Ponteiros em C

Conceitos, Aritmética, Alocação Dinâmica e Boas Práticas

02.08.2025

Gabriela Cota, Jéssica Pereira, Lucas Ramos, Mayara Barbosa & Rian Carlos

Conteúdos

1 Uso de &(address-of) e *(dereference). 3
2 Relação entre arrays e ponteiros. 8
3 Diferença entre char s[] e const char *. 15
4 Função swap com ponteiros. 20
5 Função que aloca dinamicamente um array com T** (ponteiro para ponteiro). 26
6 Ponteiro para função: exemplo com `qsort` 32
7 Diagramas 38
8 Três Armadilhas Comuns ao Trabalhar com Ponteiros 44

1 Uso de &(address-of) e *(dereference).

O que são os operadores & e *?

- `&` (address-of): obtém o endereço de memória de uma variável.
- `*` (dereference): acessa ou modifica o valor armazenado no endereço apontado por um ponteiro.

Essenciais para acesso indireto à memória e para diversas APIs em C.

Passo a passo


```c
int numero = 42;                    // 1) variável comum
printf("valor: %d\n", numero);

printf("endereco: %p\n", (void*)&numero); // 2) &numero -> endereço

int *ponteiro_para_numero;          // 3) declara um ponteiro p/ int
ponteiro_para_numero = &numero;     // 4) armazena o endereço de numero

int lido = *ponteiro_para_numero;   // 5) lê indiretamente (dereference)
printf("lido via ponteiro: %d\n", lido);

*ponteiro_para_numero = 100;        // 6) modifica indiretamente
printf("novo valor de numero: %d\n", numero);
```

- `%p` imprime endereços; converta para `(void*)` por portabilidade.
- `*ponteiro` lê/escreve o valor na posição apontada.

Visualizando a relação


```
                [numero]      [ponteiro_para_numero]
                   42   <-- guarda &numero (um endereço)

*ponteiro_para_numero   ==  numero
&numero                 ==  ponteiro_para_numero (após a atribuição)
```

Saída esperada


```
1. Variável 'numero' declarada.
   - Valor de 'numero': 42
2. Usando o operador '&' (address-of).
   - Endereço de memória de 'numero': 0x...
3. Ponteiro 'ponteiro_para_numero' declarado.
4. O ponteiro agora armazena o endereço de 'numero'.
   - Valor do ponteiro: 0x...
5. Usando o operador '*' (dereference) para LER o valor.
   - Valor no endereço apontado: 42
6. Usando o operador '*' para MODIFICAR o valor.
   - Modificando o valor para 100 via ponteiro: *ponteiro_para_numero = 100;
7. Verificando o valor da variável 'numero' original.
   - Novo valor de 'numero': 100
```

2 Relação entre arrays e ponteiros.

A Conexão Fundamental

Em C, o nome de um array é, na maioria dos contextos, um **ponteiro constante** para o seu primeiro elemento.

Isto significa que `v` e `&v[0]` são equivalentes e apontam para o mesmo endereço de memória.

```c
int v[4] = {25, 50, 75, 100};

// O código abaixo imprimirá o mesmo endereço duas vezes:
printf("Endereco do array (v): %p\n", v);
printf("Endereco do primeiro elemento (&v[0]): %p\n", &v[0]);
```

Acesso: Índice vs. Ponteiro

A notação de colchetes (`v[i]`) é, na verdade, "açúcar sintático" para a aritmética de ponteiros.

A expressão `v[i]` é internamente convertida para `*(v + i)`.

```c
// As duas formas de acesso produzem o mesmo resultado.
for (int i = 0; i < 4; i++) {
    printf(
      "Acesso por indice v[%d] = %d | "
      "Acesso por ponteiro *(v+%d) = %d\n",
      i, v[i], i, *(v+i)
    );
}
```

Saída Esperada

```
v[0] = 25 | *(v+0) = 25
v[1] = 50 | *(v+1) = 50
...
```

Aritmética de Ponteiros

Quando incrementamos um ponteiro, ele não avança 1 byte, mas sim o tamanho do tipo para o qual ele aponta (`sizeof(tipo)`).

- Se `p` é um `int*` no endereço `0x100`, `p+1` aponta para `0x104` (assumindo `sizeof(int)` de 4 bytes).
- Se `p` é um `char*` no endereço `0x100`, `p+1` aponta para `0x101`.

```c
int *p = v; // p aponta para o endereço de v[0]

// (p+i) avança o ponteiro para o próximo elemento.
// *(p+i) acessa o valor nesse endereço.
printf("Endereco: %p | Valor: %d\n", (p+i), *(p+i));
```

Navegando e Modificando com Ponteiros

Podemos percorrer e até modificar os elementos de um array usando apenas um ponteiro.

```c
// Percorrendo o array
int *p;
for (p = v; p < v + 4; p++) {
    printf("Endereco: %p | Valor: %d\n", p, *p);
}

// Modificando valores
*v = 13;        // Altera v[0] para 13
*(v + 1) = 17;  // Altera v[1] para 17
```

Ponto de Atenção

C **não** impede que um ponteiro acesse memória fora dos limites de um array.

```c
int v[4];
int *p = v;

// Isso compila, mas é um erro grave!
// Estamos escrevendo em uma área de memória que não nos pertence.
*(p + 10) = 999;
```

Este é um dos erros mais comuns e perigosos em C, podendo causar falhas de segmentação (**segmentation faults**) ou corrupção de dados.

3 Diferença entre char s[] e const char *.

`char s[]` — Um Array Mutável na Stack

- Declaração típica: `char s[] = "Gabriela";`
- O compilador:
  1. Aloca espaço na **stack** para o array (`sizeof("Gabriela")+1`).
  2. Copia a string literal para esse espaço.
- O array é **independente** da string literal e totalmente **mutável**.

```c
int main() {
    char s[] = "Gabriela"; // array mutável na stack
    printf("Nome original: %s\n", s);

    s[0] = 'g'; // permitido (modifica cópia local)
    printf("Nome modificado: %s\n", s);
    printf("Tamanho de s[]: %zu bytes\n", sizeof(s));

    return 0;
}
```

Saída Esperada

```
Nome original: Gabriela
Nome modificado: gabriela
Tamanho de s[]: 9 bytes
```

- O `sizeof(s)` retorna o **tamanho do array** em memória (9 bytes = 8 caracteres + `\0`).

`const char *s` — Ponteiro para String Literal Imutável

- Declaração típica: `const char *s = "Gabriela";`
- A string literal é armazenada em uma **área somente leitura** do programa.
- A variável `s` é apenas um **ponteiro** na stack que aponta para essa área.
- Tentar modificar o conteúdo resulta em **erro de compilação** ou **segmentation fault**.

```c
int main() {
    const char *s = "Gabriela"; // ponteiro para literal em memória só-leitura

    printf("Nome: %s\n", s);

    // s[0] = 'g'; // ERRO! não é permitido modificar literal

    printf("Tamanho do ponteiro s: %zu bytes\n", sizeof(s));

    return 0;
}
```

Saída Esperada


```
Nome: Gabriela
Tamanho do ponteiro s: 8 bytes
```

- O `sizeof(s)` retorna apenas o **tamanho do ponteiro** (4 bytes em 32 bits, 8 bytes em 64 bits).
- O conteúdo da string literal é **imutável**.

4 Função swap com ponteiros.

Por que usar swap com ponteiros?

- Passagem por referência: a função recebe ENDEREÇOS, não cópias de valores.
- Permite modificar, dentro da função, as variáveis originais do chamador.
- Padrão essencial para manipular dados em C.

Protótipo e funcionamento


```c
// Recebe dois ponteiros para int e troca os valores apontados
void swap(int* ptr_a, int* ptr_b) {
    printf("  [Dentro da função swap]\n");
    printf("  - Endereço recebido em ptr_a: %p\n", (void*)ptr_a);
    printf("  - Endereço recebido em ptr_b: %p\n", (void*)ptr_b);

    int temp = *ptr_a;  // 1) guarda o valor apontado por ptr_a
    *ptr_a = *ptr_b;    // 2) copia o valor apontado por ptr_b para ptr_a
    *ptr_b = temp;      // 3) restaura o valor antigo de ptr_a em ptr_b

    printf("  - Troca realizada dentro da função.\n");
}
```

- `ptr_a` e `ptr_b` são `int*`: acessamos os valores com `*` (dereference).
- Ao modificar `*ptr_a` e `*ptr_b`, alteramos as variáveis originais.

Uso na prática


```c
int valor1 = 10;
int valor2 = 99;

printf("Valores ANTES da troca:\n");
printf(" - valor1 = %d (no endereço %p)\n", valor1, (void*)&valor1);
printf(" - valor2 = %d (no endereço %p)\n\n", valor2, (void*)&valor2);

printf("Chamando swap(&valor1, &valor2) ...\n");
swap(&valor1, &valor2);  // passamos os ENDEREÇOS com '&'

printf("Valores DEPOIS da troca:\n");
printf(" - valor1 = %d\n", valor1);
printf(" - valor2 = %d\n", valor2);
```

Saída esperada


```
Valores ANTES da troca:
 - valor1 = 10 (no endereço 0x...)
 - valor2 = 99 (no endereço 0x...)

Chamando a função swap() e passando os ENDEREÇOS de valor1 e valor2...
  [Dentro da função swap]
  - Endereço recebido em ptr_a: 0x...
  - Endereço recebido em ptr_b: 0x...
  - Troca realizada dentro da função.
...retornamos para a main.

Valores DEPOIS da troca:
 - valor1 = 99
 - valor2 = 10
```

Pontos de atenção

- Passe os ENDEREÇOS com `&` (ex.: `swap(&a, &b)`), não os valores.
- Não dereferencie ponteiros nulos; valide entradas se necessário.
- Ao imprimir endereços com `%p`, converta para `(void*)` por portabilidade.
- Para outros tipos, ajuste a assinatura (ex.: `void swap_double(double*, double*)`).

5 Função que aloca dinamicamente um array com T (ponteiro para ponteiro).**

O que é uma Matriz Dinâmica?

Em C, uma matriz (array 2D) pode ser representada como um "ponteiro para ponteiro", por exemplo `char**`.

- Diferente de uma matriz estática (`char matriz[5][10]`), suas dimensões (`linhas` e `colunas`) podem ser definidas em **tempo de execução**.
- A memória é alocada na **heap**, não na **stack**.
- Para isso, usamos `malloc` para solicitar a memória e `free` para liberá-la.

A Estrutura: Array de Ponteiros

A ideia central é que uma matriz é um **array de ponteiros**, onde cada ponteiro aponta para uma **linha**.

- `matriz` (`char**`): Ponteiro que aponta para o início de um array de ponteiros.
- `matriz[i]` (`char*`): Cada elemento desse array é um ponteiro que aponta para o início de uma linha (um array de `char`).
- `matriz[i][j]` (`char`): O caractere na linha `i` e coluna `j`.

Alocando a Matriz: Passo a Passo

A alocação ocorre em duas fases:

```c
char **inicia_matriz(int linhas, int colunas) {
    // 1. Aloca um array de ponteiros (um para cada linha).
    char **matriz = malloc(linhas * sizeof(*matriz)); // ou sizeof(char*)
    if (matriz == NULL) { /* Tratar erro */ }

    // 2. Para cada linha, aloca um array de chars (as colunas).
    for (int i = 0; i < linhas; i++) {
        matriz[i] = malloc(colunas * sizeof(*matriz[i])); // ou sizeof(char)
        if (matriz[i] == NULL) { /* Tratar erro */ }

        // Preenche a matriz...

    return matriz;
}
```

Liberando a Memória: O Processo Inverso

Para evitar vazamentos de memória, a memória alocada com `malloc` deve ser liberada com `free`.

O processo é o **inverso** da alocação:

- Liberar cada uma das linhas.
- Liberar o array de ponteiros.

```c
void libera_matriz(char **matriz, int linhas) {
    if (!matriz) return;

    // 1. Libera cada linha (array de colunas).
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);
    }
    // 2. Libera o array de ponteiros.
    free(matriz);
}
```

Saída Esperada

- **Saída (para 5 linhas e 10 colunas):**

```
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
```

6 Ponteiro para função: exemplo com `qsort`

A função `qsort`

A função `qsort` da biblioteca padrão (`stdlib.h`) é uma função de ordenação genérica.

- **Genérica** porque pode ordenar arrays de **qualquer tipo de dado**: inteiros, floats, structs, e até mesmo strings (que são arrays de `char`).
- Para conseguir essa flexibilidade, `qsort` não sabe como comparar os elementos do array.
- Nós é que precisamos fornecer a lógica de comparação através de um **ponteiro para função**.

Sua assinatura é:

```c
void qsort(
  void *base,
  size_t num,
  size_t size,
  int (*compar)(const void *, const void *)
);
```

O problema: como ordenar strings?

Temos um array de strings que queremos ordenar em ordem alfabética.

```
char valores_string[5][10] = {
  "Jessica", "Lucas", "Rian", "Gabriela", "Mayara"
};
```

A função `strcmp` (de `string.h`) sabe como comparar duas strings. Mas sua assinatura é `int strcmp(const char *, const char *)`.

A `qsort` espera uma função com a assinatura `int (*compar)(const void *, const void *)`.

Não podemos passar `strcmp` diretamente para `qsort` devido à incompatibilidade de tipos dos ponteiros.

A solução: uma função “wrapper”

Criamos uma função “wrapper” (ou adaptadora) que compatibiliza a `strcmp` com o que a `qsort` espera.

```c
#include <string.h>

// Esta função segue a assinatura exigida por qsort.
int compara_string(void const *ponteiro_a, void const *ponteiro_b) {
    // 1. Converte os ponteiros genéricos (void *)
    //    para o tipo que realmente estamos trabalhando (char *).
    char const *primeira = (char const *)ponteiro_a;
    char const *segunda = (char const *)ponteiro_b;

    // 2. Usa strcmp para fazer a comparação real.
    return strcmp(primeira, segunda);
}
```

Juntando tudo: a chamada da `qsort`

Agora, no `main`, podemos chamar a `qsort` e passar nossa função de comparação.

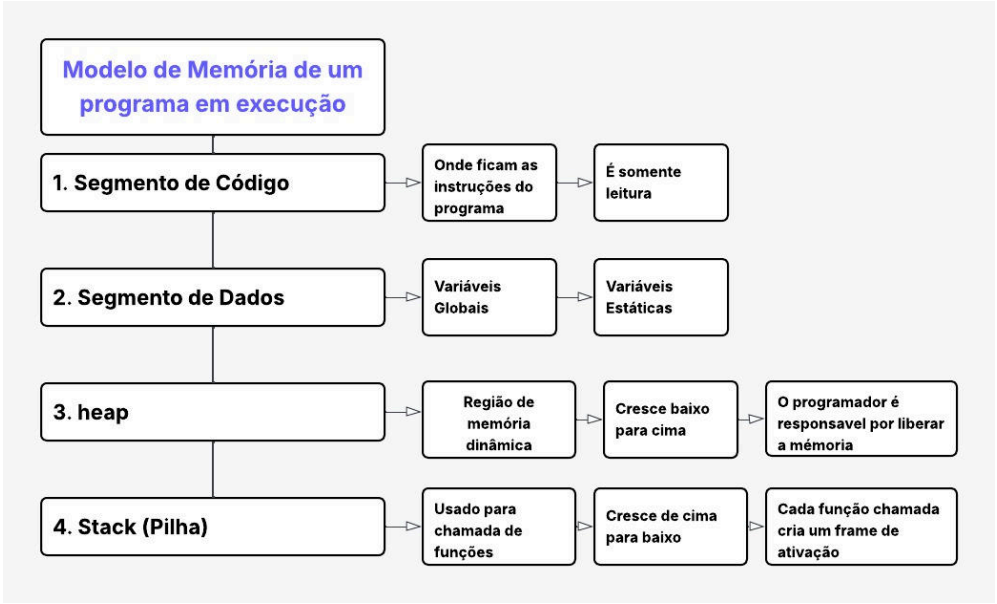
```c
int main() {
    char valores_string[5][10] = {"Jessica", "Lucas", "Rian", "Gabriela", "Mayara"};

    // ... código para imprimir o array desordenado ...

    qsort(
      valores_string, // 1. O array a ser ordenado.
      5,              // 2. O número de elementos no array.
      10,             // 3. O tamanho de cada elemento (em bytes).
      compara_string  // 4. O ponteiro para nossa função de comparação.
    );

    // ... código para imprimir o array ordenado ...
    return 0;
}
```

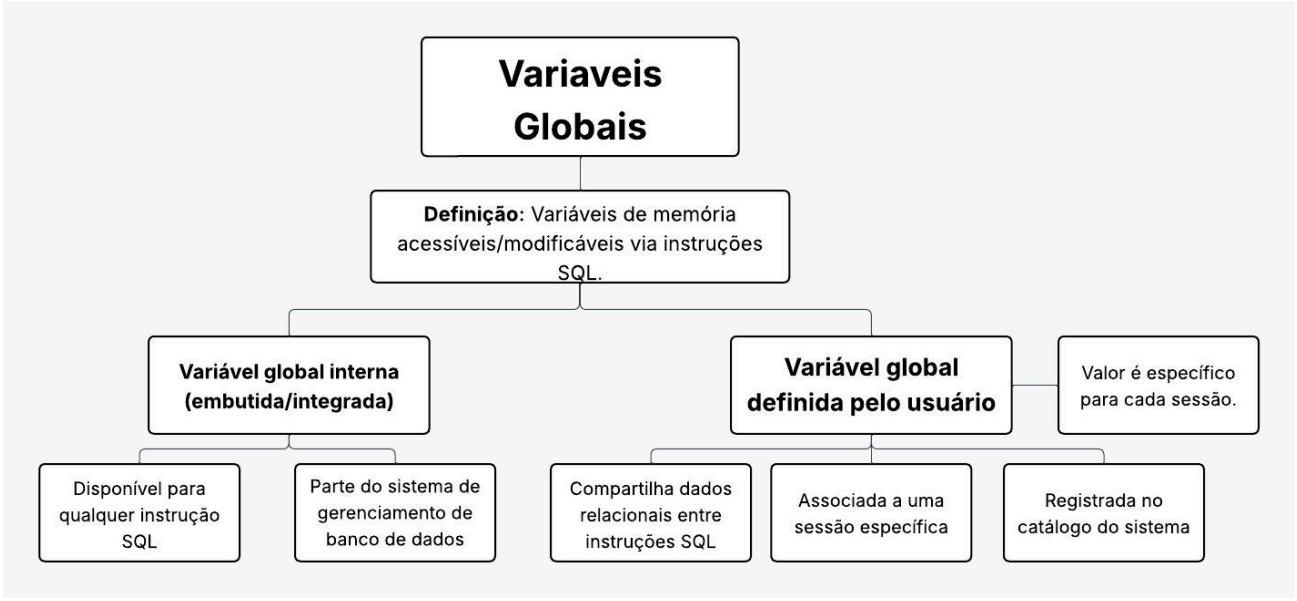
Saída Esperada

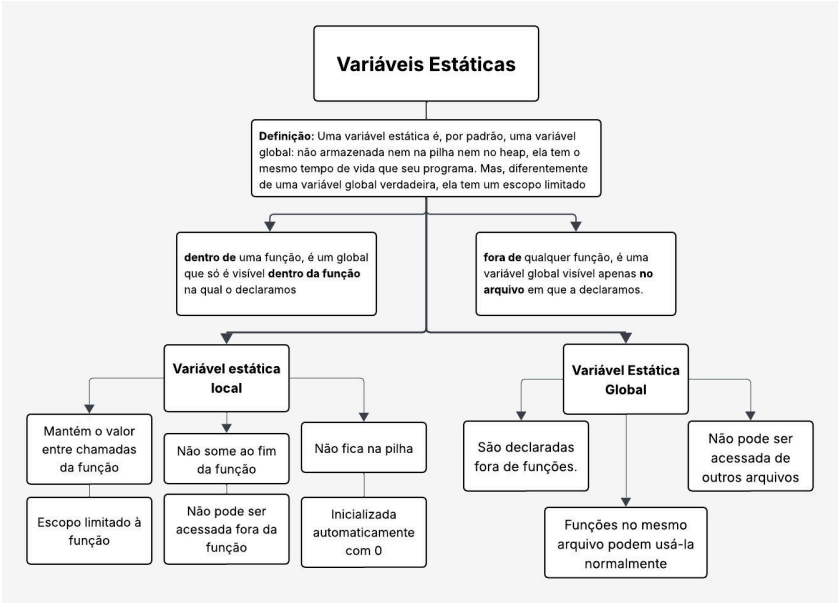
- **Saída:**

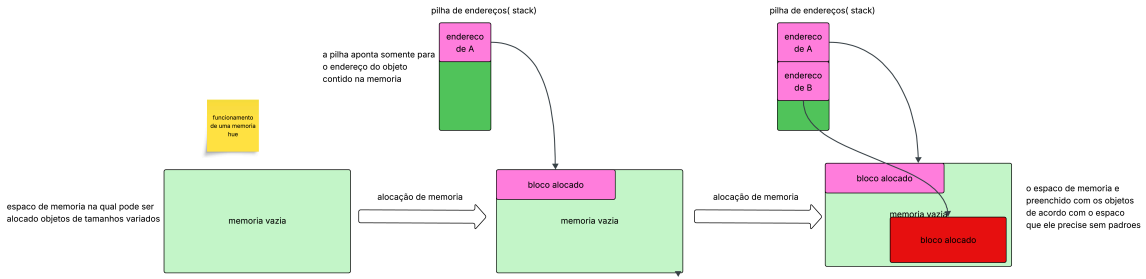
```
Strings desordenadas: Jessica Lucas Rian Gabriela Mayara
Strings ordenadas: Gabriela Jessica Lucas Mayara Rian
```

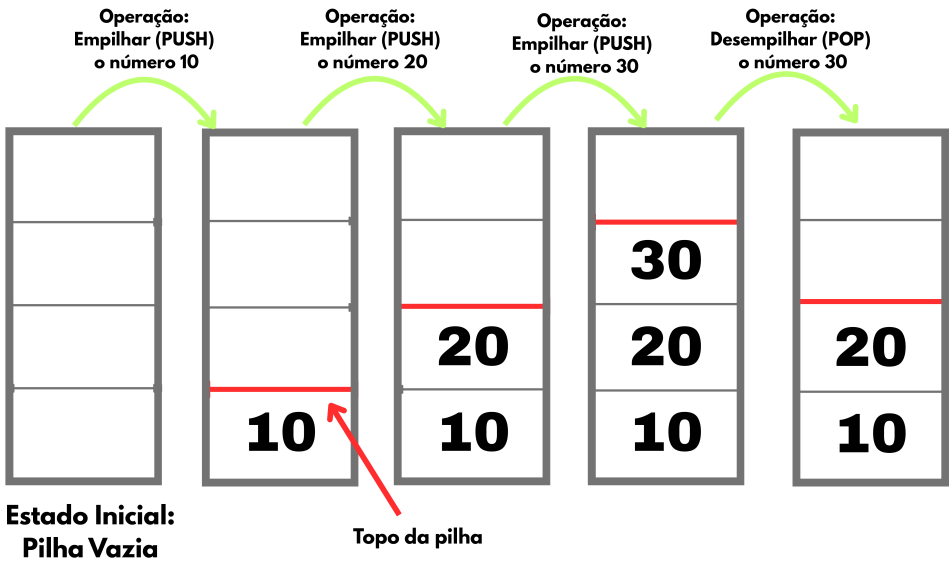
7 Diagramas


Modelo de Memória de um programa em execução
1. Segmento de Código
Onde ficam as instruções do programa
É somente leitura
2. Segmento de Dados
Variáveis Globais
Variáveis Estáticas
3. heap
Região de memória dinâmica
Cresce baixo para cima
O programador é responsavel por liberar a mémoria
4. Stack (Pilha)
Usado para chamada de funções
Cresce de cima para baixo
Cada função chamada cria um frame de ativação


Variaveis Globais
Definição: Variáveis de memória acessíveis/modificáveis via instruções SQL.
Variável global interna (embutida/integrada)
Disponível para qualquer instrução SQL
Parte do sistema de gerenciamento de banco de dados
Variável global definida pelo usuário
Valor é específico para cada sessão.
Compartilha dados relacionais entre instruções SQL
Associada a uma sessão específica
Registrada no catálogo do sistema


Variáveis Estáticas
Definição: Uma variável estática é, por padrão, uma variável global: não armazenada nem na pilha nem no heap, ela tem o mesmo tempo de vida que seu programa. Mas, diferentemente de uma variável global verdadeira, ela tem um escopo limitado
dentro de uma função, é um global que só é visível dentro da função na qual o declaramos
fora de qualquer função, é uma variável global visível apenas no arquivo em que a declaramos.
Variável estática local
Variável Estática Global
Mantém o valor entre chamadas da função
Não some ao fim da função
Não fica na pilha
Escopo limitado à função
Não pode ser acessada fora da função
Inicializada automaticamente com 0
São declaradas fora de funções.
Não pode ser acessada de outros arquivos
Funções no mesmo arquivo podem usá-la normalmente


funcionamento de uma memoria hue
espaco de memoria na qual pode ser alocado objetos de tamanhos variados
memoria vazia
alocação de memoria
pilha de endereços(stack)
endereco de A
a pilha aponta somente para o endereço do objeto contido na memoria
bloco alocado
memoria vazia
alocação de memoria
pilha de endereços(stack)
endereco de A
endereco de B
bloco alocado
memoria vazia
bloco alocado
o espaco de memoria e preenchido com os objetos de acordo com o espaco que ele precise sem padroes


Operação:
Empilhar (PUSH)
o número 10
Operação:
Empilhar (PUSH)
o número 20
Operação:
Empilhar (PUSH)
o número 30
Operação:
Desempilhar (POP)
o número 30
10
20
10
30
20
10
20
10
Estado Inicial:
Pilha Vazia
Topo da pilha

8 Três Armadilhas Comuns ao Trabalhar com Ponteiros

1. Ponteiros Não Inicializados

Um dos problemas mais comuns ocorre quando um ponteiro não recebe um valor inicial. Sem um endereço válido, ele pode conter um valor aleatório (lixo de memória) e apontar para uma área de memória imprevisível.

- Impacto no Código: Se um programa tenta desreferenciar um ponteiro não inicializado, a execução pode levar a falhas de segmentação, corrupção de dados ou comportamento imprevisível.

1. Ponteiros Não Inicializados (ii)

Código com o problema:

```c
#include <stdio.h>

int main() {
    int* p; // Ponteiro não inicializado

    // Tentativa de acessar um endereço desconhecido
    *p = 10;

    return 0;
}
```

Assim, é indicado que se atribua um valor a um ponteiro tão logo o ponteiro é declarado.

- Solução: Uma prática fundamental é sempre inicializar os ponteiros. Se ainda não houver um endereço para atribuir, deve-se usar NULL. Isso torna o ponteiro seguro para ser verificado antes de ser usado.

1. Ponteiros Não Inicializados (iii)

Código correto:

```c
#include <stdio.h>

int main() {
    int* p = NULL;

    // Verificação de segurança antes de usar
    if (p != NULL) {
        *p = 10;
    }

    return 0;
}
```

2. Vazamentos de Memória

Um vazamento de memória ocorre quando a memória que foi alocada dinamicamente com funções como malloc ou new não é liberada.

- Impacto no Código: A memória alocada permanece ocupada e inacessível para o programa. Com o tempo, o consumo contínuo pode esgotar os recursos do sistema, levando a lentidão e, eventualmente, a falhas.

2. Vazamentos de Memória (ii)

Código com o problema:

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

void alocar() {
    int* dados = (int*)malloc(100 * sizeof(int));
    // A memória é alocada, mas a função termina e o ponteiro é perdido.
}

int main() {
    alocar(); // A memória "vaza" aqui
    return 0;
}
```

- Solução: É essencial garantir que cada alocação de memória tenha uma liberação correspondente, assim cada alocação com malloc ou new precisa ter um free ou delete correspondente.

2. Vazamentos de Memória (iii)

Código correto:

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

void alocar() {
    int* dados = (int*)malloc(100 * sizeof(int));
    if (dados != NULL) {
        free(dados); // A memória é liberada antes de sair da função
    }
}

int main() {
    alocar();
    return 0;
}
```

3. Ponteiros “Pendurados” (Dangling Pointers)

Essa armadilha acontece quando um ponteiro ainda aponta para um endereço de memória que já foi liberado e não é mais válido.

- Impacto no Código: Tentar usar um ponteiro pendurado pode causar a corrupção de outros dados ou uma falha de segmentação, pois a área de memória pode já ter sido realocada para outro fim.

3. Ponteiros “Pendurados” (Dangling Pointers) (ii)

Código com o problema:

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

int main() {
    int* p = (int*)malloc(sizeof(int));
    *p = 20;

    free(p); // A memória é liberada.

    // 'p' ainda contém o endereço antigo.
    printf("Valor: %d\n", *p); // Comportamento imprevisível.

    return 0;
}
```

3. Ponteiros “Pendurados” (Dangling Pointers) (iii)

- Solução: Após liberar a memória, o ponteiro deve ser imediatamente definido para NULL. Isso previne que ele seja usado acidentalmente.

3. Ponteiros “Pendurados” (Dangling Pointers) (iv)

Código correto:

```c
int main() {
    int* p = (int*)malloc(sizeof(int));
    *p = 20;

    free(p);
    p = NULL; // O ponteiro agora é seguro.

    if (p != NULL) {
        printf("Valor: %d\n", *p);
    } else {
        printf("Ponteiro nulo. Acesso negado.\n");
    }

    return 0;
}
```

Referências

- https://www.ime.usp.br/~pf/algoritmos/aulas/pont.html
- https://www.geeksforgeeks.org/c/c-pointers/