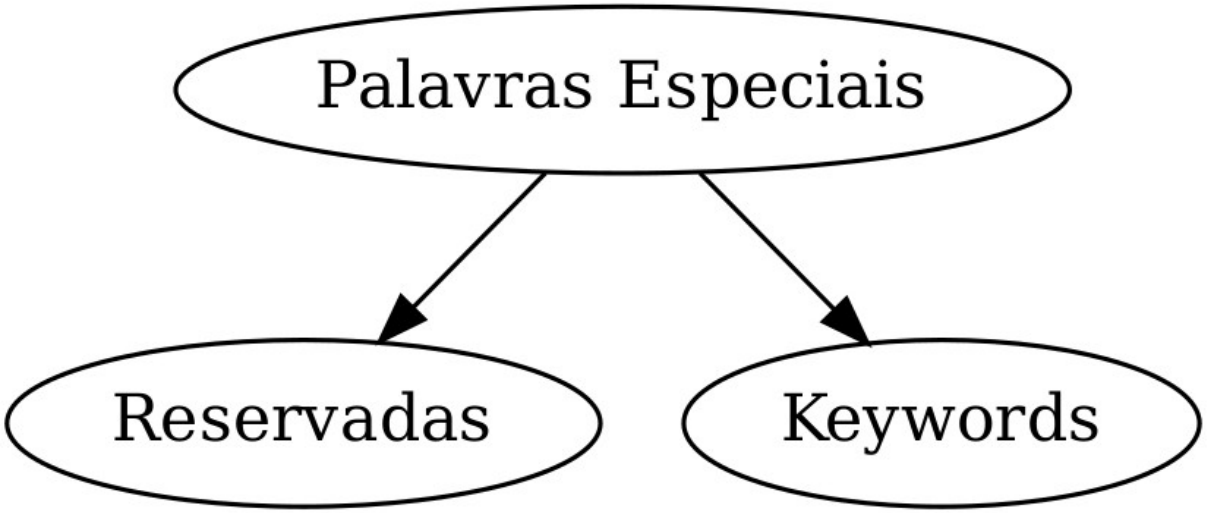
- Quase todas as variáveis tem nome. O que é um nome?
  - Um nome é uma **string (cadeia) de caracteres utilizada para identificar alguma entidade em um programa** (variável, subprograma, parâmetros etc.).
- Questões importantes:
  - Qual o tamanho máximo permitido?
  - Qual a forma permitida?
  - Qual o estilo adequado?
  - Case sensitive?
  - Tem palavras especiais?

2.1. Nomes das variáveis

- Qual o **tamanho máximo** permitido?
  - Depende da linguagem
- Qual a **forma permitida**?
  - Em geral: inicia por letra ou underscore, e é seguido por uma string de letras, dígitos ou underscores
- Qual o **estilo adequado**?
  - Underscores
  - Camel Notation
  - Siga um único!

2.1. Nomes das variáveis

- O nome é **case sensitive**?
  - Descubra e siga um único padrão!
- Que **palavras especiais** que não devem ser usadas?
  - Dão nomes às ações e separam partes sintáticas das sentenças e programas



2.2. Endereços das variáveis

- O endereço de uma variável **é o endereço da memória física ao qual a variável está associada**.
  - Associação não é simples
  - “L-value”
- **Aliases** (apelidos): múltiplas variáveis (nomes diferentes) que têm o mesmo endereço.
  - Problema para legibilidade
  - Criados de diferentes maneiras

2.2. Endereços das variáveis

- Exercício prático: descobrindo endereços em C!

2.3. Tipos das variáveis

- O tipo de uma variável indica a **faixa de valores** que podem ser armazenados, e o **conjunto de operações definidas** para os valores permitidos.

TIPO = VALORES + OPERAÇÕES

2.4. Valor das variáveis

- O valor de uma variável é o **conteúdo da(s) célula(s) de memória** associada(s) a ela.
  - Consideramos **células abstratas** de memória
    - Células físicas têm 1 byte (8 bits), pequeno demais para a maioria das variáveis em um programa
    - Células abstratas tem o tamanho exigido pela variável à qual está associada
  - “r-value”

EXTRA! Arquitetura de Memória!

- ANTES de podermos continuar o estudo de variáveis, temos que entender como é a arquitetura e a alocação da memória nos computadores. Como esse assunto não é discutido no livro texto da disciplina, faremos uma estudo extra antes de continuar.

2.5. Tempo de Vida: vinculação

- Vinculação é uma **associação entre um atributo e uma entidade**, por exemplo:
  - Tipo e faixa de valores
  - Tipo e operações permitidas
  - Variável e tipo
  - Variável e valor
  - Operação e seu símbolo
  - Etc.

2.5. Tempo de Vida: vinculação

- O momento no qual a vinculação ocorre é chamado de **tempo de vinculação**. Existem diferentes tempos:
  - Tempo de **projeto** (símbolos e suas operações)
  - Tempo de **implementação** (tipos com seus valores)
  - Tempo de **compilação** (variável e seu tipo de dado)
  - Tempo de **carga** (variáveis diversas)
  - Tempo de **ligação** (subprograma de biblioteca)
  - Tempo de **execução** (variáveis diversas)

2.5. Tempo de Vida: vinculação

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int contagem = 0;
   contagem = contagem + 1;
   printf("%d\n", contagem);
   return EXIT_SUCCESS;
}
```

- Tempo de vinculação:
  - Valor de **`int`**
  - Tipo de **`contagem`**
  - Operação de **`+`**
  - Literal **`0`** ou **`1`**
  - Valor de **`contagem`**
  - Chamada de **`stdlib`**

2.5. Tempo de Vida: vinculação

- “Tipos” de vinculação:
  - **Estática**: ocorre antes do tempo de execução e não se altera ao longo da execução do programa
  - **Dinâmica**: ocorre durante o tempo de execução e pode ser alterada ao longo da execução do programa

2.5. Tempo de Vida: vinculação

- **Vinculação estática de tipos** às variáveis:
  - Quando ocorre?
  - Como o tipo é especificado?
    - Declaração explícita do tipo
      - Indica explicitamente o tipo da variável em uma setença de declaração
    - Declaração implícita do tipo
      - Inferência de tipos (tipo do valor atribuído)
      - Convenção pelo nome
  - Tipo não muda!

2.5. Tempo de Vida: vinculação

- **Vinculação estática de tipos** às variáveis:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int contagem = 0;
   contagem = contagem + 1;
   printf("%d\n", contagem);
   return EXIT_SUCCESS;
}
```

```perl
#!/usr/bin/env perl

use strict;

my $idade = 48;

print $idade;
```

```c#
using System;

var idade = 49;
var total = 10.0;
var nome = "Abrantes";

Console.WriteLine(idade);
```

2.5. Tempo de Vida: vinculação

- **Vinculação dinâmica de tipos** às variáveis:
  - Quando ocorre?
  - Como o tipo é especificado?
    - Não tem declaração, não pode ser determinado pelo nome!
    - O tipo é vinculado na atribuição do valor!
  - Tipo é temporário, pode mudar!
  - Desvantagens:
    - Menos confiável, compilador não detecta erros
    - Cuso em tempo de execução (10x maior)

2.5. Tempo de Vida: vinculação

- **Vinculação dinâmica de tipos** às variáveis:

```python
#!/usr/bin/env python3

idade = 49

print(idade)
```

```php
<?php
$txt = "Hello world!";
$x = 5;
$y = 10.5;
?>
```

```ruby
#!/usr/bin/ruby

$idade = 49
puts "A idade é #$idade"
```

2.5. Tempo de Vida

- **Alocação**: é o processo de vincular uma variável à uma ou mais células de memória disponíveis.
- **Liberação**: é o processo de desvincular a variável das células de memória, deixando essas células disponíveis para novas vinculações.
- **Tempo de vida**: tempo durante o qual a variável está vinculada com uma posição específica de memória.
  - Estáticas (**static**)
  - Dinâmicas da pilha (**stack-dynamic**)
  - Dinâmicas do monte (**heap-dynamic**) (**explícitas** ou **implícitas**)

2.5. Tempo de Vida

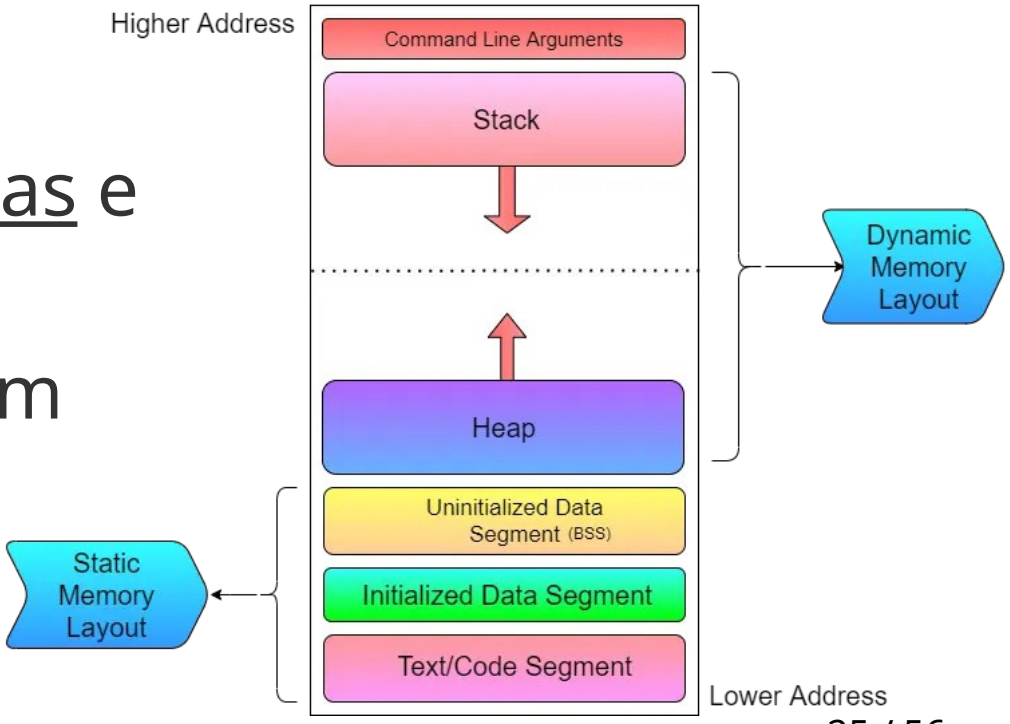
- **Variáveis Estáticas (static)**: são vinculadas às células de memória antes do início da execução do programa e permanecem vinculadas a essas mesmas células **até que a execução do programa termine**.
  - Vantagens:
    - Eficientes, endereçamento direto
    - Sem sobrecarga de vinculação em runtime
  - Desvantagnes:
    - Menor flexibilidade
    - Armazenamento não compartilhado
    - Não permite o uso de programas recursivos

2.5. Tempo de Vida

- **Variáveis Estáticas (static)**: exemplos?????

2.5. Tempo de Vida

- **Variáveis Estáticas (static)**:
  - Variáveis globais, estáticas, constantes e extern que foram inicializadas com algum valor diferente de 0 (DATA)
  - Variáveis globais, estáticas e extern que não foram inicializadas ou que foram inicializadas com 0 (BSS)



2.5. Tempo de Vida

- **Variáveis Estáticas (static):**

```c
#include <stdio.h>
//#include "externa.h"

//int global_inicializada = 10;
//const float PI = 3.14;
//extern int externa;

int main(void)
{
  //static int local_estatica = 10;
  //extern int externa2;
  //printf("global_inicializada: %p\n", &global_inicializada);
  //printf("constante PI: %p\n", &PI);
  //printf("estática local: %p\n", &local_estatica);
  //printf("extern: %p\n", &externa);
  //printf("extern2: %p\n", &externa2);
  //while(1);
  return 0;
}
```

```
 text    data     bss     dec     hex filename
 1418     544       8    1970     7b2 ./variaveis_estaticas
```

2.5. Tempo de Vida

- **Variáveis Estáticas (static):**

```c
#include <stdio.h>
#include "externa.h"

int global_inicializada = 10;
const float PI = 3.14;
extern int externa;

int main(void)
{
  static int local_estatica = 10;
  extern int externa2;
  printf("global_inicializada: %p\n", &global_inicializada);
  printf("constante PI: %p\n", &PI);
  printf("estática local: %p\n", &local_estatica);
  printf("extern: %p\n", &externa);
  printf("extern2: %p\n", &externa2);
  //while(1);
  return 0;
}
```

```
  text    data     bss     dec     hex filename
  1776     616       8    2400     960 ./variaveis_estaticas
```

2.5. Tempo de Vida

- **Variáveis Estáticas (static):**

```
global_inicializada: 0x559be25ff018
constante PI:        0x559be25fd004
estática local:      0x559be25ff01c
extern:              0x559be25ff010
extern2:             0x559be25ff014
```

```
559be25ff000-559be2600000 rw-p 00003000 08:01 9437975
559be2f46000-559be2f67000 rw-p 00000000 00:00 0                          [heap]
7ffe1a0df000-7ffe1a101000 rw-p 00000000 00:00 0                          [stack]
```

2.5. Tempo de Vida

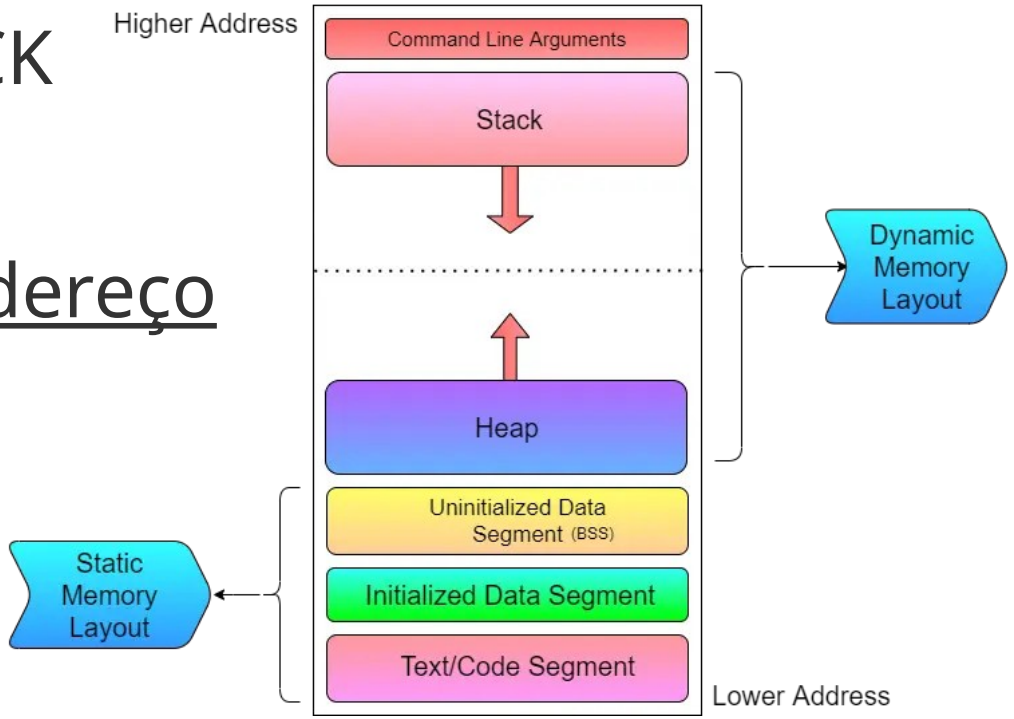
- **Variáveis Dinâmicas da Pilha (stack-dynamic)**: são vinculadas às células de memória em tempo de execução, quando a **declaração é elaborada**.
  - Declaração elaborada: quando a execução alcança a linha de código da declaração.
  - Se escalar, todos os outros atributos exceto o endereço são vinculados de modo estático. Se struct é diferente (veremos futuramente).
  - Tempo de vida: até a função/procedimento terminar
  - Vantagens: recursão, compartilhar espaço
  - Desvantagem: sobercarga em runtime

2.5. Tempo de Vida

- **Variáveis Dinâmicas da Pilha (stack-dynamic)**: exemplos?????

2.5. Tempo de Vida

- **Dinâmicas da Pilha (stack-dynamic)**
  - Tudo que é necessário em uma chamada de função (incluindo funções recursivas): STACK FRAME
    - Função chamada e endereço de retorno
    - Parâmetros com seus argumentos
    - Variáveis locais



2.5. Tempo de Vida

- **Variáveis Dinâmicas da Pilha (stack-dynamic)**:

```c
#include <stdio.h>

int soma(int x, int y)
{
  int resultado = 0;
  resultado = x + y;
  printf("resultado: %p\n", &resultado);
  return resultado;
}

int main(void)
{
  int calculo = 0;
  calculo = soma(2, 3);
  printf("calculo:   %p\n", &calculo);
  //while(1);
  return 0;
}
```

```
7ffd9e8ca000-7ffd9e8ec000 rw-p 00000000 00:00 0                          [stack]
resultado: 0x7ffd9e8e9f74
calculo:   0x7ffd9e8e9f94
```

2.5. Tempo de Vida

- **Variáveis Dinâmicas do Monte Explícitas (explicit heap-dynamic)**: células de memória não nomeadas, alocadas e liberadas manualmente pelo programador, em tempo de execução, com de instruções explícitas.
  - Só podem ser referenciadas por ponteiros ou variáveis de referência (o ponteiro é uma variável stack-dynamic!).
  - Tempo de vida: determinado pelo programador
  - Vantagem: estruturas dinâmicas (listas, grafos, árv.)
  - Desvantagem: sobrecarga, manual
  - Endereço dinâmico, resto é estático

2.5. Tempo de Vida

- **Variáveis Dinâmicas da Pilha (stack-dynamic)**:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int *ponteiro = malloc(sizeof(int));
  *ponteiro = 10;
  printf("endereço do ponteiro: %p\n", &ponteiro);
  printf("conteúdo do ponteiro: %p\n", ponteiro);
  printf("conteúdo no heap:     %d\n", *ponteiro);
  free(ponteiro);
  //while(1);
  return 0;
}
```

```
endereço do ponteiro: 0x7ffe306b0080
conteúdo do ponteiro: 0x55ff9233d2a0
conteúdo no heap:     10
```

```
7ffe30691000-7ffe306b3000 rw-p 00000000 00:00 0                          [stack]
55ff9233d000-55ff9235e000 rw-p 00000000 00:00 0                          [heap]
```

2.5. Tempo de Vida

- **Variáveis Dinâmicas do Monte Implícitas (implicit heap-dynamic)**: são vinculadas ao heap apenas quando são atribuídos valores para as variáveis. Dura até redefinir a variável.
  - Vantagem: alto grau de flexibilidade, escrita de código altamente genérico
  - Desvantagem: sobrecarga, compilador não consegue detectar alguns erros, todos os atributos são dinâmicos
  - Exemplo: Perl, PHP, JavaScript:

```
numeros = [7, 8, 8, 9, 2]
```

2.6. Escopo de variáveis

- É a **faixa de sentenças nas quais ela é visível**.
  - Visível: se puder ser referenciada ou atribuída nessa sentença
- Regras de escopo definem:
  - Como determinado um nome é associado à variável
  - Como referências à variáveis declaradas fora do bloco ou subprograma são associadas às suas declarações e atributos

2.6. Escopo de variáveis

- **Variáveis locais**:
  - São variáveis declaradas em uma unidade ou bloco do programa, sendo visíveis somente nessa unidade
- **Variáveis não locais**:
  - São variáveis visíveis dentro da unidade ou do bloco do programa, mas que não foram declaradas nessa unidade ou bloco
    - **Variáveis globais**: são uma categoria especial de variáveis não locais.

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - O escopo de uma variável pode ser determinada estaticamente (antes da execução)
    - Programador consegue ler o código e determinar o tipo ou valor de cada variável no programa
    - O compilador também!
  - Duas categorias de escopo estático:
    - Subprogramas aninhados
      - ADA, JavaScript, Lisp, Scheme, F#, Python
    - Subprograms não aninhados
      - C e derivados

2.6. Escopo de variáveis

- **Estático (ou léxico):**

```
def principal():
    def sub1():
        x = 7
        print("X em sub1: ", x)
        sub2()

    def sub2():
        y = x
        print("X em sub2: ", y)

    x = 3
    print("X em principal: ", x)

    sub1()


principal()
```

2.6. Escopo de variáveis

- **Estático (ou léxico):**

```
def principal():
    def sub1():
        x = 7
        print("X em sub1: ", x)
        sub2()

    def sub2():
        y = x
        print("X em sub2: ", y)

        def sub3():
            z = x
            print("X em sub3: ", z)

        sub3()

    x = 3
    print("X em principal: ", x)

    sub1()

principal()
```

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - A variável está declarada no bloco ou na unidade?
    - Se sim, é essa que será referenciada;
    - Se não, deve-se buscar nos **ancestrais estáticos** do bloco ou unidade
      - ATENÇÃO: **não confundir ancentral estático com quem chamou o bloco ou unidade**! Ancentral estático é uma questão de "relacionamento espacial".
  - Variáveis locais com o mesmo nome ocultam as variáveis não locais com o mesmo nome

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - Muitas linguagens permitem que escopos estáticos sejam definidos no meio do código executável em algum “bloco” separado (**linguagem estruturada por blocos**).
  - Esses “blocos” podem ter suas próprias variáveis locais com escopo minimizado, stack-dinâmicas.
  - Blocos costumam ser delimitados por **chaves** { } (C e derivados) ou por **indentação** (Python).

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - Blocos

```c
#include <stdio.h>

int main(void)
{
    for(int i = 1; i <= 3; i++)
    {
        printf("%d\n", i);
    }

    printf("%d\n", i); // ERRO!

    return 0;
}
```

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - Blocos

```c
#include <stdio.h>

int main(void)
{
    int i = 10;

    for(int i = 1; i <= 3; i++)
    {
        printf("%d\n", i);
    }

    printf("%d\n", i);

    return 0;
}
```

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - Blocos

```c
#include <stdio.h>

int main(void)
{
    int i = 10;

    for(i = 1; i <= 3; i++)
    {
        printf("%d\n", i);
    }

    printf("%d\n", i);

    return 0;
}
```

2.6. Escopo de variáveis

- **Estático (ou léxico)**:
  - Ordem

```c
#include <stdio.h>

int main(void)
{
    printf("Variável x é: %d\n", x); // ERRO
    int x = 10;
    return 0;
}
```

```python
hoje = "domingo"


def teste():
    print("Hoje é ", hoje)
    return


def teste2():
    print("Hoje é ", hoje)
    hoje = "segunda"
    print("Hoje é ", hoje)


def teste3():
    global hoje
    print("Hoje é ", hoje)
    hoje = "segunda"
    print("Hoje é ", hoje)
```

2.6. Escopo de variáveis

- **Dinâmico**:
  - O escopo de uma variável não pode ser determinado ANTES da execução, pois é **baseado na seqüência de chamadas dos subprogramas**, não em seu relacionamento “espacial” uns com os outros.
  - Portanto só pode ser determinado em tempo de exeução, através dos **ancentrais dinâmicos**!
    - Ancestrais dinâmicos referem-se a quem chamou o subprograma, não às relações espaciais!

2.6. Escopo de variáveis

- **Dinâmico**:

```
function principal() {

    function sub1() {
        var x = 7;
        sub2();
    }

    function sub2() {
        var y = x;
    }

    var x = 3;

}
```

2.6. Escopo de variáveis

- **Dinâmico**:

```
function principal() {

    function sub1() {
        var x = 7;
        sub2();
    }

    function sub2() {
        var y = x;
    }

    var x = 3;
    sub1();
    sub2();
}
```

2.6. Escopo de variáveis

- **Dinâmico**: problemas!
  - Atributos das variáveis não locais visíveis a uma sentença em um programa não podem ser determinados estaticamente
  - Uma referência ao nome de tal variável nem sempre é para a mesma variável, depende da seqüência de execução do subprograma
    - Uma variável não local pode se referir a diferentes variáveis não locais durante diferentes execuções do subprograma

2.6. Escopo de variáveis

- **Dinâmico**: problemas!
  - As variáveis locais são todas visíveis para qualquer outro subprograma que esteja sendo executado, independentemente de sua proximidade textual ou de como a execução chegou ao sugprograma que está executando atualmente
  - Os subprogramas são sempre executados no ambiente de todos os subprogramas chamados anteriormente e que ainda não completaram suas execuções. Diminui a confiabilidade!
  - Impossibilidade de verificar os tipos das referências às variáveis não locais.

2.6. Escopo de variáveis

- **Dinâmico**: problemas!
  - Programas são difíceis de serem lidos porque a seqüência de chamadas de subprogramas deve ser conhecida para se determinar o significado das referências à variáveis não locais.
  - Acesso à variáveis não locais demora muito mais.
- Tem alguma vantagem?
  - Argumentos não precisam ser passados porque são implicitamente visíveis.
  - E só...

2.6. Escopo x Tempo de Vida

- São conceitos diferentes!
  - Escopo: é um conceito espacial (se estático) ou de chamadas de funções (se dinâmico)
  - Tempo de vida: é um conceito temporal
- Podem estar relacionados ou não!

```c
#include <stdio.h>

int soma(int a, int b);

int main(void)
{
    int retorno = soma(2, 3);
    printf("A soma é: %d\n", retorno);
    return 0;
}

int soma(int a, int b)
{
    static int resultado = 0;
    resultado = a + b;
    return resultado;
}
```

2.7. Detalhes finais

- **Ambiente de referenciamento**:
  - É a coleção de todas as variáveis visíveis
    - Escopo estático: todas as variáveis locais mais as variáveis não locais dos ancentrais estáticos
    - Escopo dinâmico: todas as variáveis locais mais as variáveis não locais dos ancentrais dinâmicos
- **Constantes nomeadas**:
  - É uma “variável” que é vinculada a um valor apenas uma única vez, sem poder ser alterada posteriormente.
    - Legibilidade, parametrização

2.7. Detalhes finais

- **Inicialização**:
  - A vinculação de uma variável a um valor no momento em que ela é vinculada ao armazenamento é chamada de inicialização.
    - Se a variável é vinculada estaticamente, a inicialização é estática eo valor inicial deve ser especificado como um literal ou como uma expressão cujos operandos não literais sejam constantes nomeadas;
    - Se a variável é vinculada dinamicamente, a inicialização é dinâmica eo valor inicial pode ser qualquer expressão.

Agora acabou!

- **Dúvidas?**