PONTEIROS

O C é altamente dependente dos ponteiros. Para ser um bom programador em C é fundamental que se tenha um bom domínio deles. Por isto, recomendo ao leitor um carinho especial com esta parte do curso que trata deles. Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo para se fazer um introdução básica à linguagem C precisa-se deles.

***O Ministério da Saúde adverte**: o uso descuidado de ponteiros pode levar a sérios bugs e a dores de cabeça terríveis **:-)**.*

Como Funcionam os Ponteiros

Os **int**s guardam inteiros. Os **float**s guardam números de ponto flutuante. Os **char**s guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Voce anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um inteiro, isto é, guarda o endereço de um inteiro.

Declarando e Utilizando Ponteiros

Para declarar um ponteiro temos a seguinte forma geral:

*tipo_do_ponteiro **nome_da_variável;*

É o asterisco (*****) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp,*pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstânicias pode levar a um travamento do micro, ou a algo pior.

O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count;
```

Criamos um inteiro **count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&count** nos dá o endereço de count, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador "inverso" do operador **&**. É o operador ***. No exemplo acima, uma vez que fizemos **pt=&count** a expressão ***pt** é equivalente ao próprio **count**. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer ***pt=12**.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador * ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador * (multiplicação) não é o mesmo operador que o * (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
int main ()
{
      int num,valor;
      int *p;
      num=55;
      p=&num;     /* Pega o endereco de num */
      valor=*p;       /* Valor e igualado a num de uma
maneira indireta */
      printf ("\n\n%d\n",valor);
      printf ("Endereco para onde o ponteiro aponta:
%p\n",p);
      printf ("Valor da variavel apontada: %d\n",*p);
      return(0);
}
```

```
#include <stdio.h>
int main ()
{
      int num,*p;
      num=55;
      p=&num;     /* Pega o endereco de num */
      printf ("\nValor inicial: %d\n",num);
      *p=100; /* Muda o valor de num de uma maneira indireta
*/
      printf ("\nValor final: %d\n",num);
      return(0);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve imprimir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1=p2**. Repare que estamos fazendo com que **p1** aponte para o mesmo lugar que **p2**. Se quisermos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por **p2** devemos fazer ***p1=*p2**. Basicamente, depois que se aprende a usar os dois operadores (**&** e *) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez insisto. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;    ou    p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira. Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (**==** e **!=**). No caso de operações do tipo **>**, **<**, **>=** e **<=** estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1>p2
```

Há entretanto operações que você *não* pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair **float**s ou **double**s de ponteiros.

AUTO AVALIAÇÃO

Veja como você está.

a) Explique a diferença entre

```
p++;        (*p)++;        *(p++);
```

- O que quer dizer *(p+10);?
- Explique o que você entendeu da comparação entre ponteiros

b) Qual o valor de y no final do programa? Tente primeiro descobrir e depois verifique no computador o resultado. A seguir, escreva um /* comentário */ em cada comando de atribuição explicando o que ele faz e o valor da variável à esquerda do '=' após sua execução.

```
int main()
{
    int y, *p, x;
    y = 0;
    p = &y;
    x = *p;
    x = 4;
    (*p)++;
    x--;
    (*p) += x;
    printf ("y = %d\n", y);
    return(0);
}
```

Ponteiros e Vetores

Veremos nestas seções que ponteiros e vetores têm uma ligação muito forte.

- Vetores como ponteiros

Vamos dar agora uma idéia de como o C trata vetores.

Quando você declara uma matriz da seguinte forma:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo

O compilador então aloca este número de bytes em um espaço livre de memória. O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.

Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

nome_da_variável[índice]

Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:

**(nome_da_variável+índice)*

Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero. Então sabemos que:

**nome_da_variável* é equivalente a *nome_da_variável[0]*

Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices. Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura sequencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
int main ()
{
       float matrx [50][50];
       int i,j;
       for (i=0;i<50;i++)
              for (j=0;j<50;j++)
                     matrx[i][j]=0.0;
       return(0);
}
```

Podemos reescrevê-lo usando ponteiros:

```
int main ()
{
       float matrx [50][50];
       float *p;
       int count;
       p=matrx[0];
       for (count=0;count<2500;count++)
         {
              *p=0.0;
              p++;
         }
       return(0);
}
```

No primeiro programa, *cada* vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

 /* as operacoes a seguir sao invalidas */

vetor = vetor + 2;     /* ERRADO: vetor nao e' variavel */
vetor++;               /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;      /* ERRADO: vetor nao e' variavel */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um Lvalue. Lvalue, significa "Left value", um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros compiladores dirão que tem-se "incompatible types in assignment", tipos incompatíveis em uma atribuição.

```
/* as operacoes abaixo sao validas */
 ponteiro = vetor;      /* CERTO: ponteiro e' variavel */
 ponteiro = vetor+2;    /* CERTO: ponteiro e' variavel */
```

O que você aprendeu nesta seção é de suma importância. Não siga adiante antes de entendê- la bem.

- Ponteiros como vetores

Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como consequência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
int main ()
{
      int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int *p;
      p=matrx;
      printf ("O terceiro elemento do vetor e: %d",p[2]);
      return(0);
}
```

Podemos ver que **p[2]** equivale a ***(p+2)**.

- Strings

Seguindo o raciocínio acima, nomes de strings, são do tipo **char***. Isto nos permite escrever a nossa função **StrCpy()**, que funcionará de forma semelhante à função **strcpy()** da biblioteca:

```
#include <stdio.h>
void StrCpy (char *destino,char *origem)
{
while (*origem)
        {
        *destino=*origem;
        origem++;
        destino++;
        }
*destino='\0';
}
int main ()
{
       char str1[100],str2[100],str3[100];
       printf ("Entre com uma string: ");
       gets (str1);
       StrCpy (str2,str1);
       StrCpy (str3,"Voce digitou a string ");
       printf ("\n\n%s%s",str3,str2);
       return(0);
}
```

Há vários pontos a destacar no programa acima. Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como **gets()** e **strcpy()** funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia. No comando **while (*origem)** estamos usando o fato de que a string termina com '\0' como critério de parada. Quando fazemos **origem++** e **destino++** o leitor poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que recomendei que se deveria fazer, no final de uma seção anterior. O que o leitor talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro **origem** na função **StrCpy()** o ponteiro **str2** permanece inalterado na função **main()**.

- Endereços de elementos de vetores

Nesta seção vamos apenas ressaltar que a notação

&nome_da_variável[índice]

é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a nome_da_variável + indice. É interessante notar que, como consequência, o ponteiro **nome_da_variável** tem o endereço **&nome_da_variável[0]**, que indica onde na memória está guardado o valor do primeiro elemento do vetor.

- Vetores de ponteiros

Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

No caso acima, **pmatrx** é um vetor que armazena 10 ponteiros para inteiros.

Inicializando Ponteiros

Podemos inicializar ponteiros. Vamos ver um caso interessante dessa inicialização de ponteiros com strings.

Precisamos, para isto, entender como o C trata as strings constantes. Toda string que o programador insere no programa é colocada num banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela string (que está no banco de strings). É por isto que podemos usar **strcpy()** do seguinte modo:

```
strcpy (string,"String constante.");
```

strcpy() pede dois parâmetros do tipo **char***. Como o compilador substitui a string **"String constante."** pelo seu endereço no banco de strings, tudo está bem para a função **strcpy()**.

O que isto tem a ver com a inicialização de ponteiros? É que, para uma string que vamos usar várias vezes, podemos fazer:

```
char *str1="String constante.";
```

Aí poderíamos, em todo lugar que precisarmos da string, usar a variável **str1**. Devemos apenas tomar cuidado ao usar este ponteiro. Se o alterarmos vamos perder a string. Se o usarmos para alterar a string podemos facilmente corromper o banco de strings que o compilador criou.

Mais uma vez fica o aviso: ponteiros são poderosos mas, se usados com descuido, podem ser uma ótima fonte de dores de cabeça.

Ponteiros para Ponteiros

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

*tipo_da_variável **nome_da_variável;*

Algumas considerações: ****nome_da_variável** é o conteúdo final da variável apontada; ***nome_da_variável** é o conteúdo do ponteiro intermediário.

No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros (UFA!) e assim por diante. Para fazer isto (não me pergunte a utilidade disto!) basta aumentar o número de asteriscos na declaracão. A lógica é a mesma.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no exemplo abaixo:

```
#include <stdio.h>
int main()
{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi;            /* pf armazena o endereco de fpi */
    ppf = &pf;            /* ppf armazena o endereco de pf */
    printf("%f", **ppf);  /* Imprime o valor de fpi */
    printf("%f", *pf);    /* Tambem imprime o valor de fpi */
    return(0);
}
```

AUTO AVALIAÇÃO

Veja como você está.

Verifique o programa abaixo. Encontre o seu erro e corrija-o para que escreva o numero 10 na tela.

```
#include <stdio.h>
int main()
{
    int x, *p, **q;
    p = &x;
    q = &p;
    x = 10;
    printf("\n%d\n", &q);
    return(0);
}
```

Cuidados a Serem Tomados ao se Usar Ponteiros

O principal cuidado ao se usar um ponteiro deve ser: saiba sempre *para onde* o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como ***não*** usar um ponteiro:

```
int main () /* Errado - Nao Execute */
{
       int x,*p;
       x=13;
       *p=x;
       return(0);
}
```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

AUTO AVALIAÇÃO

Veja como você está.

Escreva um programa que declare uma matriz 100x100 de inteiros. Você deve inicializar a matriz com zeros usando ponteiros para endereçar seus elementos. Preencha depois a matriz com os números de 1 a 10000, também usando ponteiros.