Técnicas de Programação

Luiz Fernando Carvalho

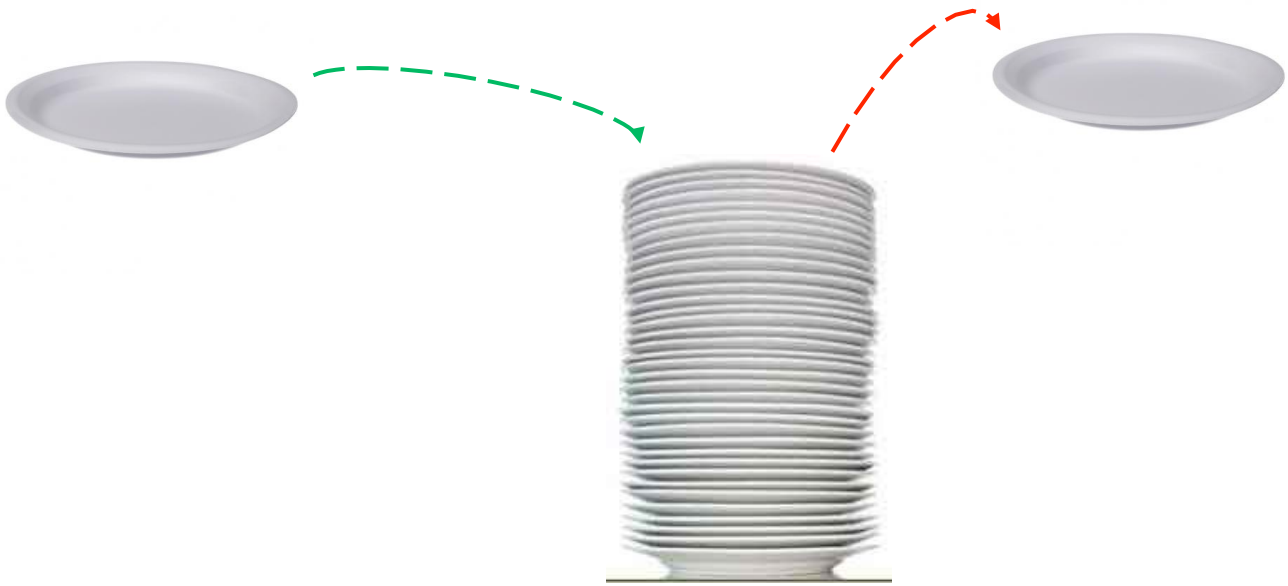
luizfcarvalhoo@gmail.com

Introdução

- Até agora declaramos e tipos primitivos da linguagem C
  - `int`, `float`, `char`, `double`, `unsigned`, `short`, etc.
- Usamos também dados mais complexos:
  - *Arrays*, matriz e `structs`.
- Toda vez que essas variáveis foram usadas, eram colocadas em uma pilha (*stack*);
- O que é uma pilha?

Pilha

- Pilha é uma estrutura de dados em que:
  - Os novos elementos só podem ser colocados no topo da pilha;
  - Os elementos só podem ser retirados do topo da pilha;
  - Chamada de LIFO (*Last IN*, *First OUT*);



Pilha

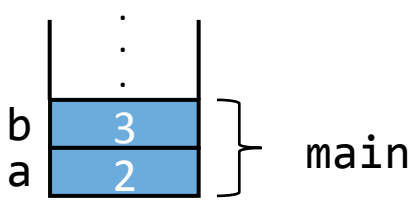
- Quando uma função a vai ser executada (incluindo `main()`), as variáveis declaradas nessa função (locais) são temporariamente armazenadas na memória em uma estrutura de pilha;
- Assim que a função finaliza a execução, suas variáveis são retiradas da pilha;
  - Sabemos que quando são tiradas da pilha, as variáveis são deletadas;
  - Essa região de memória fica disponível para “empilhar” novas variáveis;

Pilha - Exemplo

Esquema simplificado do empilhamento de variáveis

```
1  void soma(int a, int b){
2     int resultado;
3     resultado = a + b;
4
5     printf("O resultado e': %d", resultado);
6  }
7
8  int main(){
9     int a = 2, b = 3;
10
11    soma(a,b);
12
13    return 0;
14 }
```

Executando das linhas 8 a 9

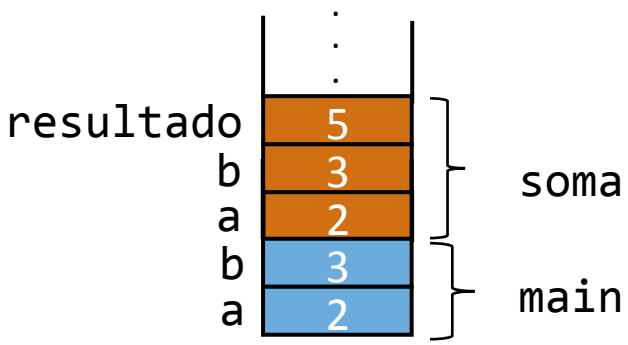


Pilha - Exemplo

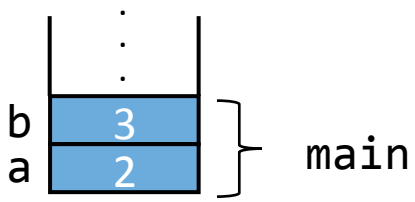
Esquema simplificado do empilhamento de variáveis

```
1  void soma(int a, int b){
2     int resultado;
3     resultado = a + b;
4
5     printf("O resultado e': %d", resultado);
6  }
7
8  int main(){
9     int a = 2, b = 3;
10
11    soma(a, b);
12
13    return 0;
14 }
```

Executando das linhas 11, 1 a 5



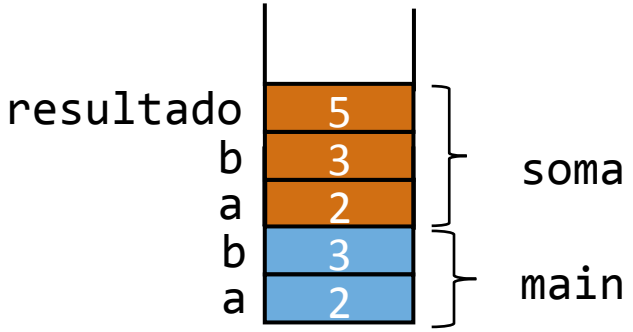
Executando das linha 6



Pilha

- Qual a vantagem de usar pilha para armazenar variáveis?
  - A memória é gerenciada automaticamente;
  - O programador não precisa alocar memória manualmente e liberar quando não mais necessária;
  - O próprio funcionamento da pilha torna o processo de gerenciar as variáveis mais ágil;

- Erro comum ao usar variáveis empilhadas:
  - Tentar acessar variável fora de seu escopo;

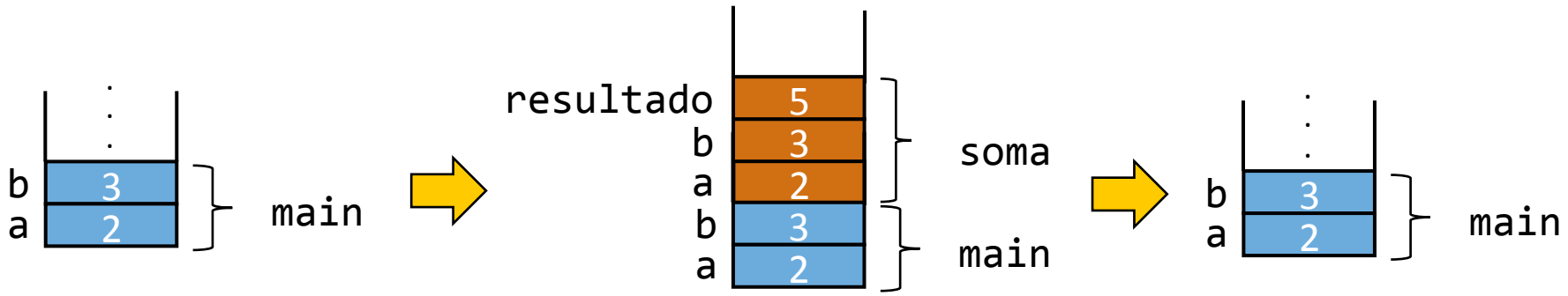


Tentar acessar a variável `resultado` na função `main()`

Pilha

Tentar acessar a variável `resultado` na função `main()`

```
1  void soma(int a, int b){
2     int resultado;
3     resultado = a + b;
4  }
5
6  int main(){
7     int a = 2, b = 3;
8
9     soma(a, b);
10    printf("O resultado e': %d", resultado);
11
12    return 0;
13 }
```



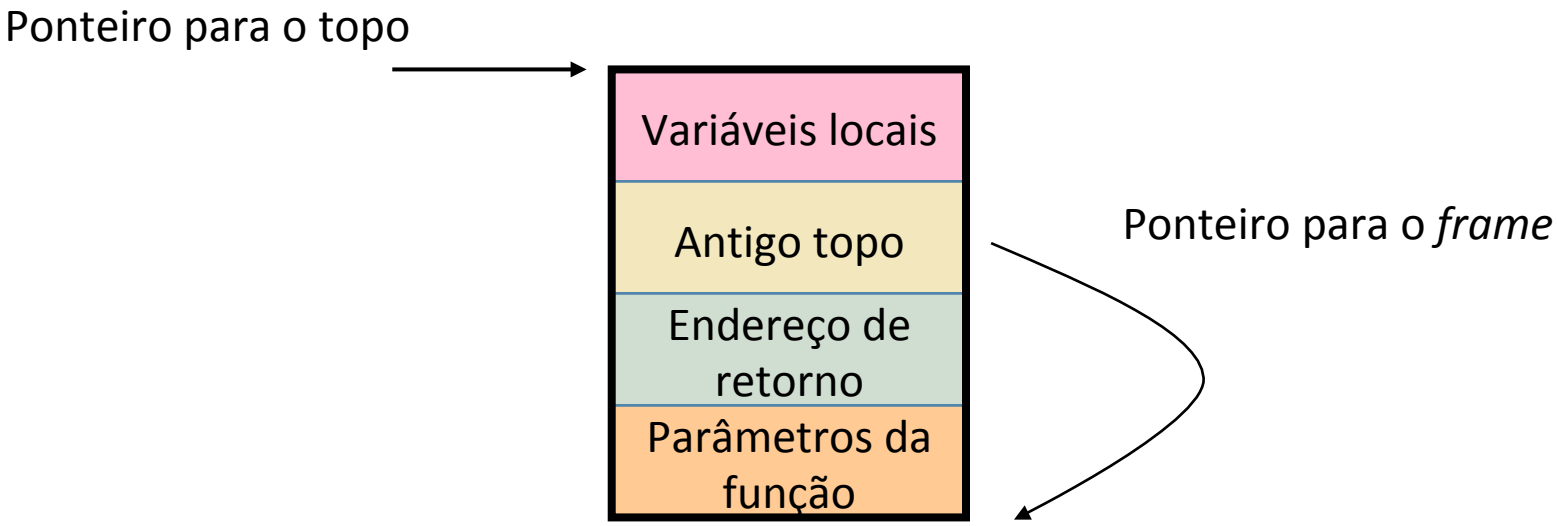
Pilha

- Existem um limite do tamanho das variáveis a serem empilhadas e esse tamanho é definindo pelo sistema operacional;

- Resumo sobre a pilha de memória:
  - Ela aumenta e diminui quando funções empilham ou desempilham variáveis locais;
  - O programador não precisa gerenciar a memória. Variáveis são alocadas e liberadas automaticamente;
  - A pilha tem um tamanho limite;
  - As variáveis só existem enquanto a função em que foram criadas está em execução.

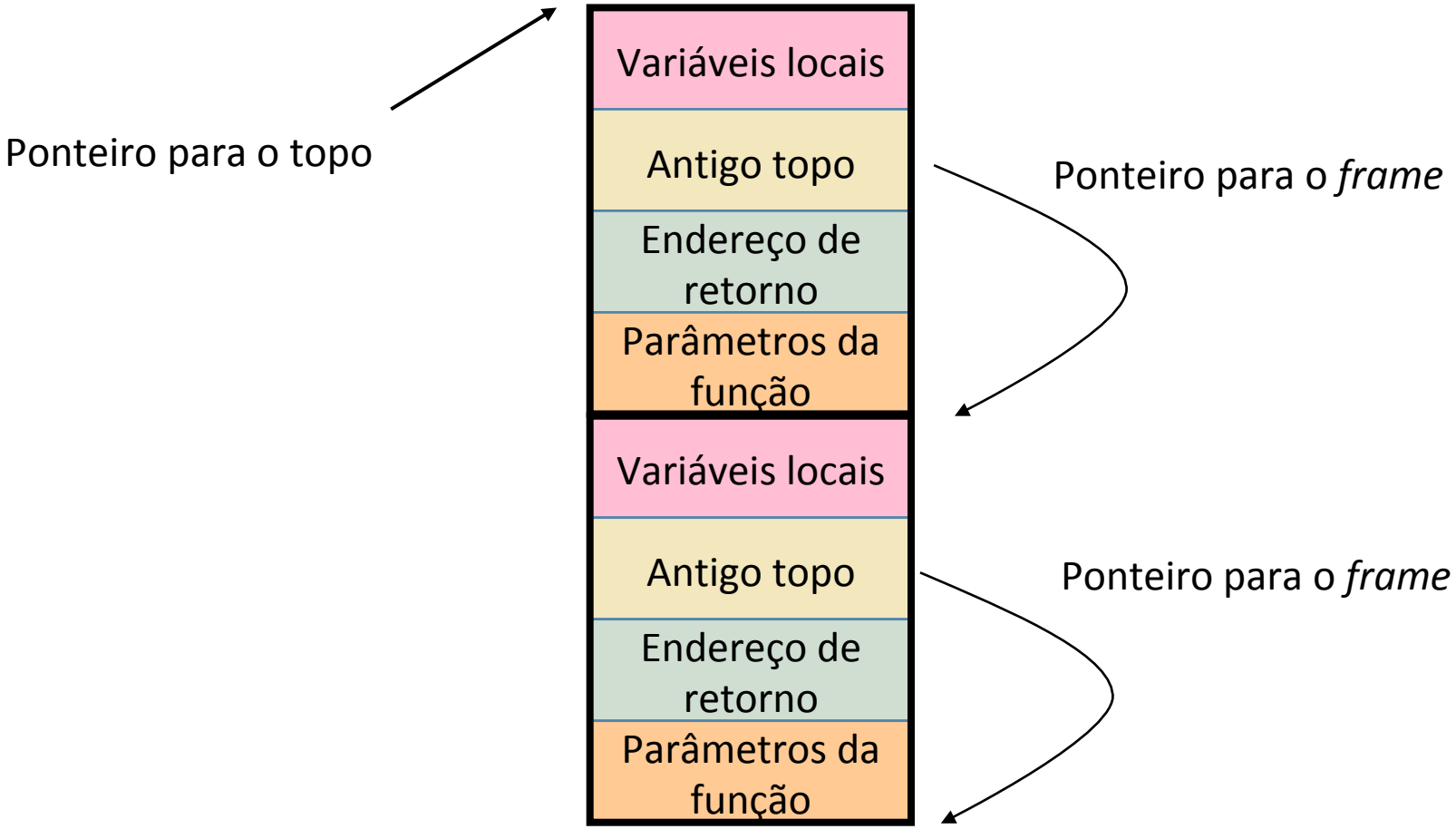
Pilha

- Nos slides anteriores, tratamos cada escopo da função dentro da pilha de maneira simplificada. Uma versão mais completa é apresentada a seguir



Pilha

- Nos slides anteriores, tratamos cada escopo da função dentro da pilha de maneira simplificada. Uma versão mais completa é apresentada a seguir



Heap

- O *Heap* é uma área da memória geralmente maior que a stack;
- Essa região não é automaticamente gerenciada;
  - Para usá-la o programador deve realizar **ALOCAÇÃO DINÂMICA** de varáveis;
  - Não existe restrição de tamanho de variáveis (exceto pela capacidade de armazenamento do *hardware*);
  - Pode-se criar uma estrutura que pode ir crescendo de acordo com a necessidade do programa;
  - Geralmente o acesso de variáveis no *Heap* é “mais lento” que a *stack* e deve-se usar ponteiros para acessa-las;
  - Não existe o conceito de topo do *Heap*;
  - Diferentemente da pilha, as variáveis alocadas na *Heap* são acessíveis para qualquer função, em qualquer parte do programa
    - Variáveis globais.

Stack vs Heap

Stack	Heap
Acesso rápido	Acesso relativamente mais lento
Não precisa alocar e desalocar espaço para variáveis	Programador deve gerenciar a memória usada pela aplicação. Pode gerar fragmentação.
Variáveis locais	Variáveis globais
Tamanho da pilha limitado pelo SO	Limitado apenas pela capacidade de armazenamento
Variáveis não podem ter seu tamanho alterado, por exemplo, vetores e matrizes	A aplicação pode alterar o tamanho do espaço usado para armazenar as variáveis

Stack vs Heap

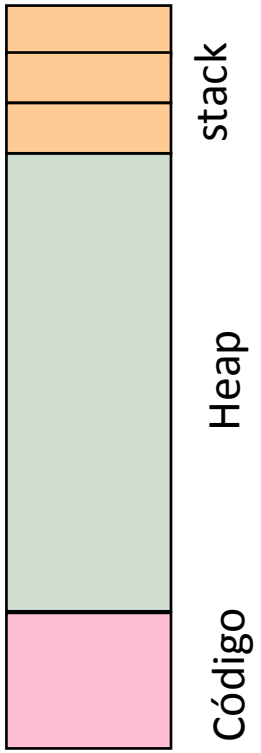
NÃO É UMA REGRA!!!

Quando usar *stack*:

- Quando se está usando variáveis relativamente pequenas;
- Quando as variáveis devem existir apenas enquanto a função onde elas foram definidas está ativa;

Quando usar *Heap*:

- Quando se deseja manipular variáveis extremamente grandes;
- Quando se deseja que as variáveis durem um longo período (e não estejam ligadas a vida de uma função);
- Se as variáveis precisarem aumentar ou diminuir de tamanho (por exemplo, vetores e matrizes);



Exercícios

- Mostre o estado da pilha a cada mudança de escopo

```c
void troca(int a, int b){
     int aux;
     aux = a;
     a = b;
     b = aux;
}
void main(){
     int x = 2, y = 5;
     troca(x, y);

     printf("%d %d", x, y);
}
```

(a)

```c
int fatorial(int n){
    int i, fat;
    for(fat = 1; n > 1; n=n - 1)
         fat = fat * n;

    return fat;
}

float soma(int n){
     int i;
     float res = 1;
     for(i = 1;i <= n;i++)
         res += 1/(float)fatorial(i);

     return res;
}

void main(){
     int n=3;
     float resultado;
     resultado = soma(n);
     printf("%f", resultado);
}
```

(b)

Exercícios

- Mostre o estado da pilha a cada mudança de escopo

```
1  int fatorial(int n){
2      int fat;
3      if(n <= 1)
4         return 1;
5      else{
6         fat = n * fatorial(n-1);
7         return fat;
8      }
9  }
10
11 Int main(){
12      int n=3, resultado;
13      resultado = fatorial(n);
14      printf("%d", resultado);
15
16     return 0;
17 }
```

(c)