Ponteiros, ponteiros e vetores e alocação dinâmica de memória

Ponteiros

Ponteiros ou apontadores (em inglês **pointers**) são variáveis cujo conteúdo é um endereço. Já vimos que variáveis são posições na memória que podem conter um determinado valor dependendo de seu tipo (char, int, float, double, etc...). Assim, uma variável do tipo ponteiro contém valores que são na verdade endereços para outras posições de memória. Os nomes apontador, ponteiro ou pointer são utilizados com o mesmo significado.

Como declarar um ponteiro

Uma variável tipo ponteiro, "aponta" para uma variável de um determinado tipo conhecido (char, int, float, double, etc.), ou seja, contém um endereço de uma variável de um determinado tipo. Assim é necessário na declaração de um ponteiro, especificar para qual tipo de variável ele irá apontar. O operador * indica que a variável é um apontador.

```
int *pi ; /* ponteiro para int */
char *pc; /* ponteiro para char */
float *pf; /* ponteiro para float */
double *pd; /* ponteiro para double */
```

Uso de ponteiro

O operador & significa "endereço de".

```
int a;
char b;

pi = &a; /* pi fica com o endereço de a */
pc = &b; /* pc fica com o endereço de c */

/* Os dois comandos abaixo são equivalentes */
a = 45;
*pi = 45;

/* idem para */
b = 'a'; e
*pc = 'a';
```

Veja o programa abaixo e o que ele imprime no vídeo. O formato **p** é um formato para imprimir o conteúdo de um ponteiro, ou seja, um endereço.

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

```
int main() {
int a, *pa;
double b, *pb;
char c, *pc;
/* atribuições de endereços */
pa = &a; pb = &b; pc = &c;
/* atribuição de valores */
a = 1; b = 2.34; c = '0';
printf("\n valores:%5d %5.21f %c", a, b, c);
printf("\n ponteiros:%5d %5.21f %c", *pa, *pb, *pc);
printf("\n enderecos:%p %p %p", pa, pb, pc);
/* mais atribuições de valores usando os ponteiros */
*pa = 77; *pb = 0.33; *pc = '#';
printf("\n valores :%5d %5.21f %c", a, b, c);
printf("\n ponteiros:%5d %5.21f %c", *pa, *pb, *pc);
printf("\n enderecos:%p %p %p", pa, pb, pc);
system ("pause"); return 0;
valores:
                2.34
ponteiros:
              1
                  2.34
enderecos:0063FDDC 0063FDD4
                              0063FDD3
valores :
            77
                 0.33
ponteiros:
             77
                  0.33
enderecos:0063FDDC 0063FDD4 0063FDD3
```

Um pouco sobre a memória e endereços

O formato **%p** mostra o endereço. Os endereços são dependentes do particular computador e do particular compilador. Também depende do lugar em que o programa está carregado na memória. Assim um mesmo programa pode ocupar posições diferentes de memória em duas execuções diferentes. O mesmo pode ter suas variáveis alocadas em lugares diferentes se for compilado com diferentes compiladores. No exemplo acima estamos usando um computador com processador INTEL onde os endereços possuem 2 partes: número do segmento e deslocamento dentro do segmento. As variáveis possuem os seguintes endereços (em hexadecimal):

```
a - 0063 FDDC (4 bytes FDDC a FDDF)
b - 0063 FDD4 (4 bytes FDD4 a FDD7)
c - 0063 FDD3 (1 byte FDD3)
```

Veja outro exemplo de alocação de endereços na memória:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int a, aa;
double b, bb;
char c, cc;
int vi[5];
double vd[5];
char vc[5];
int i;
printf("\n endereco de a = %p endereco de aa = %p", &a, &aa);
printf("\n endereco de b = %p endereco de bb = %p", &b, &bb);
printf("\n endereco de c = %p endereco de cc = %p", &c, &cc);
for (i = 0; i < 5; i++)
  printf("\n endereco de vi[%1d] = %p", i, &vi[i]);
for (i = 0; i < 5; i++)
  printf("\n endereco de vd[%1d] = %p", i, &vd[i]);
for (i = 0; i < 5; i++)
  printf("\n endereco de vc[%1d] = %p", i, &vc[i]);
printf("\n endereco de vi = %p\n endereco de vd = %p\n endereco de vc =
%p", vi, vd, vc);
system("pause"); return 0;
}
 endereco de a = 0063FDA0 endereco de aa = 0063FD9C
 endereco de b = 0063FD94 endereco de bb = 0063FD8C
 endereco de c = 0063FD8B endereco de cc = 0063FD8A
 endereco de vi[0] = 0063FDD4
 endereco de vi[1] = 0063FDD8
 endereco de vi[2] = 0063FDDC
 endereco de vi[3] = 0063FDE0
 endereco de vi[4] = 0063FDE4
 endereco de vd[0] = 0063FDAC
 endereco de vd[1] = 0063FDB4
 endereco de vd[2] = 0063FDBC
 endereco de vd[3] = 0063FDC4
 endereco de vd[4] = 0063FDCC
 endereco de vc[0] = 0063FDA7
 endereco de vc[1] = 0063FDA8
 endereco de vc[2] = 0063FDA9
 endereco de vc[3] = 0063FDAA
 endereco de vc[4] = 0063FDAB
 endereco de vi = 0063FDD4
 endereco de vd = 0063FDAC
 endereco de vc = 0063FDA7
```

Qual o endereço do ponteiro?

Ponteiros também são variáveis, portanto ocupam posições na memória. Veja o programa abaixo que mostra o endereço das variáveis e dos ponteiros.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int a, *pa;
double b, *pb;
char c, *pc;
/* atribuições de endereços */
pa = &a; pb = &b; pc = &c;
/* mostra o endereço das variáveis */
printf("\n\n enderecos de a, b e c:%p %p %p", pa, pb, pc);
/* mostra o endereço das variáveis de outra forma */
printf("\n\n enderecos de a, b e c (outra forma):%p %p %p", &a, &b,
&c);
/* mostra o endereço dos ponteiros */
printf("\n\n enderecos dos ponteiros:%p %p %p", &pa, &pb, &pc);
system("pause"); return 0;
enderecos de a, b e c:0063FDDC 0063FDD4 0063FDD3
enderecos de a, b e c (outra forma):0063FDDC 0063FDD4 0063FDD3
enderecos dos ponteiros:0063FDE8 0063FDE4 0063FDE0
```

E ponteiros de ponteiros?

Como os ponteiros também são variáveis, também é possível a construção de ponteiro para um ponteiro. Começa a ficar meio artificial, mais existem exemplos onde isso é útil.

Veja o exemplo abaixo e algumas maneiras inusitadas de somar 1 a um inteiro.

Ponteiros, ponteiros e vetores e alocação dinâmica de memória Mac122 – Marcilio

```
valor de a = 1
valor de a = 2
valor de a = 3
valor de a = 4
```

Ponteiros e o scanf

```
int i, *pi;

pi = &i;
scanf("%d", &i); é equivalente a
scanf("%d", pi); note que não é &pi
```

Ponteiros e parâmetros de retorno de funções

Já vimos anteriormente, no estudo das funções em C, que para que tenhamos parâmetros de retorno, isto é, parâmetros que são alterados dentro da função e retornam com esse novo valor, é necessário que passemos o endereço da variável como parâmetro. O endereço nada mais é que um ponteiro.

Veja a função troca abaixo, o programa e o que será impresso.

```
#include <stdio.h>
#include <stdlib.h>
/* Função que recebe 2 variáveis reais e permute o valor */
int troca (double *a, double *b) {
  double aux;
  aux = *a;
  *a = *b;
  *b = aux;
}
int main() {
double x = 2.78,
      y = 3.14,
       \star xx = &x,
      *yy = &y;
/* x e y antes e depois da troca */
printf("\n^* antes de chamar a troca x = %15.51f e y = %15.51f", x, y);
troca (&x, &y);
printf("\n* depois de chamar a troca x = %15.51f e y = %15.51f", x, y);
/* x e y antes e depois da troca usando ponteiros */
*xx = 1.23456;
 *yy = 6.54321;
```

Ponteiros, ponteiros e vetores e alocação dinâmica de memória Mac122 – Marcilio

Ponteiros e vetores

O que acontece quando somamos (ou subtraímos) algo do valor de um ponteiro? Como não sabemos em geral como as variáveis estão na memória, isto é, quem é vizinho de quem, isso não faz muito sentido. Entretanto, se o ponteiro aponta para um vetor, isso pode fazer sentido, pois os elementos do vetor estão contíguos na memória.

```
int v[100];
int *pv;

pv = &v[0];
*pv = 0; éo mesmo que v[0] = 0;

*(pv+i) = 0; éo mesmo que v[i] = 0;

for (i=0; i<100; i++) *(pv+i) = 0; zera o vetor v</pre>
```

Os incrementos ou decrementos no valor do ponteiro não são em bytes e sim acompanham o tamanho do elemento apontado.

```
int *pi;
double *pa;
char *pc;

pi = pi + k; soma k*sizeof(int) em pi
pa = pa + k; soma k*sizeof(double) em pa
pc = pc + k; soma k em pc - o sizeof(char) é 1.
```

O nome do vetor também é um ponteiro

```
int v[100];
*v = 33; éo mesmo que v[0] = 33;
```

Ponteiros, ponteiros e vetores e alocação dinâmica de memória Mac122 – Marcilio

```
*(v+2) = 44; \acute{e} o mesmo que v[2] = 44;
for (i = 0; i < 100; i++) *(v+i) = 0; zera o vetor v
```

O nome do ponteiro também é um vetor

Podemos também usar um ponteiro como um vetor.

```
int v[100];
int *pv;

pv = v; também pode ser pv = &v[0];
for (i = 0; i < 100; i++) pv[i] = 0;</pre>
```

Ponteiros e matrizes de mais de 1 índice

Supondo agora as declarações:

```
int x[100], z[10][100];
```

Quando usamos **x**[k] nos referimos ao **k**-ésimo elemento de **x**.

Quando usamos apenas x nos referimos ao endereço de início do vetor, ou seja, &x[0].

Quando usamos x + k nos referimos ao endereço de x[k], ou seja, &x[k].

Da mesma forma com matrizes de mais de 1 índice.

Quando usamos z[i][k] nos referimos ao k-ésimo elemento da i-ésima linha de z.

Quando usamos **z**[i] estamos nos referindo ao endereço de início da linha i de **z**, ou seja, &**z**[i][0].

Quando usamos **z**[i] + **k** estamos nos referindo ao endereço do **k**-ésimo elemento da linha i de **z**, ou seja, &**z**[i][k].

Usando a equivalência entre vetores e ponteiros - exemplo

Considere a função **zera** que atribui zero aos n primeiros elementos de um vetor:

```
void zera (int a[], int n) {
   int i;
   for (i = 0; i < n; i++) a[i] = 0;
}</pre>
```

Podemos usar esta mesma função para zerar um trechos internos de vetores e matrizes. Veja os exemplos abaixo:

```
int x[100], y[100], z[10][100];
/* zera 50 primeiros de x */
zera(x,500);
/* zera 50 últimos x */
zera(&x[50], 50);
/* ou de outra maneira */
zera(x + 50, 50);
/* zera 10 elementos de y a partir de y[15] */
zera(&y[15], 10);
/* ou de outra maneira */
zera(y + 15, 10);
/* zera 30 elementos iniciais da linha 5 de z */
zera(&z[5][0], 30);
/* ou de outra maneira */
zera(z[5], 30);
/* 20 elementos da linha 2 de z a partir de z[2][50] */
zera(&z[2][50], 20);
/* ou de outra maneira */
zera(z[2] + 50, 10);
```

Alocação dinâmica de memória – funções malloc calloc e free

Durante a execução de um programa é possível alocar certa quantidade de memória para conter dados do programa

A função malloc (n) aloca dinamicamente n bytes e devolve um ponteiro para o início da memória alocada.

A função calloc(n, t) aloca dinamicamente n elementos de tamanho t e devolve ponteiro para o início da memória alocada.

A função free (p) libera a região de memória apontada por p. O tamanho liberado está implícito, isto é, é igual ao que foi alocado anteriormente por malloc ou calloc. Não se deve liberar espaço que não foi obtido explicitamente por uma chamada de malloc ou calloc.

Outro detalhe é que malloc e calloc devolvem ponteiros. Como esse ponteiro é para um tipo específico de dados (int, char, double, etc...) deve-se aplicar uma função de conversão de tipos após a chamada de malloc ou calloc para prover o alinhamento necessário. Por exemplo, int tem que ser alocado em posição par de memória.

Os comandos abaixo alocam dinamicamente um inteiro e depois o liberam.

```
#include <stdlib.h>
int *pi;

pi = (int *) malloc (sizeof(int));
...
free(pi);

ou ainda:

#include <stdlib.h>
int *pi;

pi = (int *) calloc (1, sizeof(int));
...
free(pi);
```

As funções malloc e calloc não tem um tipo específico. Assim, (int *) converte seu valor em ponteiro para inteiro. Como não sabemos necessariamente o comprimento de um inteiro (2 ou 4 bytes dependendo do compilador), usamos como parâmetro a função sizeof(int).

Alocação dinâmica de vetores

Conforme sabemos, quando declaramos um vetor é necessário fixar o seu tamanho. Com isso, temos sempre que declarar o seu tamanho máximo, pois o tamanho realmente utilizado só será conhecido durante a execução do programa.

Com alocação dinâmica podemos resolver esse problema. Veja o trecho abaixo:

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  int *v;
  int i, n;
  scanf("%d", &n); /* le n */
  /* aloca n elementos para v */
  v = (int *) malloc(n*sizeof(int));
  /* zera e utiliza o vetor v com exatamente n elementos */
  for (i = 0; i < n; i++) v[i] = 0;
  ...
  /* libera os n elementos de v quando não mais necessário */
  free(v);
}</pre>
```

Outros exemplos:

```
char *pa; int pb;
...
pa = (char *) malloc(1000); /* aloca vetor com 1000 bytes */
pa = (char *) malloc(1000*sizeof(char)); /* o mesmo */
pb = (int *) malloc(1000); /* aloca vetor com 250 int */
pb = (int *) malloc(250*sizeof(int)); /* o mesmo */
```

Usando calloc, os comandos acima ficariam:

```
char *pa; int pb;
...
pa = (char *) calloc(1000,1); /*aloca vetor com 1000 bytes*/
pa = (char *) calloc(1000,sizeof(char)); /* o mesmo */
pb = (int *) calloc(1000,1); /* aloca vetor com 250 int */
pb = (int *) calloc(250,sizeof(int)); /* o mesmo */
```