Next Up Previous

Next: 29 Strings e Ponteiros **Up:** 2 Tópicos Avançados **Previous:** 27 Estruturas

Subsecções

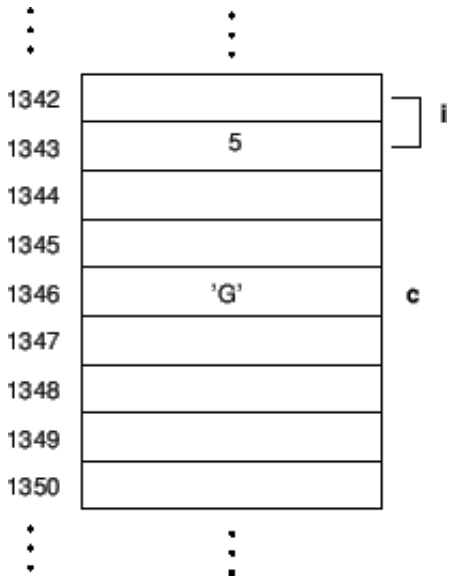
---

28 Ponteiros

Em linguagem C a cada variável está associado: (i) um nome; (ii) um tipo; (iii) um valor; e (iv) um endereço. Considere as seguintes definições de variáveis.

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

Na memória, eles podem estar armazenados da forma abaixo:



A variável inteira `i` está armazenada no endereço `1342`. Ela usa dois bytes de memória (quando um objeto usa mais de um byte, seu endereço é onde ele começa - neste caso, `1342` e não `1343`). A variável do tipo `char c` está armazenada no endereço `1346` e usa um byte de memória. O compilador é que controla do local de armazenamento destas variáveis em memória.

28.1 O operador de endereço (&)

Nós podemos usar o operador de endereço para determinar o endereço de uma objeto na memória. Este operador só pode ser usado com *lvalues* (objetos que podem estar no lado esquerdo de uma atribuição, como no caso de variáveis) porque *lvalues* tem um endereço alocado na memória.

Por exemplo, no exemplo acima, poderíamos usar o operador de endreço como nas expressões abaixo:

`&i` tem valor `1342`

`&c` tem valor `1346`

28.2 Tipo ponteiro

Em C, uma variável que contém um endereço de memória é uma variável do tipo **ponteiro**. Um valor, que é um endereço (como `&a`) é um valor de ponteiro. Quando um ponteiro (a variável) contém um determinado endereço, dizemos que ele *aponta* para o endereço de memória. Além disso, se o valor deste ponteiro é o endereço de uma outra variável qualquer, dizemos que tal ponteiro *aponta* para esta outra variável.

Há um tipo distinto de ponteiro para cada tipo básico C (como `int`, `char` e `float`). É verdade que todos os endereços tem o mesmo tamanho[4], mas nós também precisamos saber algo sobre o que é armazenado no endereço de memória apontado (quantos bytes ocupa e como os bytes devem ser interpretados).

Assim, a declaração de um tipo ponteiro em C é feita da seguinte forma:

```
tipo *nome_var;
```

Esta declaração indica que está sendo definido um ponteiro para *tipo* chamado *nome_var*. Por exemplo, um tipo ponteiro usado para *apontar* para inteiros é chamado *ponteiro para int* e isso é denotado por um `int *`. Variáveis do tipo ponteiro para `int` são usadas para armazenar endereços de memória que contem valores do tipo `int`.

Dadas as definições de `i` e `c` acima, nós podemos definir duas novas variáveis `pi` e `pc`, ambos do tipo ponteiro.

```
int *pi;
char *pc;
```

Nesta definição as variáveis não foram inicializadas com nenhum valor. Podemos inicializá-las com:

```
pi = &i;
pc = &c;
```

Depois destas atribuições, o valor de `pi` seria `1342`, e o valor de `pc` seria `1346`.

Note que nesta definição da variável `int *pi`, `pi` é o nome da variável e `int *` é o tipo de `pi` (ponteiro para `int`).

28.3 O operador de dereferência: *

Quando um ponteiro aponta para um endereço de memória, a operação para acessar o conteúdo do endereço apontado é chamado de **dereferência**. O operador unário `*` é usado para fazer a dereferência. Note que este uso do símbolo `*` não tem relação com o símbolo de multiplicação. Usando os exemplos

anteriores, `*pi` é o objeto apontado por `pi` (no caso, o valor de um inteiro).

`*pi` tem valor `5`

`*pc` tem valor `'G'`

Como um ponteiro dereferenciado (tais como `*pi` ou `*pc`) refere-se a um objeto na memória, ele pode ser usado não só como valor, mas também como um *lvalue*. Isto significa que um ponteiro dereferenciado pode ser usado no lado esquerdo de uma atribuição. Veja alguns exemplos:

```
printf("Valor= %d, Char = %c\n", *pi, *pc);
*pi = *pi + 5;
*pc = 'H';
```

`*pi` no lado esquerdo do `=` refere-se ao endereço de memória para o qual `pi` aponta. `*pi` no lado direito do `=` refere-se ao valor armazenado no endereço apontado por `pi`. A sentença `*pi = *pi + 5;` faz com que o valor armazenado no endereço apontado por `pi` seja incrementado de `5`. Note que o valor de `*pi` muda, não o valor de `pi`.

Neste exemplo, os valores das variáveis `i` e `c` poderiam ter sido alterados sem a utilização de ponteiros da seguinte forma:

```
printf("Valor = %d, Char = %c\n", i, c);
i = i + 5;
c = 'H';
```

Os exemplos acima ilustram como uma variável pode ser acessada diretamente (através do seu nome) ou indiretamente (através de um ponteiro apontando para o endereço da variável).

28.4 Atribuições envolvendo ponteiros

Um ponteiro pode ter atribuído a si um valor que seja o endereço de memória onde está armazenado um valor do mesmo tipo do ponteiro. Isto ocorre quando se usa o operador de endereço visto acima, ou quando se usa o valor de um outro ponteiro que aponte para um objeto do mesmo tipo do primeiro ponteiro. Observe-se o exemplo abaixo:

```
int *p1, *p2, x;
float *p3;

p1 = &x;       /* Correto */
p2 = p1;       /* Correto */
p3 = p1;       /* Incorreto. Compilador acusa "Warning". */
```

No exemplo acima, a linguagem C admite a atribuição de um ponteiro para outro de outro tipo (*p3 = p1;*), mas a compilação acusa uma mensagem de aviso. Posteriormente serão vistas situações em que a atribuição de ponteiros de tipos diferentes devem ocorrer e como devem ser manipuladas em C.

28.5 Aritmética de ponteiros

Apenas as operações de adição e subtração (e operadores C associados) são permitidos com ponteiros. Assim, é possível adicionar ou subtrair valores inteiros de ponteiros.

Operações de soma, subtração e comparação entre ponteiros também são válidas, desde que os ponteiros envolvidos apontem para o mesmo tipo de dados. Ainda assim, o resultado somente terá algum sentido prático se os ponteiros apontarem também para o mesmo objeto.

Alguns exemplos:

```
int num[20], *pnum, diff;
char str[30], *pstr, *pn, char nome[20];

pn = nome;
pstr = str;
pnum = num;

pnum += 3;            /* pnum = &num[3] */
*pnum = 10;           /* equivale a num[3] = 10 */

pstr++;               /* pstr = &str[1] */

diff = pstr - pnum;   /* INCORRETO. Os ponteiros apontam para
                       * tipos diferentes
                       */
diff = pstr - pn;     /* CORRETO, mas o valor não tem
                       * necessáriamente o sentido de "numero
                       * de bytes entre pn e pstr".
                       */
pn = str;
pstr = &str[30];

diff = pstr - pn;     /* CONCEITUALMENTE CORRETO. diff == 30 */
```

Um último ponto a respeito de operações sobre ponteiros: Adicionar um ponteiro a outro não produz nenhum resultado prático ou válido.

28.6 Ponteiros e *Arrays*

Em C, o nome de uma variável que foi declarada como *array* representa um ponteiro que aponta para o início do espaço de armazenamento do *array*, isto é, o endereço de memória do primeiro byte associado ao primeiro elemento do array:

```
char nome[20],
     *pstr;
int val[10],
    *ptr;


pstr = nome;       /* Equivalente a pstr = &nome[0] */
ptr = val;         /* Equivalente a ptr = &val[0] */

pstr = nome + 4;   /* Equivalente a pstr = &nome[4] */
ptr = val + 5;     /* Equivalente a ptr = &val[5] */

pstr = nome++;     /* ATENCAO: INCORRETO !!! */
                   /* "nome" NÃO É UM PONTEIRO */
```

Se um ponteiro aponta para um *array*, pode-se usar indistintamente as formas abaixo para acessar os elementos do *array*:

```
int val[10],
    x,
    *ptr;

ptr = val;         /* Equivalente a ptr = &val[0] */

*(ptr + 3) = 7;   /* val[3] = 7 */
ptr[3] = 10;      /* val[3] = 10 */
```

```
                        /* Equivalente a *(ptr + 3) = 10 */

    ptr += 4;
    ptr[3] = 20;      /* ATENCAO:  val[7] = 20 */
```

28.7 Ponteiros e Estruturas

Como em qualquer outro tipo, ponteiros para estruturas podem ser definidos. Considere o exemplo abaixo:

```
/* declara uma estrutura */
struct facil {
   int num;
   char ch;
};

main()
{
    /* definioes de variaveis */
    struct facil fac,      /* uma variavel do tipo "struct facil" */
                *pfac;    /* um ponteiro para "struct facil" */

    pfac = &fac;

    (*pfac).num = 32; /* o membro "num" da "struct facil" apontada por "pfac" */
    (*pfac).ch = 'A'; /* o membro "char" da "struct facil" apontada por "pfac" */
  }
```

Como se espera, quando se usa um ponteiro para um tipo `struct`, o ponteiro deve ter assinalado a si um valor ANTES de ser dereferenciado. A ordem pela qual um membro pode ser acessado através do ponteiro, the pointer é: primeiro o ponteiro é dereferenciado, e então o operador de membro de estrutura e o nome do membro são usados para acessar um membro em particular da estrutura apontada pelo ponteiro. Uma vez que o operador `.` tem precedência mais alta que o operador `*` (veja Tabela 6), os parenteses são necessários.

28.7.1 Acesso a membros de estrutura via ponteiro: O operador ->

Uma notação do tipo `(*pfac).ch` é confusa, de forma que a linguagem C define um operador adicional (`->`) para acessar membros de estruturas através de ponteiros. O operador `->` é formalmente usado como o operador `.`, exceto que ao invés do nome da variável de estrutura, um ponteiro para o tipo `struct` é usado à esquerda do operador `->`. No exemplo acima, as duas últimas linhas de código podem portanto ser reescritas como:

```
    pfac->num = 32;   /* o mesmo que (*pfac).num = 32;  */
    pfac->ch = 'A';   /* o mesmo que (*pfac).ch = 'A';  */
```

Basicamente, use o operador `.` se você tem uma variável de tipo `struct`, e o operador `->` caso você tenha um ponteiro para um tipo `struct`.

28.8 Ponteiros como argumentos de funções

Nos exemplos acima, pode parecer que ponteiros não são úteis, já que tudo que fizemos pode ser feito sem usar ponteiros. Agora, considere o exemplo da função `troca()` abaixo, que deve trocar os valores entre seus argumentos:

```
    #include <stdio.h>
```

```
void troca(int, int);

void troca(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);

    /* Troca a e b */
    troca(a, b);

    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

Quando `a` e `b` são passados como argumentos para `troca()`, na verdade, somente seus valores são passados. A função não pode alterar os valores de `a` e `b` porque ela não conhece os endereços de `a` e `b`. Mas se ponteiros para `a` e `b` forem passados como argumentos ao invés de `a` e `b`, a função `troca()` seria capaz de alterar seus valores; ela saberia então em que endereço de memória escrever. Na verdade, a função não sabe que os endereços de memória são associados com `a` e `b`, mas ela pode modificar o conteúdo destes endereços. Portanto, passando um ponteiro para uma variável (ao invés do valor da variável), habilitamos a função a alterar o conteúdo destas variáveis na função chamadora.

Uma vez que endereços de variáveis são do tipo ponteiro, a lista de parâmetros formais da função deve refletir isso. A definição da função `troca()` deveria ser alterada, e a lista de parâmetros formais deve ter argumentos não do tipo `int`, mas ponteiros para `int`, ou seja, `int *`. Quando chamamos a função `troca()`, nós não passamos como parâmetros reais `a` e `b`, que são do tipo `int`, mas `&a` e `&b`, que são do tipo `int *`. Dentro da função `troca()` deverá haver mudanças também. Uma vez que agora os parâmetros formais são ponteiros, o operador de dereferência, `*`, deve ser usado para acessar os objetos. Assim, a função `troca()` é capaz de alterar os valores de `a` e `b` ``remotamente".

O programa abaixo é a versão correta do problema enunciado para a função `troca()`:

```
#include <stdio.h>

void troca(int *, int *);

/* function troca(px, py)
 *  acao:        troca os valores inteiros apontados por px e py
 *  entrada:     apontadores px e py
 *  saida:       valor de  *px e *py trocados
 *  suposicoes:  px e py sao apontadores validos
 *  algoritmo:   primeiro guarda o primeiro valor em um temporario e troca
 */
void troca(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
```

```
}

main(void)
{
    int a, b;

    printf("Entre dois numeros: ");
    scanf("%d %d", &a, &b);

    printf("Voce entrou com %d e %d\n", a, b);

    /* Troca a e b -- passa enderecos */
    troca(&a, &b);

    printf("Trocados, eles sao %d e %d\n", a, b);
}
```

A saída deste programa é:

```
Entre dois numeros: 3 5
Voce entrou com  3 e 5
Trocados, eles sao 5 e 3
```

Basicamente, se a função precisa alterar o valor de uma variável na função chamadora, então passamos o endereço da variável como parâmetro real, e escrevemos a função de acordo, ou seja, com um ponteiro como parâmetro formal.

28.8.1 *Arrays* como argumentos de funções

Quando um *array* é passado como argumento para uma função, somente o ponteiro para a primeira posição do *array* é passada e não o conteúdo de todo o *array*. *Arrays* são portanto passados por referência e não por valor.

Ao se definir um *array* como o argumento formal de uma função em C, duas formas podem ser usadas. Elas podem ser vistas abaixo nas definições das funções `func_1()` e `func_2()`.

```
func_1 (char vet [], int ivet[])
{
   vet[3] = 'A';
   vet++;
   ivet += 3;
}

func_2 (char *vet, int *ivet)
{
   vet[4] = 'B';
   vet++;
   ivet += 3;
}

main()
{
  char ender[20];
  char vals[20];

  func_1(ender, vals);
  func_2(ender, vals);
}
```

Observe no exemplo acima que a passagem dos *arrays* ao se chamar as funções `func_1()` e `func_2()` é feita da mesma forma: Usa-se o NOME das variáveis declaradas como *arrays*. Note também o uso

dos argumentos formais nas funções: `vet` e `ivet` podem ser usadas como ponteiros ou como nomes de *arrays* (com a notação indexada por []).

28.8.2 Ponteiros para estruturas como argumentos de funções

Quando estruturas são passadas como argumentos para funções o valor de todo o objeto agregado é passado literalmente. Além disso, se este valor é alterado na função, ele deve ser retornado (via `return`), o que implica em copiar de volta toda a estrutura. Isto pode ser bastante ineficiente no caso de uma estrutura grande (com muitos membros, com membros de tamanho grande como *arrays*, etc.). Assim, em alguns casos é melhor passar ponteiros para estruturas. Repare a diferença com *arrays* passados como argumentos para funções vista na seção anterior.

O programa abaixo é um exemplo do uso de passagem de ponteiros de estruturas para funções:

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

void obtem_endereco(struct endereco *);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco *pender)
{
     printf("Entre rua: ");
     gets(pender->rua);
     printf("Entre cidade/estado/cep: ");
     gets(pender->cidade_estado_cep);
}

void imprime_endereco(struct endereco ender)
{
     printf("%s\n", ender.rua);
     printf("%s\n", ender.cidade_estado_cep);
}

main()
{
     struct endereco residencia;

     printf("Entre seu endereco residencial:\n");
     obtem_endereco(&residencia);

     printf("\nSeu endereco:\n");
     imprime_endereco(residencia);
}
```

Neste caso, `main()` passa para a função `obtem_endereco()` um ponteiro para a variavel `residencia`. `obtem_endereco()` pode então alterar o valor de `residencia` ``remotamente''. Este valor, em `main()`, é então passado para `imprime_endereco()`. Note-se que não é necessário passar um ponteiro para a estrutura se seu valor não será mudado (como é o caso da função `imprime_endereco()`).

De um modo geral, é melhor passar ponteiros para estruturas ao invés de passar e retornar valores de estruturas. Embora as duas abordagens sejam equivalentes e funcionem, o programa irá apenas passar ponteiros ao invés de toda uma estrutura que pode ser particularmente grande, implicando em um tempo final de processamento maior.

28.9 Precedência de operadores

A precedência dos operadores `*` e `&` é alta, a mesma que outros operadores unários. A tabela 6 apresenta a precedência de todos os operadores vistos até agora.

Tabela 6: Precedência e associatividade de operadores

Operador	Associatividade
`() [] -> .`	esquerda para direita
`! - ++ -- * & (cast)` (unários)	direita para esquerda
`* / %`	esquerda para direita
`+ -`	esquerda para direita
`< <= > >=`	esquerda para direita
`== !=`	esquerda para direita
`&&`	esquerda para direita
`||`	esquerda para direita
`= += -= *= /= %=`	direita para esquerda
`,`	esquerda para direita

Notas de rodapé

... tamanho[4]
O operador *sizeof()* pode ser usado para determinar o tamanho de um ponteiro. Por exemplo, *sizeof (char *)*

Next Up Previous
Next: 29 Strings e Ponteiros **Up:** 2 Tópicos Avançados **Previous:** 27 Estruturas
Armando Luiz Nicolini Delgado
2013-10-21