

Ponteiros

Prof. John Lenon C. Gardenghi
29 de agosto de 2019

Sumário

1	Introdução	1
2	Aplicações de ponteiros	3
2.1	Passagem de parâmetros	3
2.2	Alocação dinâmica de memória	4
2.3	Ponteiros e vetores	7
2.4	Ponteiros para funções	9

1 Introdução

A memória RAM (= random access memory) de qualquer computador é uma sequência de bytes. A posição (0, 1, 2, 3, etc.) que um byte ocupa na sequência é o endereço do byte. Se e é o endereço de um byte então $e + 1$ é o endereço do byte seguinte.

As variáveis de um programa ocupam uma certa quantidade de bytes na memória, de acordo com seu tipo. Para saber o tamanho que um dado em C ocupa na memória, existe a função `sizeof`.

Exemplo 1. *Implemente o código a seguir, e observe os tamanhos que serão exibidos na tela. Será que todo computador possui os mesmos tamanhos?*

```
#include <stdio.h>
int main() {
    int array[20];
    char string[20];
    printf( "TAMANHOS:\n"
           "\t char.....: %ld\n"
           "\t short.....: %ld\n"
           "\t int.....: %ld\n"
           "\t long.....: %ld\n"
           "\t float.....: %ld\n"
           "\t double.....: %ld\n"
           "\t long double...: %ld\n"
           "\t vetor int....: %ld\n"
```

```

        "\t string.....: %ld\n",
        sizeof(char), sizeof(short), sizeof(int), sizeof(long),
        sizeof(float), sizeof(double), sizeof(long double),
        sizeof(array), sizeof(string) );
    return 0;
}

```

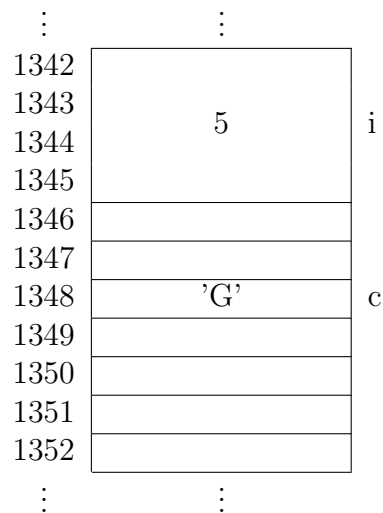
Cada posição da memória tem um endereço. Na maioria dos computadores, o endereço de uma variável é o endereço de memória que ocupa seu primeiro byte. O compilador é que controla do local de armazenamento destas variáveis em memória. Por exemplo, após as declarações

```

int i = 5;
char c = 'G';

```

as variáveis podem assumir os seguintes endereços:



O operador unário **&** é o **operador de endereço**. Retorna o endereço que uma variável ocupa na memória. Em nosso exemplo,

```
&i;  /* Contém 1342 */
```

Todavia, os endereços de memória costumam ser representados em notação *hexadecimal*.

Exemplo 2. *Implemente o código a seguir e observe os endereços de memória.*

```

#include <stdio.h>
int main() {
    int i = 5;
    printf( "&i (dec.): %ld\n", (long int) &i );
    printf( "&i (hexa.): %p\n", (void *) &i );
    return 0;
}

```

Ponteiros são variáveis que armazenam endereços de memória. Uma variável contém um valor, um ponteiro, por sua vez, contém o endereço de uma variável, que contém um valor.

Diz-se que uma variável referencia um valor *diretamente*, enquanto um ponteiro referencia um valor *indiretamente*. Por isso, chama-se o uso do ponteiro, muitas vezes, de **indireção**.

Como qualquer outra variável, um ponteiro deve ser declarado. O formato de declaração de ponteiros é:

```
tipo *ptr;
```

onde `tipo` são os tipos de variáveis em C. Um ponteiro pode ter o valor `NULL`, que é o valor inválido de ponteiros.

Se um ponteiro p armazena o endereço de uma variável i , dizemos que “ p aponta para i ”. Se um ponteiro p é diferente de `NULL`, então

`*p`

é o valor da variável apontada por p .

Exemplo 3. Rode o código a seguir e observe sua saída.

```
#include <stdio.h>
int main () {
    int i; int *p;
    i = 1234; p = &i;
    printf ("*p = %d\n", *p);
    printf (" p = %ld\n", (long int) p);
    printf (" p = %p\n", (void *) p);
    printf ("&p = %p\n", (void *) &p);
    return 0;
}
```

2 Aplicações de ponteiros

Há várias **aplicações de ponteiros**. São elas:

- Tipo de passagem de parâmetros (passagem por *valor* ou por *referência*) e múltiplo retorno de funções.
- Alocação dinâmica de memória.
- Definição conceitual de vetores.
- Referência indireta à funções.

2.1 Passagem de parâmetros

Há duas formas de passar argumentos para funções: por **valor** e por **referência**. No primeiro, cópia das variáveis originais são criadas nos parâmetros da função, e a função chamada lida apenas com cópias das variáveis originais da função chamadora. No segundo, ao invés de criar uma cópia, a própria variável original no processo chamador é passada ao procedimento chamado, por meio do seu endereço de memória.

Exemplo 4. Escreva uma função *troca* que receba dois valores inteiros *a* e *b* e troque os valores entre si.

```
void troca (int *p, int *q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}
int main() {
    int a = 2, b = 3;
    troca(&a, &b);
    printf( "a = %d, b = %d\n, a, b );
    return 0;
}
```

Note que, com isso, é possível também fazer com que uma função “retorne” vários valores, passando parâmetros por referência.

Exemplo 5. Faça uma função em C que receba um inteiro *n* e retorne n^2 e \sqrt{n} .

```
#include <math.h>
#include <stdio.h>
int calcula (double n, double *quadrado, double *raiz) {
    *quadrado = n*n;
    *raiz = sqrt (n);
}
int main () {
    double n, quadrado, raiz;
    scanf ("%lf", &n);
    calcula (n, &quadrado, &raiz);
    printf ("Quadrado = %lf\nRaiz = %lf\n", quadrado, raiz);
    return 0;
}
```

2.2 Alocação dinâmica de memória

Quando declaramos uma variável ou um vetor em C, essas entidades são alocadas pelo sistema operacional na memória na inicialização do executável. Esse tipo de alocação é chamado de *alocação estática*.

Por outro lado, suponha que desejamos alocar um vetor que não sabemos de antemão qual o tamanho. Para tanto, podemos usar *alocação dinâmica de memória*. Alocação dinâmica de memória é alocar um espaço de memória em tempo de execução do programa.

Há três funções de alocação dinâmica em C

1. `void *malloc (size_t tamanho);`

A função `malloc` (o nome é uma abreviatura de *memory allocation*) aloca espaço para um bloco de `tamanho` bytes consecutivos na memória e devolve um ponteiro para a primeira posição desse bloco. No seguinte fragmento de código, `malloc` aloca 1 byte:

```
char *ptr;
ptr = malloc (1);
scanf ("%c", ptr);
```

2. `void *calloc (size_t nmemb, size_t tamanho_tipo);`

A função `calloc` é bem similar à `malloc`, aloca memória para um vetor de `nmemb` elementos com `tamanho` bytes cada um. A diferença para o `malloc`, além dos parâmetros, é que todas as posições alocadas são zeradas. Neste exemplo, aloca-se um vetor de inteiros de 10 posições.

```
int *v;
v = calloc (10, sizeof (int));
```

3. `void *realloc (void *ptr, size_t tamanho);`

A função `realloc` realoca o vetor apontado por `ptr` para `tamanho` bytes. **Cuidado!** Há um custo implícito na função `realloc`, pois, ao realocar um vetor, se não houver memória suficiente disponível ao final, uma cópia deste vetor deve ser feita em outra região de memória. No código abaixo, realocamos o vetor `v` anterior para ter 20 posições.

```
v = realloc (v, 20*sizeof (int));
```

As funções de alocação dinâmica retornam um ponteiro de `void`. Se a alocação de memória não der certo por algum motivo, ela retorna `NULL`. Por isso, sempre que fizermos alocação dinâmica de memória, é necessário fazer a verificação.

```
double *ptr;
ptr = malloc (sizeof (double));
if (ptr == NULL) {
    scanf ("%lf", ptr);
}
```

Alocação de memória transfere para o programador a responsabilidade por aquele espaço que foi alocado. Por isso, é necessário **liberar** este espaço ao final do seu uso. Para isso, existe a função `free`. Para liberar o espaço alocado para `ptr` nos exemplos acima, basta fazer

```
free (ptr);
```

Como todas as vezes precisamos fazer a verificação se a alocação dinâmica deu certo, podemos fazer, em nossos programas, as chamadas *funções embalagens* (do inglês, *wrapper functions*). Para o `malloc`, por exemplo, podemos fazer

```
void *mallocc( size_t nbytes ) {
    void *ptr;
    ptr = malloc (nbytes);
    if (ptr == NULL) {
        printf ("Erro de alocação de memória.\n");
        exit (EXIT_FAILURE);
    }
}
```

```

    }
    return ptr;
}

```

e chamar `malloc` sempre que alocarmos um vetor. Evidentemente, o mesmo é possível para as funções `calloc` e `realloc`.

Observações

- Para fazer uso das funções de alocação dinâmica de memória, é necessário incluir a biblioteca `stdlib.h`.
- `NULL` é uma constante da `stdlib.h` que vale zero.
- A função `malloc` retorna um ponteiro para `void`. Por isso, é comum fazer um *cast* na hora da alocação dinâmica. Por exemplo,

```
int *ptr = ( int * ) malloc( 5*sizeof(int) );
```

Todavia, o *cast* é altamente não recomendável, pois o ponteiro de `void` é automaticamente convertido para outro tipo de ponteiro.

- `size_t` é um tipo de dados utilizado geralmente para representar tamanhos de objetos. É o valor de retorno do operador `sizeof` e muitas vezes é equivalente ao `unsigned int`.
- A função `exit` vem da biblioteca `stdlib` e interrompe a execução do programa e fecha todos os arquivos que o programa tenha porventura aberto. Se o argumento da função for 0, o sistema operacional é informado de que o programa terminou com sucesso; caso contrário, o sistema operacional é informado de que o programa terminou de maneira excepcional. As constantes `EXIT_SUCCESS` e `EXIT_FAILURE` valem 0 e 1, respectivamente, e estão declaradas na biblioteca `stdlib.h`. Chamar `exit (XXX)` de alguma função equivale a chamar `return XXX` do `main`.

Ponteiros também podem ser usados de forma múltipla, isto é, um ponteiro pode apontar para outro ponteiro. Por exemplo,

```

int x = 10;
int *p = &x;
int **q = &p;

```

Chamamos isto de **indireção múltipla**.

- Faz sentido usar o operador `*` várias vezes? E o `&`?

Um uso prático de indireção múltipla é a alocação dinâmica de memória para matrizes. O código a seguir aloca uma matriz de m linhas por n colunas. Tal matriz é ilustrada na Figura 1.

```

int **matriz;
int i, m, n;
scanf( "%d %d", &m, &n );
matriz = malloc( m * sizeof( int * ) );
for ( i = 0; i < m; i++ )
    matriz[i] = malloc( n * sizeof( int ) );

```

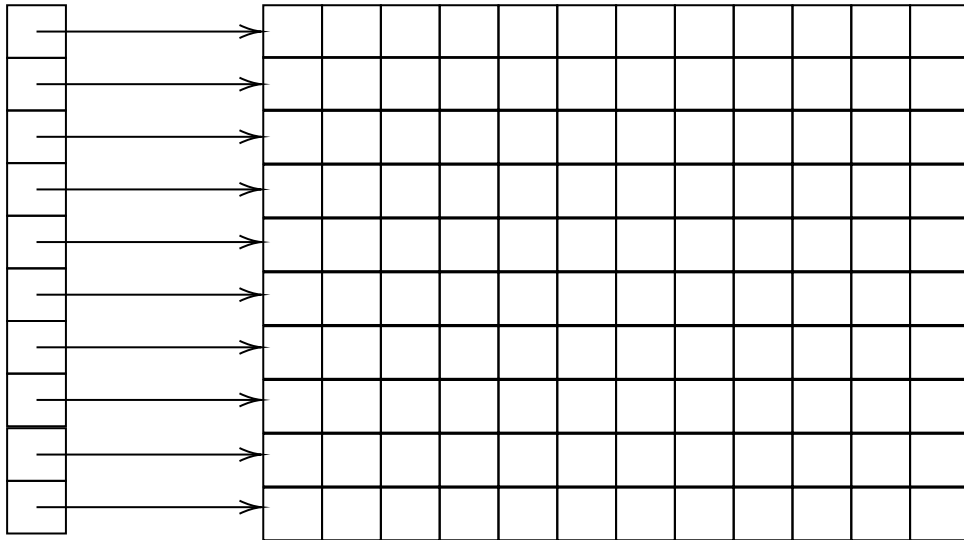


Figura 1: Exemplo de alocação dinâmica de uma matriz de $m = 10$ linhas e $n = 12$ colunas. É usado um ponteiro duplo. A primeira dimensão do ponteiro é um vetor de ponteiros (à esquerda), e cada posição aponta para a segunda dimensão do ponteiro, que são vetores de n elementos, representando cada linha da matriz alocada.

Para liberar esta matriz, basta fazer os laços na ordem inversa da alocação: primeiro desalocamos os vetores que representam as linhas, depois o vetor de ponteiros.

```
for (int i = 0; i < m; i++)
    free (matriz[i]);
free (matriz);
```

2.3 Ponteiros e vetores

O que é um vetor em C? Quando declaramos

```
int b[5];
```

significa que

- b é um vetor de 5 posições, cada uma das posições são armazenadas contiguamente na memória e
- b é um ponteiro que aponta para a primeira posição do vetor na memória.

Portanto, se fizermos

```
int *bPtr;
bPtr = b;
```

podemos operar tanto sobre b quanto $bPtr$.

Exemplo 6. *Implemente o programa a seguir e observe as saídas. O que podemos observar?*

```

int main() {
    int i, offset;
    int b[] = { 10, 20, 30, 40 };
    int *bPtr;
    bPtr = b;
    printf( "VETOR\n" );
    printf( "\n\tNotacao usando indices:\n" );
    for ( i = 0; i < 4; i++ )
        printf( "\t\tb[%d] = %d\n", i, b[i] );
    printf( "\n\tNotacao usando deslocamento\n" );
    for ( offset = 0; offset < 4; offset++ )
        printf( "\t\t*(b + %d) = %d\n", offset, *(b + offset) );
    printf( "\nPONTEIRO\n" );
    printf( "\n\tNotacao usando indices:\n" );
    for ( i = 0; i < 4; i++ )
        printf( "\t\tbPtr[%d] = %d\n", i, bPtr[i] );
    printf( "\n\tNotacao usando deslocamento\n" );
    for ( offset = 0; offset < 4; offset++ )
        printf( "\t\t*(bPtr + %d) = %d\n", offset, *(bPtr + offset) );
    return 0;
}

```

No geral,

- $v[i]$ é a notação vetorial para $*(v+i)$.
- $matriz[i][j]$ é a notação vetorial para $*(*(matriz+i)+j)$.

Neste contexto, ponteiros são operadores válidos em operações de comparação e aritméticas. Um ponteiro pode ser incrementado (++) ou decrementado (--) e um inteiro pode ser adicionado a (+ ou +=) ou subtraído de (- ou -=) um ponteiro. Isso é chamado **aritmética de ponteiros**.

A diferença para a aritmética normal para a aritmética de ponteiros é que a operação que se faz na aritmética de ponteiros é a seguinte:

```

TIPO *ptr;
int n;
ptr + n = ptr + n * sizeof (TIPO);

```

Para explicitar o que é feito na aritmética de ponteiros: suponha que um ponteiro

```
int *ptr;
```

possua o endereço 3000 armazenado. Se fizermos $ptr + 2$, o resultado usual seria 3002. Entretanto, quando um ponteiro é incrementado ou decrementado por um inteiro, o endereço de memória não é apenas alterado pelo inteiro, mas sim pelo inteiro vezes o tamanho do dados que o ponteiro referencia. Em nosso exemplo, $ptr + 2 = 3008$, já que um inteiro ocupa 4 bytes em memória. O mesmo vale para subtração, incremento e decremento unários. Por outro lado, se tivermos

```
int *ptr1, *ptr2;
```

e $ptr1$ possui o endereço 3000 e $ptr2$ possui o endereço 3008, então

```
int x = ptr2 - ptr1 = ( 3008 - 3000 ) / 4 = 2.
```


2.4 Ponteiros para funções

Da mesma forma que o nome de um vetor é, na verdade, um ponteiro para o endereço base na memória, o nome de uma função também é um ponteiro para o endereço de memória que contém o primeiro comando da função. Os demais comandos são armazenados contiguamente na memória. Por isso, ponteiros para funções podem também ser passados como argumentos de uma função.

Exemplo 7. *Suponha que queremos fazer uma função que encontre o maior ou menor elemento de um vetor.*

```
#include <stdio.h>
#include <stdlib.h>
int maior (int a, int b) {
    return a > b;
}

int menor (int a, int b) {
    return a < b;
}

void minmax (int n, int *v, int (*compara) (int a, int b)) {
    int i, elem;
    elem = v[0];
    for (i = 1; i < n; i++)
        if ((*compara) (v[i], elem))
            elem = v[i];
    printf( "elem = %d\n", elem );
}

int main () {
    int v[5] = { 5, 2, 4, 8, 6 };
    minmax (5, v, maior);
    minmax (5, v, menor);
    return 0;
}
```

Observação: É imprescindível o uso do parêntese no ponteiro da função, assim:

```
int (*compara) (int a, int b)
```

Se não houvesse parêntese, ou seja,

```
int *compara (int a, int b)
```

estariamos declarando uma função `compara` que retorna um ponteiro para inteiro.

Também é possível declarar um vetor de funções. Assim:

```
int (*f[2]) (int a, int b) = { maior, menor };
```

Deste modo, poderíamos fazer as chamadas

```
(*f[0]) (10, 2); /* Seria chamada a função maior */  
(*f[1]) (10, 2); /* Seria chamada a função menor */
```