Estruturas de Dados: Ponteiros e Alocação dinâmica de memória

Rafael Viana de Carvalho

Ponteiros

- Permitem manipulação direta de endereços de memória no C
  - São amplamente utilizados para
    - Manipulação eficiente de memória
    - Passagem de parâmetros por referência
    - Alocação dinâmica e estruturação de dados complexos.
- Variáveis do tipo ponteiro
  - Armazenam endereços de memória ao invés de valores diretos
  - É possível definir um ponteiro para cada tipo do C que seja capaz de armazenar endereços de memória em que existem valores do tipo correpondente

Ponteiros

- Funcionamento
  - Cada variável em C ocupa um espaço na memória e tem um **endereço** associado a ela.
  - Um ponteiro armazena esse **endereço de memória**
  - Pode ser usado para acessar ou modificar o valor armazenado nessa localização
- Sintaxe:
  - int a;
  - int *p; // p armazena endereço de memória em que há valor inteiro
- Exemplo:
  - int *ptr; → Declara um ponteiro ptr
  - ptr = &x; → ptr recebe o endereço de x
  - *ptr → O operador **de referência** acessa o valor armazenado no endereço apontado.

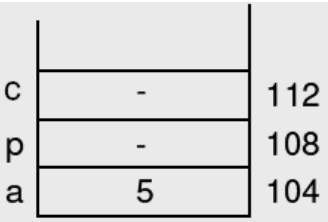
Operadores e Aritmética de ponteiros

- Operador & (“endereço de”)
  - Aplicado a variáveis, retorna o endereço da posição de memória reservada para variável
- Operador * (“conteúdo de”)
  - Aplicado a ponteiros, acessa o conteúdo de memória do endereço armazenado pela variável ponteiro
- Aritmética de Ponteiros
  - ptr + 1 move o ponteiro para o próximo inteiro (4 bytes adiante, se for um sistema de 32 bits)

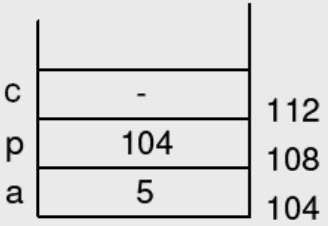
Exemplo

- int a; int* p; int c;

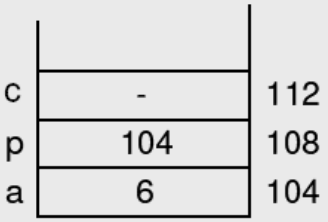
```
/ * a recebe o valor 5 */
a = 5;
```



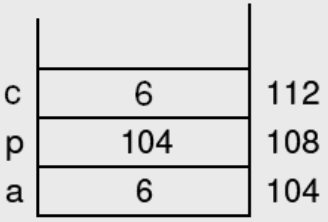

```
/* p recebe o endereço de a
   ou seja, p aponta para a */
p = &a;
```




```
/* posição de memória apontada por p
   recebe 6 */
*p = 6;
```

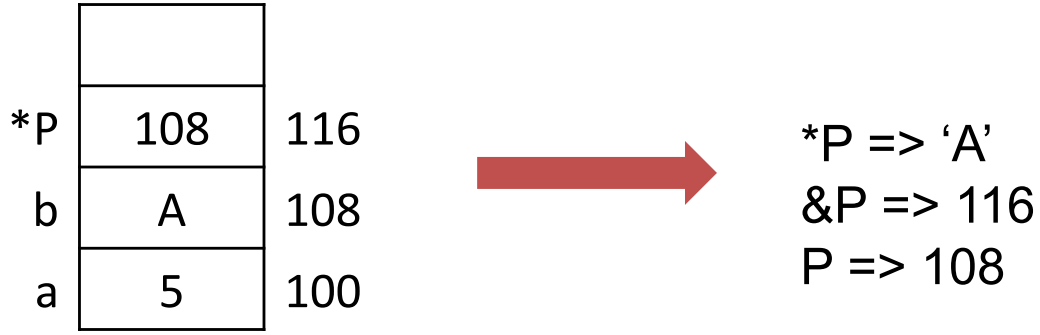



```
/* c recebe o valor armazenado
   na posição de memória apontada por p */
c = *p;
```



Operações com ponteiros

- *P (ponteiro P)
  - Indica o conteúdo da posição de memória que *P aponta
- &P (Posição de P)
  - Indica o endereço de memória que P foi alocado
- P (conteúdo P)
  - Indica o conteúdo armazenado em P, ou seja uma posição de memória.



Exemplos

- int main (void) {
  - int a;
  - int *p;
  - p = &a;
  - *p = 2;
  - printf (“ %d “, a);
  - return;
- }

Exemplos

- int main (void) {
  - int a;
  - int *p;
  - p = &a;
  - *p = 2;
  - printf (“ %d “, a);
  - return;
- }

- Imprime o valor 2

Exemplos

- int main (void) {
  - int a, b, *p;
  - a = 2;
  - *p = 3;
  - b = a + (*p);
  - printf (“ %d “, b);
  - return 0;
- }

Exemplos

- int main (void) {
  - int a, b, *p;
  - a = 2;
  - *p = 3;
  - b = a + (*p);
  - printf (“ %d “, b);
  - return 0;
- }
- ERRO!

Exemplos

- int main (void) {
  - int var = 1;
  - int *ptr;
  - ptr = &var;
  - *ptr = 3;
  - printf (“ \n Acesso direto: %d “, var);
  - printf (“ \n Acesso indireto: %d “, *ptr);
  - printf (“ \n Acesso direto: %d “, &var);
  - printf (“ \n Acesso indireto: %d “, ptr);
  - return 0;
- }

Exercícios

- 1.Quais serão os valores de x, y e p ao final do trecho de código abaixo?
  - int x, y, *p;
  - y = 0;
  - p = &y;
  - x = *p;
  - x = 4;
  - (*p)++;
  - --x;
  - (*p) += x;

Exercícios

- 2.Os programas (trechos de código) abaixo possuem erros. Qual(is)? Como deveriam ser?
  - a)
```
void main() {
        int x, *p;
        x = 100;
        p = x;
        printf("Valor de p: %d.\n", *p);
        }
```
  - b)
```
void troca (int *i, int *j) {
        int *temp;
        *temp = *i;
        *i = *j;
        *j = *temp;
        }
```

Exercícios

- 2.Os programas (trechos de código) abaixo possuem erros. Qual(is)? Como deveriam ser?
  - c) char *a, *b;
    a = "abacate";
    b = "uva”;
    if (a < b)
        printf ("%s vem antes de %s no dicionário", a, b);
    else
        printf ("%s vem depois de %s no dicionário", a, b);

Passagem de Parâmetro por Valor

- Na **passagem por valor**, uma cópia do argumento é passada para a função
  - Assim, qualquer modificação dentro da função **não altera a variável original**.

```c
#include <stdio.h>

void alterarValor(int a) {
    a = 20;  // Altera apenas a cópia local
}

int main() {
    int x = 10;
    alterarValor(x);
    printf("Valor de x após a função: %d\n", x);  // Ainda será 10
    return 0;
}
```

Passagem de Parâmetro por Referência

- Na **passagem por referência**, passamos o endereço da variável para a função
  - Assim, a função pode modificar diretamente o valor original.

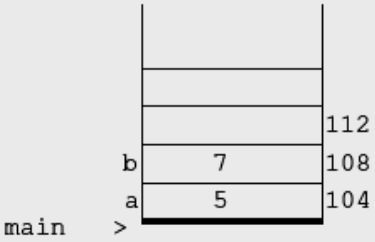
```c
#include <stdio.h>

void alterarPorReferencia(int *ptr) {
    *ptr = 20;  // Modifica o valor da variável original
}

int main() {
    int x = 10;
    alterarPorReferencia(&x);
    printf("Valor de x após a função: %d\n", x);  // Agora será 20
    return 0;
}
```

Exemplo

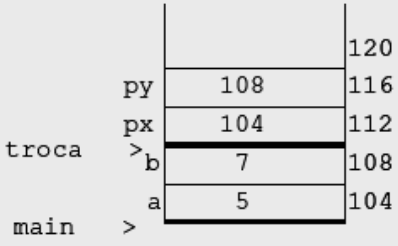
```
/* função troca */
#include <stdio.h>
void troca (int *px, int *py )
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
int main ( void )
{
   int a = 5, b = 7;
   troca(&a, &b);     /* passamos os endereços das variáveis */
   printf("%d %d \n", a, b);
   return 0;
}
```

Exemplo

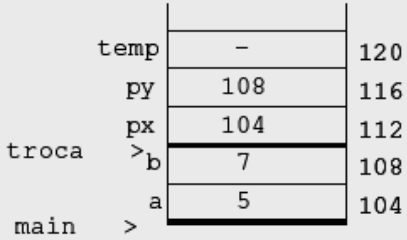
1 -Declaração das variáveis: a, b



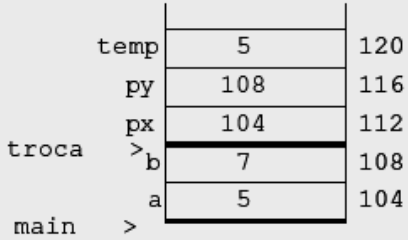
2 - Chamada da função: passa endereços



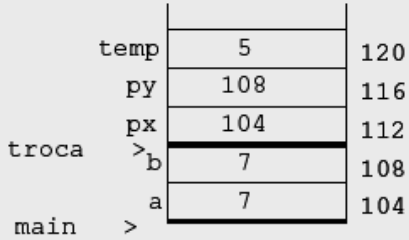
3 - Declaração da variável local: temp



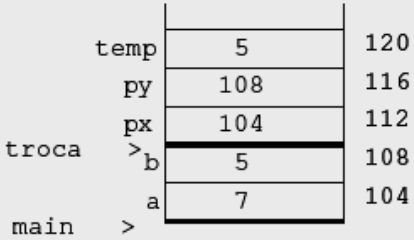
4 - temp recebe conteúdo de px



5 -Conteúdo de px recebe conteúdo de py



6 -Conteúdo de py recebe temp



Ponteiros e Arrays

- Em C, o nome de um array já é um **ponteiro para o seu primeiro elemento**
  - Podemos acessar os elementos de um array usando aritmética de ponteiros

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr;  // O ponteiro aponta para o primeiro elemento do array

    printf("Primeiro elemento: %d\n", *ptr);
    printf("Segundo elemento: %d\n", *(ptr + 1));
    printf("Terceiro elemento: %d\n", *(ptr + 2));

    return 0;
}
```

  - *(ptr + 1) acessa o segundo elemento do array

Ponteiros e Strings

- É possível manipular uma string usando ponteiros

```c
#include <stdio.h>

int main() {
    char str[] = "Hello";
    char *ptr = str;

    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;  // ptr percorre a string sem precisar de um índice.
    }

    printf("\n");  // Adiciona uma quebra de linha no final da saída
    return 0;
}
```

Ponteiros para Ponteiros

- Um **ponteiro para ponteiro** armazena o endereço de outro ponteiro
  - Esse conceito é útil para **alocação dinâmica de matrizes** e **manipulação avançada de memória**.

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;   // Ponteiro para inteiro
    int **pp = &p; // Ponteiro para ponteiro

    printf("Valor de x: %d\n", **pp); // Acessando o valor original
    return 0;
}
```

Alocação Dinâmica de Memória em C

- Permite que um programa solicite memória **durante a execução** (tempo de execução) e **libere essa memória quando não for mais necessária**
  - Essencial para criar estruturas de dados flexíveis, como **listas, árvores e vetores de tamanho variável**.

Tipo de Memória	Características
Memória Estática (Stack)	- Tamanho fixo (definido na compilação). - Automática (liberada quando a função termina). - Mais rápida.
Memória Dinâmica (Heap)	- Alocada em tempo de execução. - Tamanho pode ser variável. - Deve ser liberada manualmente (`free()`).

Funções para Alocação Dinâmica

- A biblioteca **<stdlib.h>** fornece quatro funções principais para manipular a memória dinâmica:
  - malloc( )
  - calloc( )
  - realloc( )
  - free( )

Funções para Alocação Dinâmica

- **malloc( ) – Alocação Simples**
  - **memory allocation** - aloca um bloco de memória, mas **não o inicializa**.
  - Retorna um **ponteiro genérico (void*)**, que precisa ser convertido ((int*)).

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

int main() {
    int *ptr;

    ptr = (int*) malloc(sizeof(int));  // Aloca espaço para um inteiro

    if (ptr == NULL) {
        printf("Erro ao alocar memória!\n");
        return 1;
    }

    *ptr = 42;  // Atribui um valor à memória alocada
    printf("Valor armazenado: %d\n", *ptr);

    free(ptr);  // Libera a memória
    return 0;
}
```

Funções para Alocação Dinâmica

- **calloc( ) – Alocação e Inicialização**
  - **Clear allocation** – funciona como **malloc( )** mas **inicializa a memória com zeros**

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

int main() {
    int *arr;
    int n = 5;  // Tamanho do array

    arr = (int*) calloc(n, sizeof(int));  // Aloca e inicializa com zeros

    if (arr == NULL) {
        printf("Erro ao alocar memória!\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);  // Todos os valores serão 0
    }

    free(arr);  // Libera a memória
    return 0;
}
```

Funções para Alocação Dinâmica

- **realloc( ) – Redimensionando memória**
  - Se um bloco de memória já foi alocado e **precisamos aumentar ou diminuir seu tamanho**, usamos realloc( ).

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

int main() {
    int *arr;
    int n = 3;

    arr = (int*) malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Erro ao alocar memória!\n");
        return 1;
    }

    // Preenchendo o array inicial
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    // Redimensiona para 5 elementos
    arr = (int*) realloc(arr, 5 * sizeof(int));
    if (arr == NULL) {
        printf("Erro ao realocar memória!\n");
        return 1;
    }

    // Novos elementos precisam ser atribuídos
    arr[3] = 4;
    arr[4] = 5;

    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    free(arr);
    return 0;
}
```

Funções para Alocação Dinâmica

- **free( ) – Liberando memória**
  - Toda memória alocada dinamicamente **deve ser liberada** com free() para evitar **vazamentos de memória.**
  - Importante:
    - Não tente acessar ptr depois de free(ptr). Isso pode causar **comportamento indefinido**.
    - Após free(), é boa prática definir ptr = NULL; para evitar referências inválidas.

Atividade – Vetor Dinâmico

- Crie um vetor dinâmico onde o usuário ira definir o tamanho inicial, depois preencha o vetor com valores aleatórios. Em seguida redimensione esse vetor preenchendo com novos valores se necessário. Por fim, exibas os valores na tela (não esqueça de liberar a memória)