Programação de Computadores

Raoni F. S. Teixeira

Aula 13 - Ponteiros e Alocação Dinâmica

1 Introdução

Ponteiros são variáveis especiais que armazenam endereços de memória. Em C, o conceito de ponteiro é explícito e fundamental para:

1. manipular estruturas de dados complexas,
2. alocar memória dinamicamente durante a execução do programa, e
3. permitir que funções modifiquem variáveis de outras funções.

A memória do computador pode ser vista como uma sequência de bytes, onde cada byte possui um **endereço** único. Cada variável ocupa um conjunto de bytes consecutivos na memória. Por exemplo, uma variável `char` ocupa 1 byte, enquanto uma variável `int` tipicamente ocupa 4 bytes.

2 Endereços de Memória

O operador `&` retorna o endereço de uma variável. Por exemplo, se `i` é uma variável inteira, então `&i` representa o endereço onde `i` está armazenada na memória.

Considere o seguinte exemplo:

```
int variavel = 90;
printf("Valor: %d\n", variavel);
printf("Endereco: %p\n", &variavel);
```

Este código imprime o valor armazenado na variável e seu endereço na memória. A memória RAM é organizada como uma sequência de bytes, e cada byte possui uma posição única (seu endereço). Se `e` é o endereço de um byte, então `e + 1` é o endereço do byte seguinte.

2.1 Tamanho das Variáveis

Diferentes tipos de variáveis ocupam quantidades distintas de bytes na memória. O operador `sizeof` retorna o número de bytes ocupados por um tipo ou variável:

```
printf("Tamanho␣de␣char:␣%lu\n", sizeof(char));    // 1 byte
printf("Tamanho␣de␣int:␣%lu\n", sizeof(int));      // 4 bytes
printf("Tamanho␣de␣double:␣%lu\n", sizeof(double)); // 8 bytes
```

3 Ponteiros

Um **ponteiro** é uma variável que armazena um endereço de memória. Declaramos um ponteiro usando o operador `*`. Por exemplo:

```
int *endereco;
```

Esta declaração cria um ponteiro chamado `endereco` que pode armazenar o endereço de uma variável inteira.

Um ponteiro pode ter o valor `NULL`, que representa um endereço inválido. A macro `NULL` está definida em `stdlib.h` e seu valor é 0 (zero) na maioria dos computadores.

3.1 Operador de Desreferenciação

O operador `*` (quando usado em uma expressão, não em uma declaração) acessa o **conteúdo** do endereço apontado pelo ponteiro. Dizemos que o operador *desreferencia* o ponteiro.

```
int *endereco;
int variavel = 90;
endereco = &variavel;
printf("Valor␣via␣ponteiro:␣%d\n", *endereco);
```

Neste exemplo, `endereco` armazena o endereço de `variavel`, e `*endereco` acessa o valor armazenado naquele endereço (90). A Figura 1 ilustra a relação entre ponteiros e variáveis.

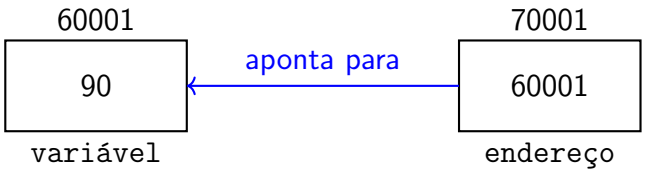

Figura 1: Representação de um ponteiro apontando para uma variável. Os endereços são fictícios.

Se um ponteiro p armazena o endereço de uma variável i, podemos dizer "p aponta para i" ou "p é o endereço de i". Neste caso, se p vale &i, então dizer "*p" é o mesmo que dizer "i".

3.2 Tipos de Ponteiros

Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros, ponteiros para registros, etc. O compilador precisa saber o tipo do ponteiro para calcular corretamente os endereços.

```
int *p;           // ponteiro para inteiro
char *c;          // ponteiro para char
double *d;        // ponteiro para double
struct reg *ptr;  // ponteiro para registro
int **r;          // ponteiro para ponteiro para inteiro
```

3.3 Aplicação Prática: Troca de Valores

Considere o problema de criar uma função que troque os valores de duas variáveis. A seguinte tentativa não funciona:

```
void troca_errada(int i, int j) {
    int temp = i;
    i = j;
    j = temp;
}
```

Este código não produz o efeito desejado porque a função recebe apenas *cópias* dos valores das variáveis. As modificações ocorrem apenas nas cópias locais, deixando as variáveis originais inalteradas. Este é um conceito fundamental: em C, argumentos são passados por valor.

A solução correta usa ponteiros para passar os endereços das variáveis:

```
void troca(int *p, int *q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Para usar esta função com variáveis i e j, chamamos troca(&i, &j). Agora a função recebe os endereços das variáveis e pode modificar diretamente seus valores na memória.

4 Alocação Dinâmica

Nem sempre sabemos antecipadamente quantas variáveis precisaremos durante a execução do programa. A alocação dinâmica permite criar variáveis durante a execução, conforme

necessário.

4.1 Organização da Memória

A memória de um programa é dividida em duas regiões principais:

- **Pilha (Stack):** Armazena variáveis locais declaradas dentro de funções. O compilador gerencia automaticamente essa memória, alocando espaço quando uma função é chamada e liberando quando ela termina. As variáveis são acessadas por nomes bem definidos.
- **Heap:** Armazena variáveis criadas dinamicamente pelo programador. Esta memória deve ser explicitamente alocada (com `malloc`) e liberada (com `free`) pelo programador. O acesso é feito através de ponteiros.

A Figura 2 ilustra a organização da memória de um programa.

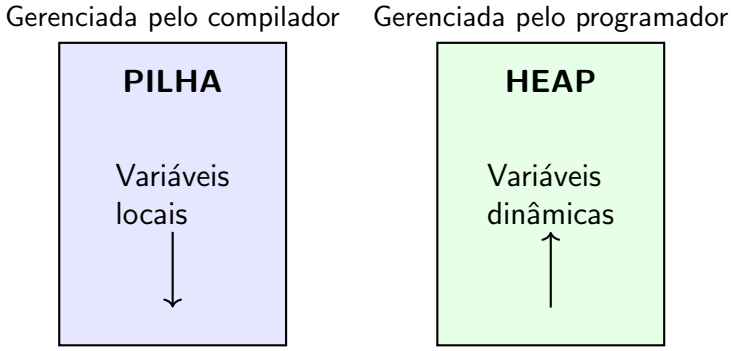

Figura 2: Organização da memória de um programa.

4.2 Função malloc

A função `malloc` (memory allocation) aloca um bloco de memória no heap. Ela recebe como parâmetro o número de bytes a serem alocados e retorna um ponteiro para o início do bloco alocado. Se não houver memória disponível, retorna `NULL`.

```
#include <stdlib.h>

int *ponteiro;
ponteiro = malloc(sizeof(int));
if (ponteiro == NULL) {
    printf("Erro: memoria insuficiente\n");
    exit(1);
```

```
}
*ponteiro = 13;
printf("Valor: %d\n", *ponteiro);
```

O operador `sizeof` retorna o número de bytes ocupados por um tipo. É essencial verificar se `malloc` retornou `NULL` antes de usar o ponteiro. Tentar acessar um ponteiro `NULL` causa erro de segmentação (segmentation fault).

Regra importante: Sempre verifique se `malloc` retornou `NULL` antes de usar o ponteiro. Isto evita erros graves no programa.

4.3 Função free

Toda memória alocada dinamicamente deve ser liberada quando não for mais necessária. A função `free` libera a memória apontada por um ponteiro:

```
free(ponteiro);
```

Não liberar a memória alocada causa vazamento de memória (*memory leak*), onde o programa consome cada vez mais memória até eventualmente esgotar os recursos disponíveis. Este é um problema grave em programas de longa duração.

Regra importante: Para cada `malloc`, deve haver um `free` correspondente. A memória não é liberada automaticamente.

4.4 Vetores Dinâmicos

Podemos usar `malloc` para criar vetores cujo tamanho é determinado durante a execução. Esta é uma das aplicações mais comuns de alocação dinâmica:

```
float *notas;
int n;
printf("Quantas notas? ");
scanf("%d", &n);

notas = malloc(n * sizeof(float));
if (notas == NULL) {
    printf("Erro de alocacao\n");
    exit(1);
}

for (int i = 0; i < n; i++) {
    printf("Nota %d: ", i+1);
    scanf("%f", &notas[i]);
```

```
}

// Usar o vetor...

free(notas);  // Liberar memoria
```

Note que usamos `notas[i]` para acessar os elementos do vetor, mesmo que `notas` seja um ponteiro. Em C, ponteiros e vetores são intercambiáveis em muitos contextos.

5 Aritmética de Ponteiros

Em C, podemos realizar operações aritméticas com ponteiros. Quando somamos um inteiro n a um ponteiro, o ponteiro avança n elementos (não n bytes). O compilador ajusta automaticamente o cálculo baseado no tamanho do tipo.

Considere um vetor `v` de inteiros. As expressões `v + i` e `&v[i]` são equivalentes e representam o endereço do elemento de índice `i`. Similarmente, `*(v + i)` é equivalente a `v[i]`.

```
int v[5] = {10, 20, 30, 40, 50};
int *p = v + 2;  // p aponta para v[2]
printf("%d\n", *p);       // imprime 30
printf("%d\n", *(p+1));   // imprime 40
printf("%d\n", *(v+3));   // imprime 40
printf("%d\n", v[3]);     // imprime 40
```

A Figura 3 ilustra a aritmética de ponteiros em um vetor.

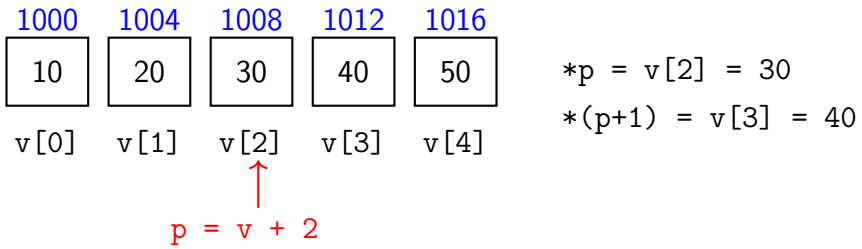

Figura 3: Aritmética de ponteiros em um vetor de inteiros.

Esta propriedade permite que ponteiros e vetores sejam usados de forma intercambiável. Quando declaramos um vetor estaticamente (como `int v[100]`), o nome do vetor é uma espécie de "ponteiro constante" cujo valor não pode ser alterado.

5.1 Percorrendo Vetores com Ponteiros

Podemos usar aritmética de ponteiros para percorrer vetores:

```
int v[5] = {10, 20, 30, 40, 50};
int *p;

// Percorrer usando indices
for (int i = 0; i < 5; i++)
    printf("%d␣", v[i]);

// Percorrer usando ponteiros
for (p = v; p < v + 5; p++)
    printf("%d␣", *p);
```

Ambos os métodos são equivalentes e produzem o mesmo resultado.

6 Regras Importantes

Ao trabalhar com ponteiros e alocação dinâmica, siga estas regras fundamentais:

1. Sempre inclua `stdlib.h` quando usar `malloc` e `free`.
2. Use `sizeof` para calcular o tamanho necessário. Nunca use valores fixos como 4 ou 8, pois o tamanho dos tipos pode variar entre sistemas.
3. Sempre verifique se `malloc` retornou `NULL` antes de usar o ponteiro. Isto evita erros de segmentação.
4. Libere toda memória alocada usando `free` quando não for mais necessária. Para cada `malloc`, deve haver um `free`.
5. Não use um ponteiro após liberar sua memória com `free`. Isto causa comportamento indefinido.
6. Não desreferencie ponteiros não inicializados. Um ponteiro deve sempre apontar para uma área de memória válida ou ser `NULL`.
7. Evite vazamentos de memória (*memory leaks*) liberando toda memória alocada antes de perder a referência ao ponteiro.

Resumo das operações com ponteiros:

- `&variavel` — obtém o endereço de `variavel`
- `*ponteiro` — acessa o valor apontado por `ponteiro`
- `malloc(n)` — aloca `n` bytes no heap
- `free(ponteiro)` — libera a memória alocada
- `sizeof(tipo)` — retorna o tamanho em bytes do `tipo`