1. **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, recomenda-se um cuidado especial no estudo desse tema. Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo os programas mais simples se utilizam deles.

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

* 1. **FUNCIONAMENTO**

Os [**int**s](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) guardam inteiros. Os [**float**s](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) guardam números de ponto flutuante. Os [**char**s](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) 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 utilizar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois utilizar.

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 empresa. 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, informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro [**int**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) aponta para um inteiro, isto é, guarda o endereço de um inteiro.

* 1. **DECLARAÇÃO E UTILIZAÇÃO**

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. Utilizar o ponteiro nestas circunstâncias 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 utilizar o operador **&**. Veja o exemplo:

                int cont=10;

                int \*pt;

                pt=&cont;

Criamos uma variável do tipo inteiro, chamada **cont** e inicializamos com o valor 10. Criamos um ponteiro para um inteiro chamado **pt**. A expressão **&cont** nos dá o endereço de cont, e esse é o endereço que armazenamos em **pt**.

Note que *não* alteramos o valor de **cont**, 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 cont usando pt. Para tanto vamos utilizar o operador "inverso" do operador &. É o operador \*. No exemplo acima, uma vez que fizemos pt=&cont a expressão \*pt é equivalente ao próprio cont. Isto significa que, se quisermos mudar o valor de cont para 12, basta fazer \*pt=12.

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

Uma observação importante: apesar de o 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 estão dois exemplos de usos simples de ponteiros:

// EXEMPLO 1

#include <stdlib.h>

#include <stdio.h>

#include <locale.h>

int main ()

{

int num, valor; // variáveis para números inteiros

int \*p; // p é um ponteiro para inteiro

setlocale(LC\_ALL, "Portuguese");

num=55;

**p=&num**; // p recebe o endereço de num

**valor=\*p**; // valor recebe o conteúdo do endereço apontado por p

printf ("\n\nConteúdo da variável valor = %d\n",valor);

printf("Endereço de num %X\n",&num); // **%X** mostra números hexa

printf ("Endereço para onde o ponteiro aponta: **%p** \n",p); //%p mostra conteúdo do ponteiro

printf ("Conteúdo de num apontado por p**: %d** \n",**\*p**);

system("PAUSE");

return(0);

}

// EXEMPLO 2

#include <stdlib.h>

#include <stdio.h>

#include <locale.h>

int main ()

{

int num,\*p;

setlocale(LC\_ALL, "Portuguese");

**num=22**;

**p=&num**; // p recebe o endereço de num

printf ("\nValor inicial: %d \n",num);

**\*p=100**; // faz mudar o valor de num de uma maneira indireta

// o conteúdo do endereço apontado por p é modificado para 100

printf ("\nValor parcial: %d\n",num);

printf("\nInforme um outro valor:");

scanf("%d", p); // pode ser &num ou então p

printf ("\nValor final: %d\n",num);

system("PAUSE");

return(0);

}

Nos dois exemplos vemos o funcionamento básico dos ponteiros. Note que primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve imprimir um endereço de memória.

**IMPORTANTE!**

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**. Note 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**, ou seja, o conteúdo do endereço apontado por **p1** passa a ser igual ao conteúdo do endereço apontado por **p2**.

Basicamente, depois que se aprende a utilizar 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 : 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 utilizar o conteúdo do ponteiro 15 posições adiante:

\*(p+15);

Outra operação é a comparação entre dois ponteiros. 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](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) ou [**double**s](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c320.html) de ponteiros.

Vamos ver esse exemplo e comparar os resultados.

#include <stdio.h>

int main(){

int x=10;

int \*p; // define um ponteiro para um número inteiro

p = &x; // p aponta para o endereço de x

printf("%d\n", \*p); // mostra o valor de x, no endereço apontado por p

(\*p)++; // incrementa o valor do endereço apontado por p

printf("%d\n", \*p); // mostra o valor de x, no endereço apontado por p

\*p++; // incrementa o endereço apontado por p - p vai apontar para o próximo endereço

printf("%d\n", \*p); // mostra o valor existente no endereço apontado por p

return 0;

}

* 1. **PONTEIROS E VETORES**

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

#### - Vetores como ponteiros

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

 Quando você declara uma matriz da seguinte forma:

tipo\_da\_variável nome\_da\_variável [tam1][tam2];

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

tam1 x tam2 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. Pois, 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 utilizar 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 utilizar um ponteiro, que vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

int main ()

{

float matriz [50][50];

int i,j;

for (i=0;i<50;i++)

        for (j=0;j<50;j++)

                matriz[i][j]=0.0;

return(0);

}

Podemos reescrevê-lo usando ponteiros:

int main ()

{

float matriz [50][50];

float \*p;

int cont;

p=matriz[0];

for (cont=0;cont<2500;cont++)

         {

        \*p=0.0;

        p++;

        }

return(0);

}

No primeiro programa, *cada* vez que se faz **matriz[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 operações a seguir são invalidas \*/

   vetor = vetor + 2;     /\* ERRADO: vetor não variável \*/

   vetor++;               /\* ERRADO: vetor não variável \*/

   vetor = ponteiro;      /\* ERRADO: vetor não variável \*/

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, como o DevC++, dirão que tem-se "*incompatible types in assignment*", tipos incompatíveis em uma atribuição.

/\* já as operações abaixo são validas \*/

     ponteiro = vetor;      /\* CERTO: ponteiro é variável \*/

     ponteiro = vetor+2;    /\* CERTO: ponteiro é variável \*/

**- 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 <stdlib.h>

#include <stdio.h>

int main ()

{

int matriz [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int \*p;

p=matriz;

printf ("O terceiro elemento do vetor e: %d\n",p[2]);

system("PAUSE");

return(0);

}

Podemos ver que **p[2]** equivale a **\*(p+2)** e assim, o programa poderia também ser escrito da seguinte forma:

#include <stdlib.h>

#include <stdio.h>

int main ()

{

int matriz [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int \*p;

p=matriz;

printf ("O terceiro elemento do vetor e: %d\n",\*(p+2));

system("PAUSE");

return(0);

}

#### - 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()**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c520.html#c522.html) da biblioteca:

#include <stdio.h>

#include <stdlib.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\n",str3,str2);

system("PAUSE");

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()**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c520.html#c522.html) 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++** você poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que se recomenda que se devesse fazer. O que você 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 \*pmatriz [10];

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

* 1. **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 utilizar [**strcpy()**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c520.html#c522.html) 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()**](../../Sistemas%20de%20Informação/Linguagem%20C/Programacao%20C/Tutorial%20UFMG/aulas/c520.html#c522.html).

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

char \*str1="String constante.";

Aí poderíamos, em todo lugar que precisarmos da string, utilizar a variável **str1**. Devemos apenas tomar cuidado ao utilizar este ponteiro. Se o alterarmos vamos perder a string. Se o utilizarmos 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.

* 1. **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.

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>

#include <stdlib.h>

int main()

{

float fpi = 3.1415, \*pf, \*\*ppf;

pf = &fpi; /\* pf armazena o endereço de fpi \*/

ppf = &pf; /\* ppf armazena o endereço de pf \*/

printf("%f\n", \*\*ppf); /\* Imprime o valor de fpi \*/

printf("%f\n", \*pf); /\* Também imprime o valor de fpi \*/

system("PAUSE");

return(0);

}

* 1. **CUIDADOS COM PONTEIROS**

O principal cuidado ao se utilizar 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*** utilizar 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).

**EXERCÍCIOS**

10.1 Qual o valor de y no final do programa? A seguir, escreva um //comentário em cada comando de atribuição explicando o que ele faz e o valor da variável após sua execução.

#include <stdlib.h>

#include <stdio.h>

int main () {

int y, \*p, x;

y = 0;

p = &y;

x = \*p;

x = 4;

(\*p)++;

x--;

(\*p) += x; //

printf ("y = %d\n", y);

system("PAUSE");

return(0); }

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

#include <stdio.h>

#include <stdlib.h>

int main()

{

int x, \*p, \*\*q;

p = &x;

q = &p;

x = 10;

printf("\n%d\n", &q);

system("PAUSE");

return(0);

}

10.3 Escreva um programa que declare uma matriz 10x10 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 100, também usando ponteiros.