Entendendo como funcionam ponteiros

Marcelo Elias Del Valle

2 de novembro de 2001

marceloelias@iname.com mailto:marceloelias@iname.com

Sumário

1 Introdução

Este documento tem o intuito de explicar como funcionam, exatamente, ponteiros para áreas de memória. Ponteiros são um recurso muito utilizado em programação em qualquer linguagem, mas seu uso se faz mais expressivo quando usando linguagens de baixo nível. Nesse documento, a linguagem utilizada será a linguagem C, devido a vários fatores, dentre estes o fato de ser a mais largamente utilizada em qualquer máquina e sistema operacional e também o fato de não impor restrições ao programador com relação às possibilidades.

2 Pré-requisitos

Presumo que o leitor desse documento já saiba o básico sobre programar em linguagem C, saiba como usar variáveis e funçõese também conheça os tipos básicos de dados (char, int, float, etc.). Leitores com conhecimentos mais avançados e mesmo aqueles que já costumam usar ponteiros também devem enxergar utilidade nesse documento, visto que além de ensinar esse documento destina-se a tentar tirar as dúvidas mais comuns que usuários costumam ter quando trabalhando com ponteiros.

3 Tipos e tamanhos de variáveis

Do ponto de vista do programador C, é muito simples a utilização de variáveis. Quando se quer usar um inteiro, o programador simplesmente declara que quer usá-lo (int i, por exemplo) e atribui valores a esse inteiro. Mas o que é um inteiro? É comum em cursos mais básicos de C o programador aprender que um inteiro do tipo short (short int) pode apenas armazenar valores de -32767 até 32768. Mas a que se deve essa restrição?

Tomemos como exemplo um pequeno programa escrito em linguagem C:

Exemplo 1

```
#include <stdio.h>   int main() {
short int i;
for (i=0; i<10; i++)
printf("i=%d\n");
return 0;
}
```

Esse programa apenas declara um inteiro e imprime 10 linhas na tela, cada uma contendo um número de 0 à 9. Para trasformar esse código em um programa, o leitor já deve saber que é necessário usar um compilador. O compilador, por sua vez, irá ler o código acima e então escrever um programa em linguagem de máquina que faça o que está escrito. A primeira coisa que o programador deve saber, então, é que o compilador entende o código descrito acima de uma forma diferente daquela que nós, humanos, entendemos.

Para nós, esse programa parece apenas imprimir dez números. Para o compilador, porém, ele faz algo mais: ele pede para o sistema operacional alocar memória para um inteiro curto (short int). Um inteiro curto, como já deve ser sabido, ocupa dois bytes ou 16bits na memória. Por esse motivo é que um inteiro pode assumir valores na faixa de -32767 até 32768. 16 bits significam 16 números que podem assumir valores 0 ou 1, ou seja, duas combinações para cada bit. Como são 16, temos dois elevado à 16 combinações possíveis, ou seja, 65536 combinações, o tanto de números que temos de -32767 até 32768. O fato é que para cada tipo de váriável, um tanto de espaço em memória é utilizado. Um short int usa dois bytes, um char usa apenas um byte, um float usa 6 bytes... É importante que o leitor tenha essa noção bem definida antes de avançar o tutorial. Para saber o tamanho de um tipo de dado em C, podemos usar um programa semelhante ao descrito abaixo:

Exemplo 2

```
#define TIPO (char)

#include <stdio.h>   int main() {
printf("O tamanho é %d\n", sizeof(TIPO));
return 0;
}
```

O operador sizeof retorna o tamanho em bytes ocupado por um tipo de dado. Apenas mude a diretiva #define acima para imprimir o tamanho de um tipo de dado diferente.

Pois bem, como dissemos acima, o compilador não apenas escreve instruções para a máquina imprimir dez valores na tela. O compilador também escreve instruções de máquina que pedem para o sistema operacional que ele aloque a memória a ser utilizada. No programa acima esse fato é transparente para op programador pois não é necessário saber disso nessa situação. Em muitas situações, contudo, é necessário não só termos ciência disso, como também termos uma maneira de saber em que local da memória aquela variável (nesse caso, aquele inteiro) foi alocada. Esse fato será estudado a seguir.

4 Como as variáveis são alocadas na memória

Tomemos novamente o caso do exemplo 1. Imaginemos que um programador tenha digitado aquele texto em um arquivo e pedido para o compilador transformá-lo em programa. Em seguida, o programador executa o programa. O momento em que o compilador transforma o texto em programa é chamado de COMPILING TIME (tempo de compilação). O momento em que o programador digita o texto é chamado de DESIGN TIME (tempo de desenho). O momento em que o programa roda é chamado de RUN TIME (tempo de execução).

Em tempo de compilação, o compilador escreveu que a variável i deveria ser alocada e que 2 bytes deveriam ser alocados para essa variável. Em tempo de execução, logo após o usuário pedir para o sistema operacional executar o programa, o sistema aloca os dois bytes, conforme escrito no programa em linguagem de máquina, e devolve para o programa o local na memória (endereço) onde esse inteiro foi alocado para o programa.

No seu programa em linguagem C, você usa o E COMERCIAL (&) como operador de endereço, isto é, você precede uma variável por esse operador quando quer saber o endereço da mesma. No exemplo 1, poderiamos usar &i para saber o endereço de memória onde i foi alocado. O programa a seguir imprime o endereço e o valor de uma variável chamada i:

Exemplo 3:

```
#include <stdio.h>   int main() {
int i;
i=6;
printf(" O valor e i é %d e i está alocado no endereço %d na memória!\n", i,
&i);
return 0;
}
```

Pois bem, a essa altura o leitor já deve ter noção de que, quando falamos em uma variável, não estamos falando apenas de seu valor, devemos sempre ter em mente que aquela variável está sendo armazenada em algum lugar na memória e que esse local na memória é o endereço da variável. E o que é um endereço? É um número que contém a localização exata da variável na memória, ou seja, que APONTA para a variável na memória.

Pergunta (importante): De que tipo de variável é o endereço? ou seja, i é um short int, mas e &i, é o quê? A resposta mais correta para essa pergunta é "não sei"! Em computadores mais antigos, &i era um inteiro sem sinal de 4bytes (de 0 até 4294967296) . Em computadores mais novos, contudo, 4 bytes não são suficientes para endereçar toda a memória, então uma variável de tamanho maior é usada. Esse detalhe é importantíssimo e é aí que a maioria das dúvidas com relação a ponteiros são geradas.

De forma mais clara, do mesmo jeito que existe um tipo de variável chamada char e outra do tipo int, existe uma variável chamada "endereço", ou conforme o jargão, ponteiro. Ponteiro nada mais é do que um tipo de variável que aponta para uma posição de memória. A seguir veremos mais sobre esse tipo de variável e sobre o seu uso.

5 Os ponteiros

Como vimos na seção anterior, ponteiro é apenas um tipo de dado que aponta para uma região na memória. Em linguagem C, a sintaxe de declaração de variáveis do tipo ponteiro é feita como segue:

```
tipo_de_variavel *nome_da_variavel;
```

Por exemplo, declarações de ponteiros válidas seriam:

```
int *i;   char *a;
float *f;
int *k[10];
char *nomes[100];
```

Quando declaramos uma variável como ponteiro, ou seja, com um asterísco (no caso de linguagem C) entre o tipo de dado e o nome da variável, estamos dizendo que o nome da variável não referencia o tipo de dado, mas um endereço que aponta para aquele tipo. Por exemplo, se declaramos i como:

```
char i;
```

e j como:

```
char *j;
```

Qual a diferença entre i e j? A diferença é que i é uma variável do tipo char, ou seja, tem 1 byte, e j é uma variável do tipo ponteiro para char, ou seja, tem o número de bytes suficiente para endereçar uma região na memória. *j, contúdo, refere-se a um char, contido no endereço de memória j. O mapa de memória a seguir ilustra isso:

endereço:	0	1	2	3	4	5	6	7	8	9
conteudo:	i='a'	'5'	'y'	'k'	'z'	'Z'	j=3	(j)	(j)	(j)

No mapa, &i = 0 e &j = 6. Nesse exemplo, considere j como sendo um endereço de 4 bytes (ou seja, um endereço em um sistema onde 4 bytes são suficientes para endereçar a memória). i tem apenas um bytes, então ocupa apenas o endereço 0 (zero). Como j tem 4 bytes, ocupa os endereços 6, 7, 8 e 9. No exemplo, i='a' (um char de 1 bytes) e j = 3 (um inteiro de 32bits que armazena o número 3).

Grosso modo, dizemos que j aponta para o endereço 3. Perceba que j foi declarado como (char *), mas na memória ocupa 4 bytes. Isso porque (char *) é um ponteiro para char, ou seja, um endereço que aponta para um caracter. *j, por sua vez, é o conteúdo desse endereço. Nesse exemplo, é o char contido no endereço 3, 'k'.

Mas e se j tivesse sido declarado como um ponteiro para short int (short int *)? O que seria j? Continuaria sendo o conteúdo dos endereços 6, 7, 8 e 9, ou seja, um endereço que aponta para um int. E o que seria &j? Seria o endereço onde esse endereço está armazenado, ou seja, 6. E o que seria *j? Seria o short int contido no endereço 3, ou seja, os dois bytes contidos nos endereços 3 e 4: 'k' e 'z'.

Esse é o básico sobre ponteiros. É muito importante um bom programador não ter dúvidas em ponteiros, pois ponteiros são usados em praticamente todas as aplicações sérias no tocante à programação. Não é possível criar estruturas de dados, como listas ligadas, tabelas de espalhamento, árvores binárias e várias outras estruturas tão usadas em programação. Esse talvez seja o usomais frequente de ponteiros usado em programação. Outro uso, que é passagem de valores para funções, será visto mais adiante. Por hora, Vamos a alguns exercícios.

5.1 Exercícios

- 1. Se i foi declarado da seguinte forma:

int *i; e sabendo que i=5, responda:
Qual o significado do valor &i? Qual o tamanho de &i?
Qual o significado e qual o tamanho de *i? Qual o tamanho de i?
Se j foi declarado como
int j;
e &j=5 e j=10, quanto vale *i?
(respostas: &i é o endereço de i e tem o tamanho de um endereço, ou de um ponteiro. *i é o conteúdo de i, ou a região da memória apontada por i e tem tamanho de um inteiro, pois i foi declarado como (int *), ou seja, ocmo um ponteiro para um inteiro. O tamanho de i é também o tamanho de um endereço. Se &j=5 e j=10, *i vale 10.)

- 2. Ao terminar a sua execução, sabendo que &a=10 e &b=80, qual o valor, no programa a seguir, de *b?

#include <stdio.h> int main() {
char a, *b;
a=5;
b = &a;
return 0;
}

- 3. Qual o erro do programa a seguir? Escreva um programa semelhante que funcione como deveria.

#include <stdio.h> int main() {
int *i;
*i=5;
printf("*i=%d\n", *i);
return 0;
}

Resposta: O erro está em i não apontar para lugar nenhum. *i é o conteúdo de que endereço? O programa a seguir corige o problema:

```
#include <stdio.h>
int main() {
int *i, j;
i=&j;
*i=5;
printf("*i=%d\n", *i);
return 0;
}
```

Como declaramos j como um inteiro, foi declarado memória para um inteiro j no início do programa. Se fizermos i apontar para essa variável, podemos alterar essa memória alocada para j Perceba que quando fazemos *i=5 estamos

alterando j para 5 também, pois tanto *i quanto j podem ser entendidos como o conteúdo de i ou de &j.

6 Funções - como funcionam de fato

Ok, já vimos um pouco sobre como funcionam ponteiros. Agora, para completarmos o nosso estudo, vamos ver um pouco mais sobre funções e a seguir veremos o uso de ponteiros na passagem de parâmetros de funções e um pouco sobre alocação dinâmica de memória. Essa seção não é absolutamente necessária. Se você achá-la muito complicada, pule para o último parágrafo e continue na próxima seção. A informação aqui contida é apresentada para que não fiquem dúvidas na cabeça do leitor sobre como o programa está funcionando, de fato. Como vimos no início desse documento, sempre que declaramos variáveis em um programa, o compilador escreve, em tempo de compilação, um código que aloca memória para essas variáveis (que pede uma área de memória disponível para o sistema operacional). Imaginemos como fica essa questão de alocação no caso de funções. Considere uma função declarada como segue:

```
void foo(short int arg1, char arg2, float arg3) {   if (arg1==0)
foo(1, 'b', 4.6);
printf("args = %d %c %f\n", arg1, arg2, arg3);
}
```

Agora imagine o programa a seguir, que chama essa função uma vez:

Exemplo 4:

```
#include <stdio.h>   int main() {
foo(0, 'a', 4.5);
return 0;
}
```

O programa acima chama a função duas vezes. Pergunta: Quando e como foi alocada memória para os argumentos dessa função? Poderiamos pensar que o compilador escreveu um código que pedisse ao sistema operacional que alocasse um inteiro (int), um caracter (char) e um número de precisão simples (float) no início do programa. Contudo, ao olharmos mais atentamente para o programa acima, vemos que isso não resolveria o problema, pois dentro da função foo há uma chamada para ela mesma.

Colocando isso em outras palavras, se o sistema operacional tivesse alocado apenas um inteiro, um char e um float para a função, o que aconteceria quando ela chamasse ela mesma? Vamos acompanhar o fluxo do programa para entendermos melhor. No início do programa, a função foo é chamada pela primeira vez. Imaginemos que &arg1 fosse 1, &arg2 fosse 3 e &arg3 fosse 4. Teriamos na memória:

0	1	2	3	4	5	6	7	8	9
	arg1	(arg1)	arg2	arg3	(arg3)	(arg3)	(arg3)	(arg3)	(arg3)

Na chamada da função foo de dentro da função main, esse valores seriam alterados para 0, 'a' e 4.5. Dentro da função foo, era de se esperar então que a função foo novamente fosse chamada com os novos argumentos, esses novos argumentos fossem impressos e então fosse impresso 0, 'a' e 4.5. Contudo, ao se passar os argumentos na segunda vez para a função foo, os argumentos arg1, arg2 e arg3 seriam sobre-escritos e a saída ficaria errada. Abaixo são mostradas a saída esperadas e a obtida:

Saída esperada:

1 'b' 4.6 0 'a' 4.5

Saída obtida:

1 'b' 4.6 1 'b' 4.6

O que deve ficar claro para o leitor aqui é que cada vez que uma função é chamada, novos endereços de memória tem de ser alocados. Não é possível alocar esses endereços uma única vez no carregamento do programa (hora em que executamos o programa). Como isso é feito então?

Quando o compilador transforma nosso código fonte (nosso texto) em linguagem C em código de máquina, ele aloca um certo espaço de memória chamado PILHA do programa. O tamanho dessa pilha normalmente pode ser configurado dependendo do compilador que se esteja utilizando. Uma pilha é uma estrutura de dados simples semelhante a uma pilha de pratos, você vai colocando um prato em cima do outro e o último prato que você colocou é o primeiro que você vai tirar.

Por esse motivo pilhas são também chamadas de LIFOs (Last In First Out). O funcionamento das chamadas de função em um programa segue o mesmo esquema. Quando chamamos uma função (por exemplo a foo) uma vez, colocamos os argumentos dos dados na pilha de

programa, sem apagar o outros dados já existentes nas pilhas. Pense como se tivessemos colocado um prato na mesa. Quando chamamos outra função, colocamos outro prato em cima, ou seja, colocamos os valores dos argumentos no topo da pilha. Quando saímos da função, retiramos os dados do topo da pilha. Tente imaginar essa situação no caso do exemplo 4. Na primeira chamada da função foo, os dados foram colocados no nível 0 da pilha (na mesa). Quando a função foo foi chamada pela segunda vez, os dados foram colocados no nível 1, alguns endereços de memória à frente (em cima do outro prato). Quando saimos função foo chamada pela segunda vez, retiramos os argumentos do nível 1 e os argumentos do nível 0 voltam a ser usados, dentro da primeira chamada da função. Quando saímos novamente da função foo, que tinha sido a primeira a ser chamada, esvaziamos a pilha e voltamos para a função main.

O importante a ser percebido pelo leitor é que em chamadas de funções, sempre que uma função é chamada é alocada nova memória para seus argumentos. No caso, é alocada na pilha do programa, mas é alocada. Esse fato será de vital importância para a próxima seção.

7 Passagem por valor e por referência

Na seção anterior vimos que quando argumentos são passados para uma função, novas áreas de memória são alocadas para os mesmo. A seguir, ilustraremos de uma forma um pouco mais prática esse processo e veremos como, quando e porque usar ponteiros ao passarmos argumentos para uma função.

Para ilustrar melhor o processo, vamos mostrar um caso prático. Vejamos o exemplo a seguir:

Exemplo 5:

```
#include <stdio.h>   void soma_valor(int a, int b, int c) {
c = a + b;
}
void soma_referencia(int a, int b, int *c) {
*c = a + b;
}
int main() {
int c=7;
soma_valor(1,2,c);
printf("soma_valor = %d \n", c);
soma_referencia(1,2,&c);
printf("soma_referencia = %d \n", c);
return 0;
```

```
}
```

Você consegue adivinhar qual a saída desse programa? Observe as duas funções soma_valor e soma_referencia. O objetivo é somar a com b e colocar o resultado em c. Vamos analisar primeiro como trabalha a função soma_valor. Pelos motivos que discutimos na seção anterior, quando se chama uma função, qualquer que seja, estamos alocando memória nova para seus argumentos.

Dessa forma, quando declaramos int c na função main, o compilador escreveu um código pedindo ao sistema operacional para reservar memória para um inteiro. Mas quando passamos esse inteiro para a função soma_valor, um novo inteiro é alocado na pilha do programa (na verdade 3: a, b e c) e o valor do inteiro c declarado na função mais é copiado para essa nova região na pilha. Quando estamos referenciando o inteiro c na função soma_valor, não estamos referenciando a mesma variável (a mesma região de memória) que alocamos na função main, mas sim um outra variável (outra região de memória) alocada na pilha quando a função é chamada.

Preferimos chamar o inteiro declarado na função soma_valor de c e o inteiro declarado na função main de c também, mas são variáveis diferentes, são regiões diferentes alocadas na memória. Dizemos, pelo jargão, que essas variáveis são variáveis locais. O inteiro declarado dentro da função main é local dentro da função main e os inteiros declarados como argumentos na função soma_valor também são locais para a função soma_valor.

Observe novamente o exemplo. Note que a função soma_valor altera o inteiro (int c) da função soma_valor, mas não altera o inteiro (int c) declarado na função main. O primeiro printf, então, nos dá uma saída errada do programa. 1 somado a dois resulta em 3, mas o primeiro printf imprime o número 7 (pois tinhamos inicializado c com 7 no início da função main). Se tivessemos colocado um printf dentro da função soma_valor, contudo (experimente fazer isso), o resultado seria 3.

Deve estar bem claro para o leitor a essa altura que as variáveis locais de uma função nada tem a ver com as variáveis locais de outra função. Nós passamos argumentos como passado na função soma_valor quando não queremos que a função altere o valor de nossas variáveis locais. Esse modo de passar argumentos para uma função é chamado de passagem por valor, pois passamos o valor do dado para a função. Já na função soma_referencia, dizemos que o argumento é passado por

referência, pois estamos passando um endereço, um ponteiro para uma variável, uma referência para uma região de memória.

O argumento c da função soma_referencia foi declarado como (int *). (int *), como já vimos em seções anteriores, é um ponteiro para um inteiro, isso é um endereço de memória onde está alocado um inteiro. Quando passamos, no exemplo 5, os argumentos para a função soma_referencia, perceba que não passamos o valor da variável c (c), mas sim o endereço de c (&c). Quando a função soma_referencia é chamada, são alocados dois inteiros na pilha e um endereço. Então, o valor 1 é copiado para o primeiro inteiro, o valor 2 é copiado para o segundo inteiro e o endereço de c (c declarado na função main) é copiado para o endereço alocado na pilha.

Se a função soma_referencia alterasse o valor de sua variável local (seu argumento) c, ela estaria alterando a cópia do endereço. Contúdo, perceba que não alteramos c, mas sim *c, ou seja, o inteiro referenciado por c. Alterar *c é alterar o conteúdo contido no endereço c da função soma_referencia, mas o conteudo contido no endereço c da função soma_referencia é o conteúdo contido no endereço &c na função main, que é a variável c declarada na função main.

Para simplificar, vamos chamar a variável declara na função main de c_main e a variável declarada na função soma_referencia de c_ref. Quando chamamos a função soma_referencia, estamos passando &c_main (endereço de c_main), ou seja, c_ref, a variável que foi alocada na pilha do programa quando a função foi chamada, irá conter o endereço de c_main:

$$c\_ref = \&c\_main;$$

Quando alteramos, dentro da função, *c_ref, estamos alterando a variável c_main, pois:

$$*cref = *(\&c\_main) = c\_main;$$

Logo, o segundo printf do exemplo imprime o resultado correto (3). Sempre que você quiser criar uma função onde os argumentos passados para ela devem ser alterados, declare esse argumento como um ponteiro e passe o endereço de uma variável para uma função, ao invés de seu valor. Um exemplo típico é a função scanf, tão conhecida por qualquer programador c. Considere o exemplo a seguir:

Exemplo 6:

```
#include <stdio.h>   int main() {
int a;
scanf("%d ", &a);
printf("%d", a);
return 0;
}
```

O programa não faz nada muito complicado, apenas recebe um número digitado pelo usuário e o exibe novamente na tela. Por que passamos o endereço de a (&a) na chamada da função scanf e passamos apenas o valor de a (a) na chamada da função printf? Por que a função scanf altera o valor de a. Se passarmos a variável a para a função scanf, estaremos passando a por valor e não será possível alterar seu valor. Os programadores que criaram a função scanf, então, decidiram que a deveria ser passado por referência, deveriamos passar um (int *) e não um (int). Na função printf, como não precisamos alterar o valor de a dentro da função, podemos passar por valor mesmo.

7.1 Exercícios

- 1. Crie uma função divide, que recebe três argumentos, a, b e c. Declare esses argumentos conforme achar necessário (int, int *, etc.), sendo que a função deve retornar no terceiro argumento (c) o resultado da divisão de a por b.

Resposta:
```
void divide(int a, int b, int *c) {
if (b==0) { //divisão por zero
printf(stderr, "divisão por zero\n");
return;
}
*c=b/a;
}
```

8 Ponteiros e vetores

Já vimos como funcionam ponteiros e como usá-los na passagem de argumentos para uma função, quando queremos que os dados sendo passados sejam alterados. Agora vamos tentar entender a relação de ponteiros com vetores e matrizes e também nos aprofundaremos um pouco no que são matrizes e vetores exatamente.

Imaginemos que tenhamos declarado um vetor de short ints, como segue:

```
short int nome[100];
```

O que o programa que o compilador escreveu faz no lugar dessa declaração? O compilador nada mais faz senão escrever um código que pede para o sistema operacional alocar memória para um endereço (um ponteiro) chamado nome e então alocar mais 200 bytes (100 short ints) na memória e colocar o endereço do primeiro desses 200 bytes em nome. O mapa de memória a seguir ilustra melhor o processo (considerando que um endereço ocupa 4 bytes de memória):

endereço	0	1	2	3	4	5	...	202	203
conteudo	nome=4	(nome)	(nome)	(nome)	nome[0]	nome[0]	nome[...]	nome[99]	nome[99]

Ou seja, quando alocamos um vetor de 100 short ints, não estamos alocando apenas 200 bytes, mas também um endereço para apontarmos para o primeiro short int. Pode parecer incrível numa primeira olhada, mas vetores e ponteiros são praticamente a mesma coisa. O que seria *(nome), então, de acordo com o mapa de memória acima? Seria o conteúdo dos endereços 4 e 5, ou seja, nome [0]. Mas nome é o endereço do primeiro vetor, então nome = &(nome[0]).

Agora vem uma nota importante. nome vale quatro, quanto vale nome+1?? Se você respondeu 5, se enganou! A resposta é 6. Por que 6? Por que estamos somando um ponteiro para short int (short int *), e um short int tem dois bytes. Se nome fosse um ponteiro para um inteiro longo (long int *), nome+1 seria 8, pois um long int tem 4 bytes! A aritmética entre ponteiros é importantíssima em operações com vetores! Observe: nome +1 aponta para o segundo short int do vetor, ou seja, nome+1=&(nome[1]). Analogamente, podemos deduzir que *(nome+1)=nome[1]. Vamos generalizar isso. Sendo i um inteiro válido, temos:

nome+i=&(nome[i]) e
*(nome+i)=nome[i]

Isso é tudo o que você precisa saber para usar ponteiros com vetores. Apenas para consolidar a informação, vamos a um exemplo:

Exemplo 7:

```
#include <stdio.h>   void imprime_nome(char *nome) {
printf("O nome é '%s'\n", nome);
}
int main() {
char nome[50];
scanf("%s", nome);
imprime_nome(nome);
return 0;
```

```
}
```

O exemplo é bem simples. Apenas perceba que nome foi declarado como um vetor de bytes (char), mas foi passado para duas funções que esperavam receber ponteiros. A primeira, scanf, altera os bytes apontados por nome (ou seja, os bytes do vetor). A segunda apenas imprime o nome usando printf.

9 Ponteiros para void

Até agora, sempre que declaramos ponteiros estavamos falando de ponteiros para inteiros, para bytes, para números de ponto flutuante, etc. Contudo, seria válida uma declaração de um ponteiro para void (vazio)? Considere as duas declarações a seguir:

```
int *i;

void *v;
```

Se declaramos i como ponteiro para i, sabemos que &i é o endereço do ponteiro, i é um endereço e *i é um inteiro. Mas se declaramos v como ponteiro para void, sabemos que &v é o endereço do ponteiro v, v é um endereço, mas e *v? É o que? Um inteiro? Um char?

Na verdade, se tentarmos des-referenciar um ponteiro para void (des-referenciar é assumir o conteúdo da referência, se a referência é i, *i é a des-referência), o compilador nos dará um erro. Ponteiros do tipo (void *) são um tipo especial de ponteiros usados quando queremos apontar para qualquer endereço. Vejamos o seguinte exemplo:

Exemplo 8:

```
#include <stdio.h>   int main() {
int *a, c;
char *b;
a = &c;
b = a;
return 0;
}
```

Esse programa gera um aviso (warning) quando tentamos compilá-lo, pois na instrução b = a estamos copiando o valor de um tipo de dado (int *) para outro (char *). Embora ambos sejam endereços e um endereço possa ser perfeitamente copiado para outro endereço, não importa o tipo, o compilador nos avisar quando isso acontece, pois normalmente, se quisessemos copiar a para b teriamos declarado ambos com o mesmo tipo de dado.

Podemos dizer ao compilador, contudo, que estamos cientes de que estamos copiando um tipo de dado para outro alterando a instrução com um cast: b = (char *)a; Ou seja, estamos convertendo a para uma variável do tipo ponteiro para char (char *) antes de copiarmos para b. Essa é a forma que temos de dizer ao compilador que sabemos o que estamos fazendo.

Contudo, para algumas aplicações, queremos apenas um ponteiro, que ora pode estar apontando para um char, ora para um int, etc. Nesses casos, declaramos o ponteiro como um ponteiro para void (void *). Vejamos mais um exemplo:

Exemplo 9:

```
#include <stdio.h>    int main () {
void *v;
int *i, c;
c=5;
i = &c;
v = i;
printf("*v = %d *i=%d \n", *((int *)v), *c );
return 0;
}
```

Esse programa compila e roda normalmente. Perceba que não precisamos fazer um cast para atribuir o valor do endereço i para o endereço v (v=i), pois v foi declarado como void e pode apontar para qualquer tipo de dado. A declaração v = (void *) i; também seria válida. Observe, porém, que na função printf não pudemos usar *v para referenciar um inteiro, tivemos que converter v para um ponteiro para inteiro (int *) antes de de desreferenciá-lo: *((int *)v).

10 Alocação dinâmica de memória

Até agora,vimos que para usarmos variáveis (regiões de memória) em um programa, precisamos declarar essas variáveis no código fonte e quando compilamos o programa o compilador escreve instruções

em código de máquina que pedem ao sistema operacional que este aloque a memória que declaramos e devolva o endereço dessa memória alocada para o nosso programa.

Isso funciona muito bem, mas imagine o caso de um processador de textos, por exemplo. Imagine como seria se fossemos programar um processador de textos e tivessemos que declarar o tamanho do texto que o usuário poderia digitar, quando estivessemos fazendo nosso programa. Isso seria horrível e totalmente limitado.

Por esse motivo, algumas vezes nós não desejamos alocar a memória que utilizaremos no código fonte do programa. Ao invés disso, nós apenas declaramos ponteiros para áreas de memória (ou seja, alocamos apenas endereços quando o programa é carregado) e pedimos para o sistema operacional alocar memória (que apontaremos com esses endereços) conforme a necessidade.

A memória que é alocada quando o programa é inicializado (as variáveis declaradas no programa) é chamada de memória alocada estaticamente, pois as regiões de memória que são utilizadas pelas variáveis do programa são alocadas quando o programa é carregado e desalocada quando o programa é finalizado. Quando pedimos, no meio da execução do programa, para que o sistema operacional aloque memória para nosso programa, estamos alocando memória dinamicamente.

Veremos agora como é feito para que o nosso programa aloque memória no meio de sua execução. Isso pode ser feito em linguagem C através de uma função chamada malloc (memory alloc). Mas antes de falarmos sobre essa função, vejamos o exemplo a seguir.

Exemplo 10:

```
#include <stdio.h>   int main() {
int *i;
*i=5;
printf("%d\n", *i);
return 0;
}
```

Esse é, de longe, o erro mais comum que programadores C cometem ao aprender a trabalhar com ponteiros. O pior é que o programa compila e roda. Na linha ***i=5**, atribuimos um valor para ***i**. Mas no início do nosso programa, nós não declamos, em lugar algum, um inteiro. Nós só declaramos um endereço, isso é, um ponteiro, mas esse endereço não aponta para lugar nenhum ainda, não foi alocada

memória para um inteiro, para que pudessemos apontar para o mesmo. Como i aponta para um endereço qualquer da memória (pois não o inicializamos com valor algum), o nosso programa seria finalizado ao executarmos o mesmo e receberiamos um erros de Segmentation Fault (Falta de segmentação) por estamos tentando acessar um endereço de memória que não alocamos.

Se quisermos colocar o valor 5 em um inteiro, precisamos alocá-lo primeiro. O exemplo a seguir usa a função malloc para fazer isso. Exemplo 11:

```
#include <stdio.h>   #include <stdlib.h>
int main() {
int *i;
i = (int *) malloc(sizeof(int));
*i=5;
printf("%d\n", *i);
return 0;
}
```

Esse programa não apenas declara um ponteiro para um inteiro. Eis o protótipo da função malloc:

```
void *malloc(size_t size);
```

size_t pode ser entendido como um inteiro. Pense que está escrito int size (tamanho). Quando a função malloc é chamada, é enviada uma chamada para o sistema operacional (system call) pedindo para que size bytes de memória sejam alocados e é devolvido um ponteiro para void (void *) referenciando o primeiro bytes da região alocada. No exemplo, usamos o operador sizeof() para alocarmos o tamanho em bytes de um inteiro. Como i foi declarado como (int *), usamos um cast para converter o ponteiro para void retornado pela função malloc para um ponteiro para int.

Existem algumas variantes da função malloc como a calloc, que preenche a área de memória alocada com zeros, além de alocá-la. A região de memória alocada pelo sistema operacional não é necessariamente preenchida com zeros quando a alocamos, pois outro programa poderia estar usando aquela memória antes de alocarmos. Veja a documentação do seu compilador para conhecer outras variantes.

Como último exemplo sobre o assunto, vamos mostrar um programa que alloca não uma variável na memória, mas um vetor:

Exemplo 12:

```
#include <stdio.h>   #include <stdlib.h>
int main() {
int *i, j;
i = (int *) malloc(50 * sizeof(int));
for (j=0; j<50; j++) {
i[j]=j;
printf("%d\n", *(i+j));
}
return 0;
}
```

Esse programa aloca 50 * sizeof(int), ou seja, 50 inteiros. A seguir, um loop for assinala um valor para cada inteiro do vetor e exibe o valor na tela. Perceba que a notação *(i+j) ou i[j] significam, na verdade, a mesma coisa.

11 Ponteiros que apontam para ponteiros

Consideremos agora o caso de um ponteiro que aponta para outro. Vejamos o seguinte exemplo:

```
#include <stdio.h>   #include <stdlib.h>
int main() {
int **i;
i = (int **) malloc(sizeof(int *));
*i = (int *) malloc(sizeof(int));
**i=10;
printf("%d\n", **i);
return 0;
}
```

Esse exemplo é mesmo complicado. Ao declararmos int **i, estamos declarando um ponteiro que aponta para um ponteiro, ou seja:

1. i é um endereço que aponta para a região de memória onde está *i
2. *i é um endereço que aponta para a região de memória onde está **i
3. **i é um inteiro.

No primeiro malloc, alocamos memória para o endereço *i e fizemos o endereço i apontar para esse endereço, ou seja, alocamos uma região de memória que tem tamanho sizeof(int *). No segundo, alocamos memória para esse inteiro (i) e fizemos com que o endereço *i apontasse para ele. Na instrução seguinte, atribuimos um valor para esse inteiro e então usamos printf para imprimí-lo.**

11.1 Exercícios

- 1. Faça um programa que aloque um vetor de ponteiros para char na memória e, então, faça cada elemento do vetor apontar para um vetor de char.

Resposta:
```
#include <stdio.h>
#include <stdlib.h>
int main() {
char **nomes;
int i;
//50 nomes, ou 50 vetores a alocar
nomes = (char **) malloc(50*sizeof(char *));
for (i=0; i<50; i++) {
//Cada nome tem até 20 bytes
*(nomes+i) = (char *) malloc(20*sizeof(char));
sprintf(nomes[i], "nome de número %d", i);
printf("nomes[%d]='%s'\n", i, nomes[i]);
}
return 0;
}
```

Perceba na resposta do exercício que para cada ponteiro do vetor de ponteiro alocado, foi alocado um vetor de char.

12 Ponteiros e estruturas

Não há muito a se dizer sobre a utilização de ponteiros com structs. Contudo, apenas para que o leitor já tenha visto uma vez essa utilização de ponteiros, colocaremos um exemplo aqui. A utilização de ponteiros com structs é especialmente importante quando trabalhando com orientação a objetos em uma linguagem não orientada a objetos, como C, mas isso está fora do escopo desse documento.

Exemplo 13:

```
#include <stdio.h>   #include <stdlib.h>
typedef struct _teste {
short int a, b, c, d;
} teste;
void imprime_teste(teste *t) {
printf("Os valores da estrutura encontrada no endereço %p são: %d, %d, %d
e %d\n",
t, t->a, t->b, t->c, t->d);
}
int main() {
teste t1, *t2;
t1.a=5; t1.b=6; t1.c=7; t1.d=8;
t2 = (teste *) malloc(sizeof(teste));
t2->a=4; t2->b=3; t2->c=2; t2->d=1;
imprime_teste(&t1);
imprime_teste(t2);
}
```

Nesse exemplo, foi declarada uma struct chamada _teste e foia tribuido (com typedef) o nome teste como alias para (struct _teste). A função imprime_teste apenas imprime os valores internos da estrutura e o endereço de memória onde a estrutura está alocada. A função main além de setar variáveis internas da estrutura e chamar a função imprime_teste, usa a função malloc para alocar memória para uma estrutura.

A primeira coisa que se deve saber ao se usar uma struct é que os dados declarados dentro da mesma são alocados na memória em sequência, ou seja, se a estrutura é formada por 4 inteiros curtos (short ints), ter um ponteiro para essa estrutura seria o mesmo que ter um vetor de 4 inteiros curtos. Structs, contudo, custumam ser usadas quando se quer usar tipos diferentes de dados (diferente do caso do vetor, onde só um tipo de dado é usado). Esse fato permite que aloquemos memória para uma estrutura com a função malloc.

Outro detalhe muito importante é o modo como acessamos as variáveis internas de uma struct. Se declaramos uma variável do tipo teste chamada t1 (teste t1), podemos acessar, como já é sabido, suas variáveis internas por:

t1.a, t1.b, t1.c e t1.d

Mas e se tivermos declarado um ponteiro para teste chamado t2 (teste *t2), como podemos acessar essas variáveis internas? Uma forma é:

(*t2).a, (*t2).b, (*t2).c e *(t2).d

Contudo, essa forma é muito pouco utilizada, por ser trabalhosa de se escrever. Podemos usar a notação a seguir para referenciar as variáveis internas de t2:

t2->a é a mesma coisa que (*t2).a

t2->b é a mesma coisa que (*t2).b

t2->c é a mesma coisa que (*t2).c

t2->d é a mesma coisa que (*t2).d

Essa foi a forma utilizada no programa de exemplo.

12.1 Exercícios

- 1. Altere a função imprime_teste do programa de exemplo acima de modo que ele use outra notação para referencias as variáveis internas da struct.

Resposta: void imprime_teste(teste *t) {
printf("Os valores da estrutura encontrada no endereço %p são: %d, %d, %d e %d\n",
t, (*t).a, (*t).b, (*t).c, *(t).d);
}

Referências

[1] MPage - O portal brasileiro de desenvolvimento de jogos `http://mlinuxer.cjb.net`