

Endereços e ponteiros



Os conceitos de *endereço* e *ponteiro* são fundamentais em qualquer linguagem de programação. Em C, esses conceitos são explícitos; em algumas outras linguagens eles são ocultos (e representados pelo conceito mais abstrato de *referência*). Dominar o conceito de ponteiro exige algum esforço e uma boa dose de prática.

Endereços

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* (= *address*) do byte. (É como o endereço de uma casa em uma longa rua que tem casas de um lado só.) Se *e* é o endereço de um byte então *e+1* é o endereço do byte seguinte.

Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador. Uma variável do tipo [char](#) ocupa 1 byte. Uma variável do tipo [int](#) ocupa 4 bytes e um `double` ocupa 8 bytes em muitos computadores. O número exato de bytes de uma variável é dado pelo operador **sizeof**. A expressão `sizeof (char)`, por exemplo, vale 1 em todos os computadores e a expressão `sizeof (int)` vale 4 em muitos computadores.

Cada variável (em particular, cada registro e cada vetor) na memória tem um *endereço*. Na maioria dos computadores, o endereço de uma variável é o endereço do seu primeiro byte. Por exemplo, depois das declarações

```
char c;  
int i;  
struct {  
    int x, y;  
} ponto;  
int v[4];
```

as variáveis poderiam ter os seguintes endereços (o exemplo é fictício):

c	89421
i	89422
ponto	89426
v[0]	89434
v[1]	89438
v[2]	89442

O endereço de uma variável é dado pelo operador **&**. Assim, se **i** é uma variável então **&i** é o seu endereço. (Não confunda esse uso de "&" com o operador lógico *and*, que se escreve "&&" em C.) No exemplo acima, **&i** vale 89422 e **&v[3]** vale 89446.

Outro exemplo: o segundo argumento da função de biblioteca [scanf](#) é o endereço da variável que deve receber o valor lido do teclado:

```
int i;  
scanf ("%d", &i);
```

Exercícios 1

1. TAMANHOS. Compile e execute o seguinte programa:

```
int main (void) {
    typedef struct {
        int dia, mes, ano;
    } data;
    printf ("sizeof (data) = %d\n",
           sizeof (data));
    return EXIT_SUCCESS;
}
```

2. Compile e execute o seguinte programa. (O [cast](#) (long int) é necessário para que &i possa ser impresso com [especificação de formato](#) %ld. É mais comum imprimir endereços em [notação hexadecimal](#), usando a especificação de formato %p, que exige o cast (void *).)

```
int main (void) {
    int i = 1234;
    printf (" i = %d\n", i);
    printf ("&i = %ld\n", (long int) &i);
    printf ("&i = %p\n", (void *) &i);
    return EXIT_SUCCESS;
}
```

Ponteiros

Um *ponteiro* (= apontador = *pointer*) é um tipo especial de variável que armazena um endereço. Um ponteiro pode ter o valor

NULL

que é um endereço "inválido". A [macro](#) NULL está definida na [interface stdlib.h](#) e seu valor é 0 (zero) na maioria dos computadores.

Se um ponteiro p armazena o endereço de uma variável i, podemos dizer "p aponta para i" ou "p é o endereço de i". (Em termos um pouco mais abstratos, diz-se que p é uma *referência* à variável i.) Se um ponteiro p tem valor diferente de NULL então

*p

é o *valor* da variável apontada por p. (Não confunda esse operador "*" com o operador de multiplicação!) Por exemplo, se i é uma variável e p vale &i então dizer "*p" é o mesmo que dizer "i".

A seguinte figura dá um exemplo. Do lado esquerdo, um ponteiro p, armazenado no endereço 60001, contém o endereço de um inteiro. O lado direito dá uma representação esquemática da situação:



Tipos de ponteiros. Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para [registros](#), etc. O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro p para um inteiro, escreva

```
int *p;
```

(Há quem prefira escrever [int* p](#).) Para declarar um ponteiro p para um registro reg, diga

```
struct reg *p;
```

Um ponteiro r para um ponteiro que apontará um inteiro (como no caso de uma [matriz de inteiros](#)) é declarado assim:

```
int **r;
```

Exemplos. Suponha que a, b e c são variáveis inteiras e veja um jeito bobo de fazer "c = a+b":

```
int *p; // p é um ponteiro para um inteiro
int *q;
p = &a; // o valor de p é o endereço de a
q = &b; // q aponta para b
c = *p + *q;
```

Outro exemplo bobo:

```
int *p;
int **r; // ponteiro para ponteiro para inteiro
p = &a; // p aponta para a
r = &p; // r aponta para p e *r aponta para a
c = **r + b;
```

Exercícios 2

1. Compile e execute o seguinte programa. (Veja um dos exercícios [acima](#).)

```
int main (void) {
    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 EXIT_SUCCESS;
}
```

Aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos i e j. É claro que a função

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

não produz o efeito desejado, pois [recebe apenas os valores das variáveis](#) e não as variáveis propriamente ditas. A função recebe "cópias" das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas. Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

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

Para aplicar essa função às variáveis i e j basta dizer `troca (&i, &j);` ou então

```
int *p, *q;
p = &i; q = &j;
troca (p, q);
```

Exercícios 3

1. Verifique que a troca de valores de variáveis discutida acima poderia ser obtida por meio de uma macro do [pré-processador](#):

```
#define troca (X, Y) {int t = X; X = Y; Y = t;}

troca (i, j);
```

2. Por que o código abaixo está errado?

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

3. Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função `hm` que converta minutos em horas-e-minutos. A função recebe um inteiro `mnts` e os endereços de duas variáveis inteiras, digamos `h` e `m`, e atribui valores a essas variáveis de modo que `m` seja menor que 60 e que $60 \cdot h + m$ seja igual a `mnts`. Escreva também uma função `main` que use a função `hm`.
4. Escreva uma função `mm` que receba um vetor inteiro `v[0..n-1]` e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.

Aritmética de endereços

Os elementos de qualquer [vetor](#) são armazenados em bytes consecutivos na memória do computador. Se cada elemento do vetor ocupa k bytes, a diferença entre os endereços de dois elementos consecutivos é k . Mas o [compilador](#) ajusta os detalhes internos de modo a criar a ilusão de que a diferença entre os endereços de dois elementos consecutivos é 1, qualquer que seja o valor de k . Por exemplo, depois da declaração

```
int *v;
v = malloc (100 * sizeof (int));
```

o endereço do primeiro elemento do vetor é `v`, o endereço do segundo elemento é `v+1`, o endereço do terceiro elemento é `v+2`, etc. Se `i` é uma variável do tipo `int` então

$$v + i$$

é o endereço do $(i+1)$ -ésimo elemento do vetor. As expressões `v + i` e `&v[i]` têm exatamente o mesmo valor e portanto as atribuições

```
*(v+i) = 789;
v[i] = 789;
```

têm o mesmo efeito. Analogamente, qualquer dos dois fragmentos de código abaixo pode ser usado para preencher o vetor `v`:

```
for (i = 0; i < 100; ++i) scanf ("%d", &v[i]);
for (i = 0; i < 100; ++i) scanf ("%d", v + i);
```

Todas essas considerações também valem se o vetor for alocado estaticamente (ou seja, antes que o programa comece a ser executado) por uma declaração como

```
int v[100];
```

mas nesse caso `v` é uma espécie de "ponteiro constante", cujo valor não pode ser alterado.

Exercícios 4

1. Suponha que os elementos de um vetor `v` são do tipo `int` e cada `int` ocupa 4 bytes no seu computador. Se o endereço de `v[0]` é 55000, qual o valor da expressão `v + 3`?
2. Suponha que `i` é uma variável inteira e `v` um vetor de inteiros. Descreva, em português, a sequência de operações que deve ser executada para calcular o valor da expressão `&v[i + 9]`.

3. Suponha que v é um vetor. Descreva a diferença conceitual entre as expressões $v[3]$ e $v + 3$.

4. O que faz a seguinte função?

```
void imprime (char *v, int n) {  
    char *c;  
    for (c = v; c < v + n; c++)  
        printf ("%c", *c);  
}
```

5. O programa abaixo produziu a seguinte resposta, que achei surpreendente:

```
x: 111  
v[0]: 999
```

Os valores de x e $v[0]$ não deveriam ser iguais?

```
void func1 (int x) {  
    x = 9 * x;  
}  
  
void func2 (int v[]) {  
    v[0] = 9 * v[0];  
}  
  
int main (void) {  
    int x, v[2];  
    x = 111;  
    func1 (x); printf ("x: %d\n", x);  
    v[0] = 111;  
    func2 (v); printf ("v[0]: %d\n", v[0]);  
    return EXIT_SUCCESS;  
}
```

6. O seguinte fragmento de código pretende decidir se "abacate" vem antes ou depois de "uva" no dicionário. (Veja a seção [Cadeias constantes](#) do capítulo *Strings*.) O que está errado?

```
char *a, *b;  
a = "abacate"; b = "uva";  
if (a < b)  
    printf ("%s vem antes de %s\n", a, b);  
else  
    printf ("%s vem depois de %s\n", a, b);
```

Perguntas e respostas

- PERGUNTA: É verdade que devemos atribuir um valor a um ponteiro tão logo o ponteiro é declarado?
RESPOSTA: Há quem recomende inicializar todos os ponteiros imediatamente, ou seja, escrever `int *p = NULL;` em vez de simplesmente `int *p;`. Isso poderia ajudar a encontrar eventuais imperfeições no seu programa e poderia [proteger contra a ação de hackers](#). Este sítio não segue essa recomendação para não cansar o leitor com detalhes repetitivos. (Além disso, é deselegante atribuir um valor a uma variável sem que isso seja logicamente necessário...)

Veja [Pointers in C and C++](#) em [GeeksforGeeks](#).

Veja o verbete [Pointer \(computer programming\)](#) na Wikipedia.

Veja o verbete [C syntax](#) na Wikipedia.

[Aula em vídeo sobre ponteiros](#) no [Academic Earth](#) (usa C++, mas os conceitos são os mesmos de C).
[Aula em vídeo sobre aritmética de ponteiros](#) na The Open Academy.

Veja o sítio-ferramenta [cdecl: C gibberish → English](#), que decodifica expressões complexas em C, especialmente aquelas envolvendo ponteiros.

Atualizado em 2018-09-25
<https://www.ime.usp.br/~pf/algoritmos/>
Paulo Feofiloff
[DCC-IME-USP](#)

