AULA 9

INTRODUÇÃO AOS PONTEIROS

Ponteiros ou apontadores, do inglês *pointers*, são certamente uma das características mais destacáveis da linguagem de programação C. Os ponteiros agregam poder e flexibilidade à linguagem de maneira a diferenciá-la de outras linguagens de programação de alto nível, permitindo a representação de estruturas de dados complexas, a modificação de valores passados como argumentos a funções, alocação dinâmica de espaços na memória, entre outros destaques. Nesta aula iniciaremos o contato com esses elementos.

Esta aula é baseada especialmente nas referências [2, 7].

9.1 Variáveis ponteiros

Para bem compreender os ponteiros, precisamos inicialmente compreender a idéia de indireção. Conceitualmente, um ponteiro permite acesso indireto a um valor armazenado em algum ponto da memória. Esse acesso é realizado justamente porque um **ponteiro** é uma variável que armazena um valor especial, que é um endereço de memória, e por isso nos permite acessar indiretamente o valor armazenado nesse endereço.

A memória de um computador pode ser vista como constituída de muitas posições (ou compartimentos ou células), dispostas continuamente, cada qual podendo armazenar um valor, na base binária. Ou seja, a memória nada mais é do que um grande vetor que pode armazenar valores na base binária e que, por sua vez, esses valores podem ser interpretados como valores de diversos tipos. Os índices desse grande vetor, numerados seqüencialmente a partir de 0 (zero), são chamados de **endereços de memória**.

Quando escrevemos um programa em uma linguagem de programação de alto nível, o nome ou identificador de uma variável é associado diretamente a um índice desse vetor, isto é, a um endereço da memória. A tradução do nome da variável para um endereço de memória, e vice-versa, é feita de forma automática e transparente pelo compilador da linguagem de programação. Essa é uma característica marcante de uma linguagem de programação de alto nível, que se diferencia das linguagens de programação de baixo nível, já que não há necessidade de um(a) programador(a) preocupar-se com os endereços de memória quando armazena/recupera dados na memória.

Em geral, a memória de um computador é dividida em bytes, com cada byte sendo capaz de armazenar 8 bits de informação. Cada byte tem um único endereço que o distingue de outros bytes da memória. Se existem n bytes na memória, podemos pensar nos endereços como números de intervalo de 0 a $n - 1$, como mostra a figura 9.1.

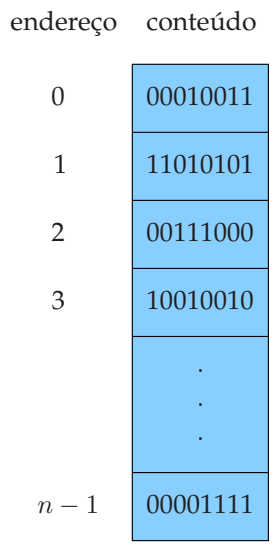

Figura 9.1: Uma ilustração da memória e de seus endereços.

Um programa executável é constituído por trechos de código e dados, ou seja, por instruções de máquina que correspondem às sentenças no programa original na linguagem C e também de variáveis. Cada variável do programa ocupa um ou mais bytes na memória. O endereço do primeiro byte de uma variável é dito ser o endereço da variável. Na figura 9.2, a variável i ocupa os bytes dos endereços 2000 e 2001. Logo, o endereço da variável i é 2000.

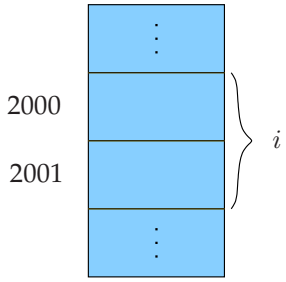

Figura 9.2: Endereço da variável i.

Os ponteiros têm um papel importante em toda essa história. Apesar de os endereços serem representados por números, como ilustrado nas figuras 9.1 e 9.2, o intervalo de valores que esses objetos podem assumir é diferente do intervalo que os números inteiros, por exemplo, podem assumir. Isso significa, entre outras coisas, que não podemos armazenar endereços em variáveis do tipo inteiro. Endereços são armazenados em variáveis especiais, chamadas de variáveis ponteiros.

Quando armazenamos o endereço de uma variável i em uma variável ponteiro p, dizemos que p **aponta para** i. Em outras palavras, um ponteiro nada mais é que um endereço e uma variável ponteiro é uma variável que pode armazenar endereços.

Ao invés de mostrar endereços como números, usaremos uma notação simplificada de tal forma que, para indicar que uma variável ponteiro p armazena o endereço de uma variável i, mostraremos o conteúdo de p – um endereço – como uma flecha orientada na direção de i, como mostra a figura 9.3.

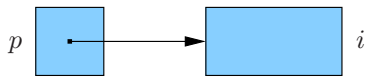

Figura 9.3: Representação de uma variável ponteiro.

Uma variável ponteiro pode ser declarada da mesma maneira que uma variável qualquer de qualquer tipo, como sempre temos feito, mas com um asterisco precedendo seu identificador. Por exemplo,

```
int *p;
```

Essa declaração indica que p é uma variável ponteiro capaz de apontar para objetos do tipo `int`. A linguagem C obriga que toda variável ponteiro aponte apenas para objetos de um tipo particular, chamado de **tipo referenciado**.

Diante do exposto até este ponto, daqui para frente não mais distinguiremos os termos 'ponteiro' e 'variável ponteiro', ficando então subentendido o seu valor (conteúdo) e a própria variável.

9.2 Operadores de endereçamento e de indireção

A linguagem C possui dois operadores para uso específico com ponteiros. Para obter o endereço de uma variável, usamos o **operador de endereçamento (ou de endereço) &**. Se v é uma variável, então `&`v é seu endereço na memória. Para ter acesso ao objeto que um ponteiro aponta, temos de usar o **operador de indireção** `*`. Se p é um ponteiro, então `*`p representa o objeto para o qual p aponta no momento.

A declaração de uma variável ponteiro reserva um espaço na memória para um ponteiro mas não a faz apontar para um objeto. Assim, é crucial inicializar um ponteiro antes de usá-lo. Uma forma de inicializar um ponteiro é atribuir-lhe o endereço de alguma variável usando o operador `&`:

```
int i, *p;

p = &i;
```

Atribuir o endereço da variável i para a variável p faz com que p aponte para i, como ilustra a figura 9.4.

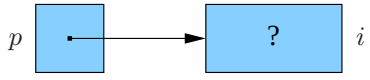

Figura 9.4: Variável ponteiro p contendo o endereço da variável i.

É possível inicializar uma variável ponteiro no momento de sua declaração, como abaixo:

```
int i, *p = &i;
```

Uma vez que uma variável ponteiro aponta para um objeto, podemos usar o operador de indireção `*` para acessar o valor armazenado no objeto. Se p aponta para i, por exemplo, podemos imprimir o valor de i de forma indireta, como segue:

```
printf("%d\n", *p);
```

Observe que a função `printf` mostrará o valor de i e não o seu endereço. Observe também que aplicar o operador `&` a uma variável produz um ponteiro para a variável e aplicar o operador `*` para um ponteiro retoma o valor original da variável:

```
j = *&i;
```

Na verdade, a atribuição acima é idêntica à seguinte:

```
j = i;
```

Enquanto dizemos que p **aponta para** i, dizemos também que `*`p é um **apelido** para i. Ademais, não apenas `*`p tem o mesmo valor que i, mas alterar o valor de `*`p altera também o valor de i.

Uma observação importante que auxilia a escrever e ler programas com variáveis ponteiros é sempre "traduzir" os operadores unários de endereço `&` e de indireção `*` para *endereço da variável* e *conteúdo da variável apontada por*, respectivamente. Sempre que usamos um desses operadores no sentido de estabelecer indireção e apontar para valores, é importante traduzi-los desta forma para fins de clareza.

Observe que no programa 9.1, após a declaração das variáveis c e p, temos a inicialização do ponteiro p, que recebe o endereço da variável c, sendo essas duas variáveis do mesmo tipo `char`. Também ocorre a inicialização da variável c. É importante sempre destacar que o valor, ou conteúdo, de um ponteiro na linguagem C não tem significado até que contenha, ou aponte, para algum endereço válido.

A primeira chamada da função `printf` no programa 9.1 mostra o endereço onde a variável c se localiza na memória e seu conteúdo, inicializado com o caractere `'a'` na linha anterior. Note que um endereço pode ser impresso pela função `printf` usando o especificador de tipo `%p`. Em seguida, um outra chamada à função `printf` é realizada, mostrando o endereço onde a varíavel p se localiza na memória, o conteúdo da variável p e o conteúdo da variável apontada por p. Como a variável p aponta para a variável c, o valor apresentado na saída é também aquele armazenado na variável c, isto é, o caractere `'a'`. Na segunda vez que ocorrem as mesmas chamadas às funções `printf`, elas são precedidas pela alteração do conteúdo da variável c e, como a variável p mantém-se apontando para a variável c, o caractere `'/'` seja

Programa 9.1: Um exemplo do uso de ponteiros.

```
#include <stdio.h>

int main(void)
{
   char c, *p;

   p = &c;
   c = 'a';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   c = '/';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   *p = 'Z';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   return 0;
}
```

apresentado na saída quando solicitamos a impressão de `*p`. É importante notar que, a menos que o conteúdo da variável p seja modificado, a expressão `*p` sempre acessa o conteúdo da variável c. Por fim, o último conjunto de chamadas à função `printf` é precedido de uma atribuição que modifica o conteúdo da variável apontada por p e conseqüentemente da variável c. Ou seja, a atribuição a seguir:

```
*p = 'Z';
```

faz também com que a variável c receba o caractere `'Z'`.

A execução do programa 9.1 em um computador com processador de 64 bits tem o seguinte resultado na saída:

```
&c = 0x7fffffffc76f    c = a
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = a

&c = 0x7fffffffc76f    c = /
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = /

&c = 0x7fffffffc76f    c = Z
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = Z
```

Observe que um endereço de memória é impresso na base hexadecimal com o especificador de tipo `%p`. A figura 9.5 ilustra o início da execução do programa 9.1.

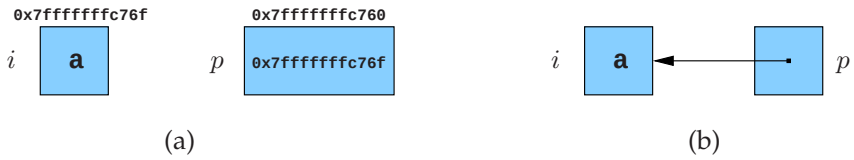

Figura 9.5: Ilustração das variáveis do programa 9.1 após as inicializações das variáveis. (a) Representação com endereços. (b) Representação esquemática.

A linguagem C permite ainda que o operador de atribuição copie ponteiros, supondo que possuam o mesmo tipo. Suponha que a seguinte declaração tenha sido feita:

```
int i, j, *p, *q;
```

Então, a sentença:

```
p = &i;
```

é um exemplo de atribuição de um ponteiro, onde o endereço de i é copiado em p. Um outro exemplo de atribuição de ponteiro é dado a seguir:

```
q = p;
```

Essa sentença copia o conteúdo de p, o endereço de i, para q, fazendo com que q aponte para o mesmo lugar que p aponta, como podemos visualizar na figura 9.6.

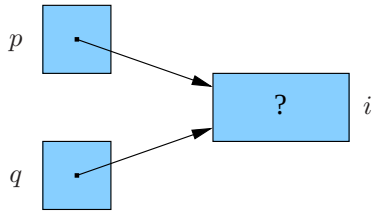

Figura 9.6: Dois ponteiros para um mesmo endereço.

Ambos os ponteiros p e q apontam para i e, assim, podemos modificar o conteúdo de i indiretamente através da atribuição de valores para `*`p e `*`q.

9.3 Ponteiros em expressões

Ponteiros podem ser usados em expressões aritméticas de mesmo tipo que seus tipos referenciados. Por exemplo, um ponteiro para uma variável do tipo inteiro pode ser usado em uma

expressão aritmética envolvendo números do tipo inteiro, supondo o uso correto do operador de indireção `*`.

É importante observar também que os operadores `&` e `*`, por serem operadores unários, têm precedência sobre os operadores binários das expressões aritméticas em que se envolvem. Por exemplo, em uma expressão aritmética envolvendo números do tipo inteiro, os operadores binários `+`, `-`, `*` e `/` têm menor prioridade que o operador unário de indireção `*`.

Vejamos a seguir, como um exemplo simples, o programa 9.2 que usa um ponteiro como operando em uma expressão aritmética.

Programa 9.2: Outro exemplo do uso de ponteiros.

```
#include <stdio.h>

int main(void)
{
   int i, j, *ptr1, *ptr2;

   ptr1 = &i;
   i = 5;
   j = 2 * *ptr1 + 3;
   ptr2 = ptr1;

   printf("i = %d, &i = %p\n\n", i, &i);
   printf("j = %d, &j = %p\n\n", j, &j);
   printf("&ptr1 = %p, ptr1 = %p, *ptr1 = %d\n", &ptr1, ptr1, *ptr1);
   printf("&ptr2 = %p, ptr2 = %p, *ptr2 = %d\n\n", &ptr2, ptr2, *ptr2);

   return 0;
}
```

A execução do programa 9.2 tem a seguinte saída:

```
i =  5, &i = 0x7fffffffc55c
j = 13, &j = 0x7fffffffc558

&ptr1 = 0x7fffffffc550, ptr1 = 0x7fffffffc55c, *ptr1 = 5
&ptr2 = 0x7fffffffc548, ptr2 = 0x7fffffffc55c, *ptr2 = 5
```

Exercícios

9.1 Se i é uma variável e p é uma variável ponteiro que aponta para i, quais das seguintes expressões são apelidos para i?

(a) `*`p

(b) `&`p

(c) `*&`p

(d) `&*`p

(e) `*`i

(f) `&`i

(g) `*&`i

(h) `&*`i

9.2 Se i é uma variável do tipo `int` e p e q são ponteiros para `int`, quais das seguintes atribuições são corretas?

(a) p `= ` i`;`

(b) `*`p `= &`i`;`

(c) `&`p `= ` q`;`

(d) p `= &`q`;`

(e) p `= *&`q`;`

(f) p `= ` q`;`

(g) p `= *`q`;`

(h) `*`p `= ` q`;`

(i) `*`p `= *`q`;`

9.3 Entenda o que o programa 9.3 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 9.3: Programa do exercício 9.3.

```
#include <stdio.h>

int main(void)
{
   int a, b, *ptr1, *ptr2;

   ptr1 = &a;
   ptr2 = &b;
   a = 1;
   (*ptr1)++;
   b = a + *ptr1;
   *ptr2 = *ptr1 * *ptr2;

   printf("a=%d, b=%d, *ptr1=%d, *ptr2=%d\n", a, b, *ptr1, *ptr2);

   return 0;
}
```

9.4 Entenda o que o programa 9.4 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

9.5 Entenda o que o programa 9.5 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 9.4: Programa do exercício 9.4.

```
#include <stdio.h>

int main(void)
{
   int a, b, c, *ptr;

   a = 3;
   b = 7;
   printf("a=%d, b=%d\n", a, b);

   ptr = &a;
   c = *ptr;
   ptr = &b;
   a = *ptr;
   ptr = &c;
   b = *ptr;
   printf("a=%d, b=%d\n", a, b);

   return 0;
}
```

Programa 9.5: Programa do exercício 9.5.

```
#include <stdio.h>

int main(void)
{
   int i, j, *p, *q;

   p = &i;
   q = p;
   *p = 1;
   printf("i=%d, *p=%d, *q=%d\n", i, *p, *q);

   q = &j;
   i = 6;
   *q = *p;
   printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);

   return 0;
}
```