

Apêndice D

Endereços e ponteiros

Os conceitos de endereço e ponteiro são fundamentais em qualquer linguagem de programação, embora sejam mais visíveis em C que em outras linguagens. O conceito de ponteiro é difícil; para dominá-lo, é preciso fazer um certo esforço.

D.1 Endereços

A memória de qualquer computador é uma sequência de bytes. Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente e o número de um byte é o seu **endereço**.

Cada objeto na memória do computador ocupa um certo número de bytes consecutivos. No meu computador, um `char` ocupa 1 byte, um `int` ocupa 4 bytes e um `double` ocupa 8 bytes. Cada objeto na memória do computador tem um **endereço**. (Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte.)

Em C, o endereço de um objeto é dado pelo operador `&`. (Não confunda este `&` com o operador lógico *and*, representado por `&&`.) Assim, se `i` é uma

<code>char c;</code>	<code>c</code>	<code>89421</code>
<code>int i;</code>	<code>i</code>	<code>89422</code>
<code>struct {</code>	<code>ponto</code>	<code>89426</code>
<code>int x, y;</code>	<code>v[0]</code>	<code>89434</code>
<code>} ponto;</code>	<code>v[1]</code>	<code>89438</code>
<code>int v[4];</code>	<code>v[2]</code>	<code>89442</code>

Figura D.1: O lado direito da figura dá os endereços das variáveis declaradas do lado esquerdo. Portanto, `&i` vale `89422` e `&v[3]` vale `89446`. (Os endereços não são muito realistas...)

variável então `&i` é o seu endereço.

O operador `&` aparece frequentemente nas invocações da função `scanf` (veja Seção H.1), por exemplo. Se `i` é uma variável inteira, podemos escrever `scanf ("%d", &i)`.



Figura D.2: Um ponteiro `p`, armazenado no endereço `60001`, contém o endereço de um inteiro. Neste exemplo, `p` vale `89422` enquanto `&p` vale `60001` e `*p` vale `-9999`. A parte direita da figura é uma representação esquemática muito útil da parte esquerda.

D.2 Ponteiros

Um **ponteiro** (ou **apontador**) é um tipo especial de variável destinado a armazenar endereços. Todo ponteiro pode ter o valor

NULL

que é um endereço “inválido”. A constante NULL está definida no arquivo-interface `stdlib` (veja Seção K.1) e seu valor é `0` 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`”. Se um ponteiro `p` tem valor diferente de NULL então

`*p`

é o valor do objeto apontado por `p`. (Não confunda esse uso de `*` com o operador de multiplicação!) Por exemplo, se `i` é uma variável e `p = &i` então escrever “`*p`” é o mesmo que escrever “`i`”.

Há vários tipos de ponteiros: ponteiros para caracteres, ponteiros para inteiros, ponteiros para registros, ponteiros para ponteiros para inteiros etc. Para declarar um ponteiro `p` para um inteiro, escreva¹

```
int *p;
```

¹ O leiaute das declarações de ponteiros é sabidamente desconfortável. Conceitualmente, um ponteiro-para-int é um novo tipo de dados, o que sugere que se escreva “`int* p`”. Do ponto de vista técnico, entretanto, “`*`” modifica a nova variável e não o “`int`”. Isto sugere que se escreva “`int *p`”. O compilador C aceita qualquer das formas. Também aceita “`int * p`”.

Para declarar um ponteiro `q` para um registro `cel`, escreva

```
struct cel *q;
```

```
int x, i = 888;
int *p;          /* p é um ponteiro para um inteiro */
p = &i;          /* p aponta para i */
x = *p + 999;
```

Figura D.3: Exemplo de ponteiro. O código mostra um jeito bobo de fazer “`x = i+999`”.

Exercícios

D.2.1 Se `i` é uma variável do tipo `int`, que sentido fazem as expressões `*&i` e `&&i`?

D.2.2 Leia o verbete *Pointer* na Wikipedia [21].

D.3 Uma aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras `i` e `j`. A função

```
void troca (int i, int j) { /* errado! */
    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. 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 a função às variáveis `i` e `j` basta dizer

```
troca (&i, &j);
```


Exercícios

D.3.1 Por que o código abaixo está errado?

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

D.3.2 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 t e os endereços de duas variáveis inteiras, digamos h e m , e atribui valores não negativos a essas variáveis de modo que tenhamos $m < 60$ e $60h + m = t$. Escreva também uma função `main` que use a função `hm`.

D.3.3 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 nestas 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`.

D.4 Vetores e endereços

Os elementos de qualquer vetor são alocados consecutivamente na memória do computador. Se cada elemento ocupa l bytes, a diferença entre os endereços de dois elementos consecutivos será l . Para livrar o programador da preocupação com o valor de l , o compilador C cria a ilusão de que l vale 1 qualquer que seja o tipo dos elementos do vetor. Por exemplo, depois dos comandos

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

o ponteiro `v` aponta o primeiro elemento de um vetor de 100 inteiros. O endereço do segundo elemento do vetor é `v+1` e o endereço do terceiro é `v+2`. Se i é uma variável do tipo `int` então `v+i` é o endereço do $(i+1)$ -ésimo elemento do vetor.

A propósito, as expressões `v+i` e `&v[i]` têm exatamente o mesmo valor e portanto as atribuições

$$*(v+i) = 987 \quad \text{e} \quad v[i] = 987$$

têm o mesmo efeito. Todas estas considerações também valem se o vetor tiver sido alocado estaticamente, por uma declaração como

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

(Mas nesse caso `v` é uma espécie de “ponteiro constante”, cujo valor não pode ser alterado.)

Exe

D.4.1
dor. S

D.4.2

D.4.3

Descre
o valor

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

Figura D.4: Qualquer dos dois fragmentos de código pode ser usado para “carregar” o vetor `v[0..99]`.

Exercícios

D.4.1 Seja `v` um vetor de `ints`. Suponha que cada `int` ocupa 8 bytes no seu computador. Se o endereço de `v[0]` é `55000`, qual o valor da expressão `v + 3`?

D.4.2 Se `v` é um vetor, qual a diferença conceitual entre as expressões `v[3]` e `v+3`.

D.4.3 Suponha que o vetor `v` e a variável `k` foram declarados assim:

```
int v[100], k;
```

Descreva, em português, a sequência de operações que deve ser executada para calcular o valor da expressão `&v[k+9]`.

A expressão `p->mês` é uma abreviatura muito útil da expressão `(*p).mês`:

```
p->mês = 12;      /* mesmo efeito que (*p).mês = 12 */  
p->ano = 2008;
```

Exercícios

E.2.1 Defina um registro **empregado** para armazenar os dados (nome, data de nascimento, número do documento de identidade, data de admissão, salário) de um empregado de sua empresa. Defina um vetor de **empregados** para armazenar todos os empregados de sua empresa.

E.2.2 Um *racional* é qualquer número da forma p/q , sendo p inteiro e q inteiro não nulo. É conveniente representar cada racional por um registro:

```
struct racional { int p, q; };
```

Vamos convencionar que o campo q de todo racional é estritamente positivo e que o máximo divisor comum dos campos p e q é 1. Escreva uma função que receba inteiros a e b e devolva o racional que representa a/b . Escreva uma função que receba um racional x e devolva o racional $-x$. Escreva funções que recebam racionais x e y e devolvam (1) o racional que representa a soma de x e y , (2) o racional que representa o produto de x por y e (3) o racional que representa o quociente de x por y .