

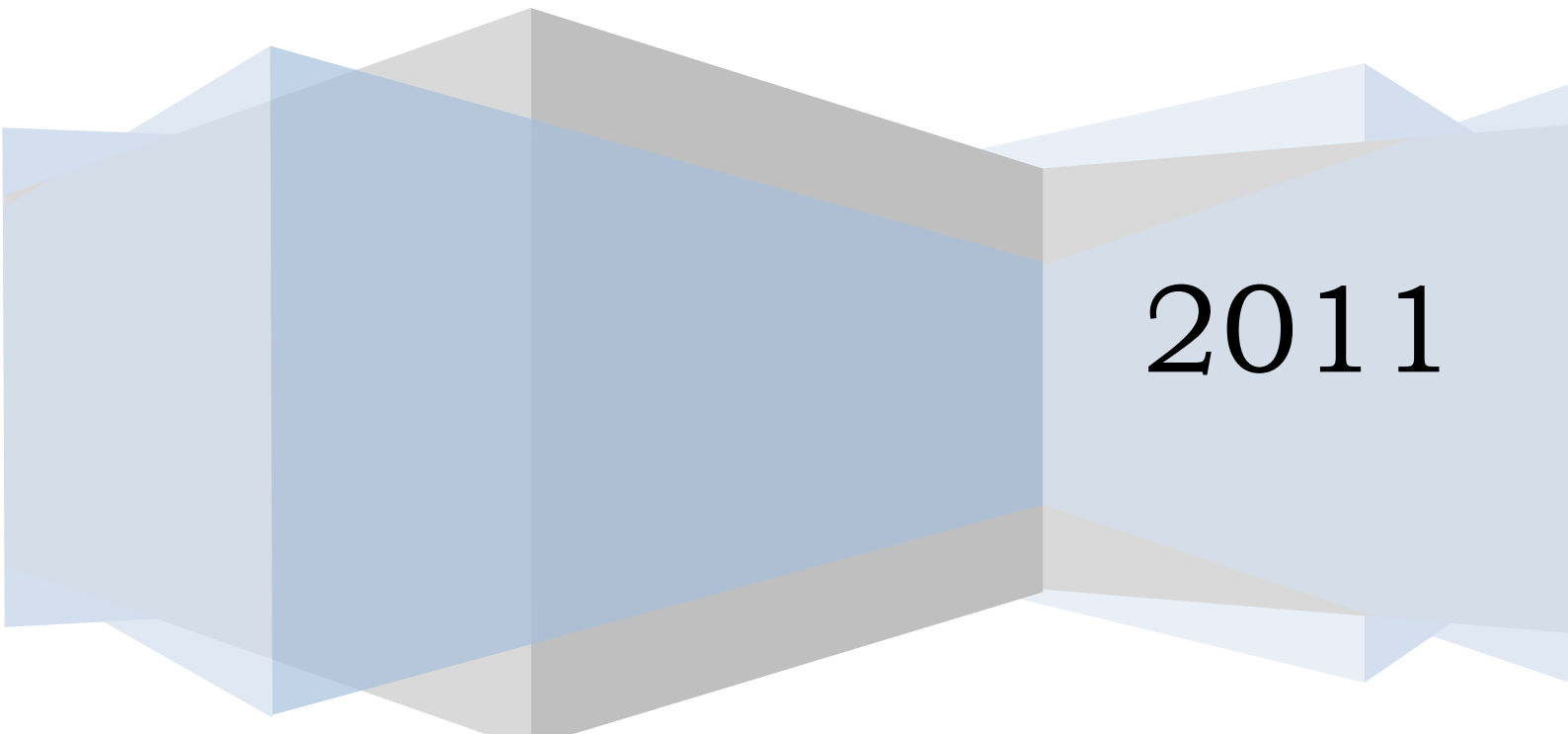
Flávio Augusto de Freitas

Introdução à Programação em Linguagem C/C++

<http://flavioaf.blogspot.com>

C/C++

Tutorial 17 (usando Dev-C++ versão 4.9.9.2)



2011

1 INTRODUÇÃO

Esta série de tutoriais foi escrita usando o **Microsoft Windows 7 Ultimate** e o **Bloodshed Dev-C++** versão 4.9.9.2, que pode ser baixada em <http://www.bloodshed.net>. Se alguém quiser adquirir mais conhecimentos e quiser aprofundar no assunto, visite <http://www.cplusplus.com/>.

Qualquer **exemplo de programa** nesta série de tutoriais implica **em um novo projeto**, dentro de uma **nova pasta**. Se não souber como criar uma nova pasta, volte e refaça o tutorial 1.

2 PONTEIROS

A compreensão e o uso correto de ponteiros são críticos para criação de muitos programas de êxito na linguagem C. Há três motivos para isso. Primeiro, os ponteiros proveem o meio para que as funções possam modificar os seus argumentos de chamada. Segundo, eles são usados para suportar rotinas de alocação dinâmica da linguagem C. Terceiro, podem ser substituídos pelas matrizes em muitas situações - proporcionando aumento de eficiência. Além disso, muitas das características do C apoiam-se firmemente nos ponteiros, portanto, um completo entendimento de ponteiros é muito importante. Além de ser uma das características mais fortes da linguagem C, os ponteiros também são muito perigosos. Por exemplo, quando não-inicializados ou descuidados, eles podem causar o travamento do sistema. E pior: é fácil usar ponteiros incorretamente, o que causa "bugs" (erros) difíceis de serem encontrados.

2.1 PONTEIROS SÃO ENDEREÇOS

Um ponteiro é uma variável que guarda um endereço de memória de outro objeto. Mais comumente, esse endereço é a localização de outra variável na memória, embora ele possa ser o endereço de uma porta ou de propósito-especial na RAM, como uma área auxiliar (buffer). Se uma variável contém o endereço de outra variável, a primeira é conhecida como um ponteiro para a segunda. Essa situação é ilustrada abaixo:

Seja o seguinte código:

```
float mult= 315.2;
int sum = 32000;
char letter = 'A'; /* equivalente a char letter = 65 */
int *set; /* declara o ponteiro set */
float *ptr1;
set=&sum; /* inicializa pointer set */
ptr1=&mult; /* inicializa pointer ptr1 */
```

O código anterior gera o seguinte mapa de memória:

Endereço de memória: (hexadecimal)	Valor da variável na memória (decimal exceto indicação contrária):	
FA10		
FA11	3.152 E 02	→ mult (4 bytes)
FA12		
FA13		
FA14	32000	→ sum (2 bytes) □
FA15		
FA16	65	→ letter (1 byte) □ Valor ASCII do caractere 'A'
FA17	FA14 (hex)	→ set (2 bytes) □ set aponta para sum
FA18		
FA19	FA10 (hex)	→ ptr1 (2 bytes) □ ptr1 aponta para mult
FA1A		

2.2. VARIÁVEL PONTEIRO:

Se uma variável vai armazenar um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste em um tipo base, um * e o nome da variável. A forma geral para declaração de uma variável ponteiro é:

```
tipo * nome-da-variável;
```

onde o tipo pode ser qualquer tipo válido da linguagem C e o nome-da-variável é o nome da variável ponteiro. O tipo base do ponteiro define qual tipo de variáveis o ponteiro pode apontar. Por exemplo, estas declarações criam um ponteiro caractere e dois ponteiros inteiros.

```
char *p;

int *temp, *início;
```

2.3 OS OPERADORES DE PONTEIROS:

Existem dois operadores especiais de ponteiro: o & e o *. O & é um operador unário que retorna o endereço de memória do seu operando. (Um operador unário requer somente um operando.) Por exemplo,

```
count_addr = &count;
```

coloca em `count_addr` o endereço de memória da variável `count`. Esse endereço é a localização interna da variável no computador. Ele não faz nada com o valor de `count`. A operação do operador `&` pode ser lembrada como "o retorno de endereço da variável que a ele sucede". Portanto, a atribuição dada pode ser determinada como "a variável `count_addr` recebe o endereço de variável `count`".

Para entender melhor essa atribuição, assumo que a variável `count` esteja localizada no endereço 2000. Então, após a atribuição, `count_addr` terá o valor de 2000.

O segundo operador é o `*`, e é o complemento do operador `&`. Ele é um operador unário que retorna o valor da variável localizada no endereço que o segue. Por exemplo, se `count_addr` contém o endereço de memória da variável `count`, então

```
val = *count_addr;
```

colocará o valor de `count` em `val`. Se `count` originalmente tem o valor 100, então `val` terá o valor de 100, porque este é o valor armazenado na localização 2000, que é o endereço de memória que foi atribuído a `count_addr`. A operação do operador `*` pode ser lembrada como "valor contido no endereço". Neste caso, então, a declaração pode ser lida como "val recebe o valor que está no endereço `count_addr`".

Com relação ao mapeamento de memória anterior, envolvendo as variáveis `mult`, `sum`, `letter` e os ponteiros `set` e `ptr1`, se tivéssemos feito a declaração de outra variável inteira, digamos, de nome `teste`, e fizéssemos a seguinte declaração:

```
teste= *set; /* teste recebe o valor da variável p/
a qual set aponta */
```

`teste` receberia o valor de `sum` que é 32000.

De mesma maneira se `teste` fosse declarada como `float` e fizéssemos

```
teste = *ptr1;
```

`teste` receberia o valor de `mult` que é 315.2.

Infelizmente, o sinal de multiplicação e o sinal "valor no endereço" são os mesmos. Isso algumas vezes confunde iniciantes na linguagem C. Esses operadores não se relacionam um com o outro. Tanto `&` como `*` têm precedência maior que todos os outros operadores aritméticos, exceto o menos unário, com o qual eles se igualam.

Aqui está um programa que usa esses operadores para imprimir o número 100 na tela:

```
/* imprime 100 na tela */
#include <stdio.h>
main()
{
    int *count_addr, count, val;
    count = 100;
    count_addr= & count; /* pega o endereço de count */
    val=*count_addr; /*pega o valor que está no endereço
count_addr*/
    printf("%d", val); /*exibe 100*/
}
```

Tipo Base:

Como o compilador sabe quantos bytes copiar em `val` do endereço apontado por `count_addr`? Mais genericamente, como o compilador transfere o número adequado de bytes em qualquer atribuição usando um ponteiro? A resposta é que o tipo base do ponteiro determina o tipo de dado que o compilador assume que o ponteiro está apontando. Nesse caso, uma vez que `count_addr` é um ponteiro inteiro, 2 bytes de informação são copiados em `val` a partir do endereço apontado por `count_addr`. Se ele fosse um ponteiro `double`, então 8 bytes teriam sido copiados.

```
#include <stdio.h>
/* Este programa não funciona adequadamente */
main()
{
    float x=10.1, y;
    int *p;
    p=&x; /* observar warning emitido pelo compilador */
    y=*p;
    printf("%f",y);
}
```

Este programa não atribuirá o valor da variável `x` para `y`. Tendo em vista que `p` é declarado para ser um ponteiro para um inteiro, somente 2 bytes de informação serão transferidos para `y`, não os 4 bytes que normalmente formam um número de ponto flutuante.

2.4 EXPRESSÕES COM PONTEIROS:

2.4.1 ATRIBUIÇÃO DE PONTEIROS:

Como qualquer variável, um ponteiro pode ser usado no lado direito de uma declaração para atribuir seu valor para outro ponteiro. Por exemplo:

```
#include<stdio.h>
main()
{
    int x;
    int *p1, *p2;
    x=101;
    p1=&x;
    p2=p1;
    /* imprime o valor hexadecimal do endereço */
    /* de x e não o valor de x */
    printf("Na localização %p ", p2);

    /* agora imprime o valor de x */
    printf("está o valor %d\n", *p2);
}
```

O endereço, em hexadecimal, de x é exibido usando-se outro dos códigos de formatação da função printf(). O %p especifica que um endereço de ponteiro é para ser exibido usando-se a notação hexadecimal.

2.4.2 ARITMÉTICA COM PONTEIROS:

Somente duas operações aritméticas podem ser usadas com ponteiros: adição e subtração. Para entender o que ocorre na aritmética com ponteiros, consideremos que p1 seja um ponteiro para um inteiro com o valor atual 2000.

Depois da expressão

```
p1++;
```

o conteúdo de p1 será 2002, não 2001. Cada vez que é incrementado, p1 apontará para o próximo inteiro. O mesmo é válido para decrementos. Por exemplo,

```
p1--;
```

fará com que a variável p1 tenha o valor 1998, assumindo-se que ela tinha anteriormente o valor 2000.

Cada vez que é incrementado, um ponteiro apontará para a próxima localização de memória do seu tipo base. Cada vez que é decrementado, apontará para a localização do elemento anterior. No caso de ponteiros para caracteres, parecerá ser uma aritmética normal. Porém, todos os outros ponteiros incrementarão ou decrementarão pelo tamanho do tipo dado para o qual apontam. Por exemplo, assumindo-se caracteres de 1 byte e inteiros de 2 bytes, quando um ponteiro caractere é incrementado, seu valor é incrementado de um; entretanto, quando um ponteiro inteiro é incrementado, seu valor é incrementado de dois bytes. Um ponteiro float seria incrementado de 4 bytes.

Exemplo:

```
char *chptr;
int *iptr;
float *flptr;
chptr=0x3000;
iptr=0x6000;
flptr=0x9000;
chptr++; /* ch=0x3001 */
iptr++; /* i=0x6002 */
flptr++; /* flptr=0x9004 */
```

Porém, você não está limitado a somente incrementar e decrementar. Pode-se adicionar ou subtrair inteiros para ou de ponteiros. A expressão:

```
p1=p1+9;
```

fará p1 apontar para o nono elemento do tipo base de p1 considerando-se que ele está apontando para o primeiro.

Exemplo:

```
double *p1;
p1=0x1000; /* p1 = 4096 */
p1=p1+9 /* p1 = 0x1048 = 4096 + 9*8 */
```

2.4.6 COMPARAÇÃO COM PONTEIROS:

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros p e q, a seguinte declaração é perfeitamente válida:

```
if(p<q) printf("p aponta para um endereço de memória mais baixo que q\n");
```

Geralmente, comparações entre ponteiros devem ser usadas somente quando dois ou mais ponteiros estão apontando para um objeto em comum.

2.5 PONTEIROS E MATRIZES:

Há uma relação entre ponteiros e matrizes. Considere este fragmento de código:

```
char str[80], *p1;
p1 = str;
```

Aqui, p1 foi associado ao endereço do primeiro elemento da matriz em str. Na linguagem C, um nome de matriz sem um índice é o endereço para o começo da matriz (em geral um ponteiro contém o endereço do início de uma área de armazenamento ou transferência de dados). O mesmo resultado - um ponteiro para o primeiro elemento da matriz str - pode ser gerado com a declaração abaixo:

```
p1=&str[0];
```

Entretanto, ela é considerada uma forma pobre por muitos programadores em C. Se você deseja acessar o quinto elemento em `str`, pode escrever:

```
str[4]
```

ou

```
*(p1+4)
```

As duas declarações retornarão o quinto elemento.

LEMBRE-SE: *Matrizes começam em índice zero, então um 4 é usado para indexar `str`. Você também pode adicionar 4 ao ponteiro `p1` para obter o quinto elemento, uma vez que `p1` aponta atualmente para o primeiro elemento de `str`.*

A linguagem C permite essencialmente dois métodos para acessar os elementos de uma matriz. Isso é importante porque a aritmética de ponteiros pode ser mais rápida do que a indexação de matriz. Uma vez que a velocidade é frequentemente importante em programação, o uso de ponteiros para acessar elementos da matriz é muito comum em programas em C.

Para ver um exemplo de como ponteiros podem ser usados no lugar de indexação de matriz, considere estes dois programas - um com indexação de matriz e um com ponteiros - , que exibem o conteúdo de uma string em letras minúsculas.

```
#include <stdio.h>
#include <ctype.h>
/* versão matriz */
main()
{
    char str[80];
    int i;
    printf("digite uma string em letra maiúscula: ");
    gets(str);
    printf("aqui está a string em letra minúscula: ");
    for(i=0; str[i]; i++) printf("%c", tolower(str[i]));
}
#include <stdio.h>
#include <ctype.h>
/* versão ponteiro */
main()
{
    char str[80], *p;
    printf("digite uma string em letra maiúscula: ");
    gets(str);
    printf("aqui está a string em letra minúscula: ");
    p=str; /* obtém o endereço de str */
    while (*p) printf("%c", tolower(*p++));
}
```

A versão matriz é mais lenta que a versão ponteiro porque é mais demorado indexar uma matriz do que usar o operador `*`.

Algumas vezes, programadores iniciantes na linguagem C cometem erros pensando que nunca devem usar a indexação de matrizes, já que ponteiros são muito mais eficientes. Mas não é o caso. Se a matriz será acessada em ordem estritamente ascendente ou descendente, ponteiros são mais rápidos e fáceis de usar. Entretanto, se a matriz será acessada randomicamente, a indexação da matriz pode ser uma melhor opção, pois geralmente será tão rápida quanto a avaliação de uma expressão complexa de ponteiros, além de ser mais fácil de codificar e entender.

2.6 INDEXANDO UM PONTEIRO:

Em C, é possível indexar um ponteiro como se ele fosse uma matriz. Isso estreita ainda mais o relacionamento entre ponteiros e matrizes. Por exemplo, este fragmento de código é perfeitamente válido e imprime os números de 1 até 5 na tela:

```
/* Indexando um ponteiro semelhante a uma matriz */
#include <stdio.h>
main()
{
    int i[5]={1, 2, 3, 4, 5};
    int *p,t;
    p=i;
    for(t=0; t<5; t++) printf("%d ",*(p+t));
}
```

Em C, a declaração `p[t]` é idêntica a `*(p+t)`.

3 PONTEIROS E STRINGS:

Uma vez que o nome de uma matriz sem um índice é um ponteiro para o primeiro elemento da matriz, o que está realmente acontecendo quando você usa as funções de string discutidas nos capítulos anteriores é que somente um ponteiro para strings é passado para função, e não a própria string. Para ver como isso funciona, aqui está uma maneira como a função `strlen()` pode ser escrita:

```
strlen(char *s)
{
    int i=0;
    while(*s){
        i++;
        s++;
    }
    return i;
}
```

Quando uma constante string é usada em qualquer tipo de expressão, ela é tratada como um ponteiro para o primeiro caractere na string. Por exemplo, este programa é perfeitamente válido e imprime a frase "este programa funciona" na tela:

```
#include <stdio.h>
main()
{
    char *s;
    s= "este programa funciona";
    printf(s)
}
```

4 OBTENDO O ENDEREÇO DE UM ELEMENTO DA MATRIZ:

É possível atribuir o endereço de um elemento específico de uma matriz aplicando-se o operador & para uma matriz indexada. Por exemplo, este fragmento coloca o endereço do terceiro elemento de x em p:

```
p= &x[2];
```

Essa declaração é especialmente útil para encontrar uma substring. Por exemplo, este programa imprimirá o restante de uma string, que é inserida pelo teclado, a partir do ponto onde o primeiro espaço é encontrado:

```
#include <stdio.h>
/* Exibe a string à direita depois que o primeiro espaço
é encontrado.*/
main()
{
    char s[80];
    char *p;
    int i;
    printf("digite uma string: ");
    gets(s);
    /* encontra o primeiro espaço ou o fim da string */
    for (i=0; s[i] && s[i]!=' '; i++);
    p=&s[i];
    printf(p);
}
```

Esse programa funciona, uma vez que p estará apontando para um espaço ou para um nulo (se a string não contiver espaços). Se há um espaço, o restante da string será impresso. Se há um nulo, a função printf() não imprime nada. Por exemplo, se "Olá a todos" for digitado, "a todos" será exibido.

5 MATRIZES DE PONTEIROS:

Podem existir matrizes de ponteiros como acontece com qualquer outro tipo de dado. A declaração

para uma matriz de ponteiros, do tipo int, de tamanho 10 é:

```
int *x[10];
```

Para atribuir o endereço de uma variável ponteiro chamada var ao terceiro elemento da matriz de ponteiros, você deve escrever:

```
x[2] = &var;
```

Para encontrar o valor de var, você deve escrever:

```
*x[2];
```

Um uso bastante comum de matrizes de ponteiros é armazenar ponteiros para mensagens de erros. Você pode criar uma função que emite uma mensagem, dado o seu número de código, como mostrado na função serror() abaixo:

```
serror(int num)
{
    char *err[]={
        "não posso abrir o arquivo\n",
        "erro de leitura\n",
        "erro de escrita\n",
        "falha na mídia\n"
    };
    printf("%s",err[num]);
}
```

Como você pode ver, a função printf() é chamada dentro da função serror() com um ponteiro do tipo caractere, que aponta para uma das várias mensagens de erro indexadas pelo número do erro passado para a função. Por exemplo, se num é passado com valor 2, a mensagem de erro de escrita é exibida.

Outra aplicação interessante para matrizes de ponteiros de caracteres inicializadas usa a função de linguagem C system(), que permite ao seu programa enviar um comando ao sistema operacional. Uma chamada a system() tem esta forma geral:

```
system("comando");
```

onde comando é um comando do sistema operacional a ser executado, inclusive outro programa. Por exemplo, assumindo-se o ambiente DOS, esta declaração faz com que o diretório corrente seja exibido:

```
system("DIR");
```

O programa seguinte implementa uma pequena interface com o usuário dirigida por menu que pode executar quatro comandos do DOS: DIR, CHKDSK, TIME e DATE.


```

/* Uma interface simples do sistema do sistema opera-
cional com o usuário
dirigida por menu.*/
#include <stdio.h>
#include<stdlib.h>
#include <conio.h>
main()
{
/* cria uma matriz de strings */
char *comando[] = {
"DIR",
"CHKDSK",
"TIME",
"DATE"
};
char ch;
for(;;){
do {
printf("1: diretório\n");
printf("2: verifica o disco\n");
printf("3: atualiza a hora\n");
printf("4: atualiza a data\n");
printf("5: sair\n");
printf("\nopção: ");
ch=getche();
} while ((ch<'1') || (ch>'5'));
if(ch=='5')break; /* fim */
/* executa o comando específico */
printf("\n");
system(comando[ch-'1']);
}
}

```

Qualquer comando do DOS, inclusive a chamada a outros programas pode ser implementada desta maneira.

Simples Tradutor Inglês-Português:

A primeira coisa que você precisa é a tabela inglês-português mostrada aqui, entretanto, você é livre para expandi-la se assim o desejar.

```

char trans[][20] = {
"is", "é",
"this", "isto",
"not", "nao",
"a", "um",
"book", "livro",
"apple", "maça",
"I", "eu",
"bread", "pao",
"drive", "dirigir",
"to", "para",
"buy", "comprar",
"", ""
}

```

Cada palavra em inglês é colocada em par com a equivalente em português. Note que a palavra mais longa não excede 19 caracteres.

A função main() do programa de tradução é mostrada aqui junto com as variáveis globais necessárias:

```

char entrada[80];
char palavra[80];
char *p;
main()
{
int loc;
printf("Informe a sentença em inglês; ");
gets(entrada);
p = entrada; /* dá a p o endereço da matriz entrada */
printf(tradução rústica para o português: ");
pega_palavra(); /* pega a primeira palavra */
/* Este é o laço principal. Ele lê uma palavra por vez
da matriz entrada e
traduz para o português.*/
do {
/* procura o índice da palavra em inglês em trans */
loc = verifica(palavra);
/* imprime a palavra em português se uma corres-
pondência é encontrada*/
if(loc!=-1) printf("%s ", trans[loc+1]);
else printf("<desconhecida> ");
pega_palavra(); /* pega a próxima palavra /
} while(*palavra); /*repete até que uma string nula é
encontrada */
}

```

O programa opera desta maneira: primeiro, o usuário é solicitado a informar uma sentença em inglês. Essa sentença é lida na string entrada. O ponteiro p é, então, associado ao endereço do início de entrada e é usado pela função pega_palavra() para ler uma palavra por vez da string entrada e colocá-la na matriz palavra. O laço principal verifica, então, cada palavra, usando a função verifica(), e retorna o índice da palavra em inglês ou -1 se a palavra não estiver na tabela. Adicionando-se um a este índice, a palavra correspondente em português é encontrada.

A função verifica() é mostrada aqui:

```

/* Esta função retorna a localização de uma corres-
pondência entre a string apontada pelo parâmetro s e
a matriz trans.*/
verifica(char *s)
{
int i;
for(i=0; *trans[i]; i++)
if(!strcmp(trans[i], s)) break;
if(*trans[i]) return i;
else return -1;
}

```

Você chama verifica() com um ponteiro para a palavra em inglês e a função retorna o seu índice se ela estiver na tabela e -1 se não for encontra-
da.

A função `pega_palavra()` é mostrada a seguir. Da maneira como a função foi concebida, as palavras são delimitadas somente por espaços ou por terminador nulo.

```
/* Esta função lerá a próxima palavra da matriz entrada. Cada palavra é considerada como sendo separada por um espaço ou pelo terminador nulo. Nenhuma outra pontuação é permitida. A palavra retornada será uma string de tamanho nulo quando o final da string entrada é encontrado. */
pega_palavra()
{
    char *q;
    /* recarrega o endereço da palavra toda vez que a função é chamada */
    q = palavra;
    /* pega a próxima palavra */
    while(*p && *p!=' ') {
        *q = *p;
        p++;
        q++;
    }
    if(*p==' ') p++;
    *q = '\0'; /* termina cada palavra com um terminador nulo */
}
```

Assim que retorna da função `pega_palavra()` a variável global conterá ou a próxima palavra em inglês na sentença ou um nulo.

O programa completo da tradução é mostrado aqui:

```
/* Um tradutor (bastante) simples de inglês para português. */
/* Sugestão: Colocar Watches em: entrada<->p , palavra<->q , loc */
#include<stdio.h>
#include<string.h>
char trans[][20] = {
    "is", "é",
    "this", "isto",
    "not", "nao",
    "a", "um",
    "book", "livro",
    "apple", "maça",
    "I", "eu",
    "bread", "pao",
    "drive", "dirigir",
    "to", "para",
    "buy", "comprar",
    "", ""
};
char entrada[80];
char palavra[80];
char *p;
main()
{
    int loc;
    printf("Informe a sentença em inglês: ");
```

```
gets(entrada);
p = entrada; /* dá a p o endereço da matriz entrada */
printf("tradução rústica para o português: ");
pega_palavra(); /* pega a primeira palavra */
/* Este é o laço principal. Ele lê uma palavra por vez da matriz entrada e traduz para o português. */
do {
    /* procura o índice da palavra em inglês em trans */
    loc = verifica(palavra);
    /* imprime a palavra em português se uma correspondência é encontrada */
    if(loc!=-1) printf("%s ", trans[loc+1]);
    else printf("<desconhecida> ");
    pega_palavra(); /* pega a próxima palavra */
} while(*palavra); /* repete até que uma string nula é encontrada */
}
/* Esta função retorna a localização de uma correspondência entre a string apontada pelo parâmetro s e a matriz trans. */
verifica(char *s)
{
    int i;
    for(i=0; *trans[i]; i++) if(!strcmp(trans[i], s)) break;
    if(*trans[i]) return i; /* se não é o fim da matriz trans retorna i */
    else return -1; /* se é, retorna -1 (desconhecida) */
}
/* Esta função lerá a próxima palavra da matriz entrada. Cada palavra é considerada como sendo separada por um espaço ou pelo terminador nulo. Nenhuma outra pontuação é permitida. A palavra retornada será uma string de tamanho nulo quando o final da string entrada é encontrado. */
pega_palavra()
{
    char *q;
    /* recarrega o endereço da palavra toda vez que a função é chamada */
    q = palavra; /* palavra é global */
    /* pega a próxima palavra */
    while(*p && *p!=' ') { /* p é global */
        *q = *p;
        p++;
        q++;
    }
    if(*p==' ') p++;
    *q = '\0'; /* termina cada palavra com um terminador nulo */
}
```

6 PONTEIROS PARA PONTEIROS:

Indireção simples:

Ponteiro		Variável
endereço	→	valor

Indireção múltipla:

Ponteiro		Ponteiro		Variável
endereço	→	endereço	→	valor

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um asterisco adicional na frente do seu nome. Por exemplo, esta declaração diz ao compilador que `novobalanco` é um ponteiro para um ponteiro do tipo `float`:

```
float **novobalanco;
```

É importante entender que `novobalanco` não é um ponteiro para um número de ponto flutuante, mas, antes, um ponteiro para um ponteiro `float`.

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

```
#include <stdio.h>
main()
{
    int x, *p, **q;
    x = 10;
    p = &x;
    q = &p;
    printf("%d", **q); /* imprime o valor de x */
}
```

Aqui, `p` é declarado como um ponteiro para um inteiro e `q` como um ponteiro para um ponteiro para um inteiro. A chamada a `printf()` imprimirá o número 10 na tela.

7 INICIALIZANDO PONTEIROS:

Depois que é declarado, mas antes que tenha sido associado a um valor, um ponteiro conterá um valor desconhecido. Se tentarmos usar um ponteiro antes de dar a ele um valor, provavelmente não só travará o programa como também o sistema operacional - um tipo de erro muito desagradável!

Por convenção, um ponteiro que está apontando para nenhum lugar deve ter valor nulo. Entretanto, somente o fato de um ponteiro ter um valor nulo não o torna "seguro". Se usarmos um ponteiro nulo no lado esquerdo de uma declaração de atribuição, correremos o risco de travar o programa e o sistema operacional.

Como um ponteiro nulo é considerado inútil, podemos usá-lo para tornar muitas rotinas de ponteiros fáceis de codificar e mais eficientes. Por exemplo, podemos usar um ponteiro nulo para marcar o final de uma matriz de ponteiros. Se isso é feito, a rotina que acessa aquela matriz saberá que chegou ao fim quando um valor nulo

é encontrado. Esse tipo de abordagem é ilustrado pelo laço for mostrado aqui:

```
/* Observe um nome assumindo o último elemento de
p como um nulo.*/
for(t=0; p[t]; ++t)
    if(!strcmp(p[t], nome)) break;
```

O laço será executado até que seja encontrada uma correspondência ou um ponteiro nulo. Uma vez que o fim da matriz está marcado com um nulo, a condição que controla o laço falhará quando ele for encontrado.

É uma prática muito comum em programas profissionais escritos na linguagem C inicializar strings. Vimos dois exemplos anteriores neste capítulo na seção de matrizes de ponteiros. Outra variação desse tema é o seguinte tipo de declaração de string:

```
char *p="Alô mundo\n";
```

Esse tipo de declaração coloca o endereço da string "Alô mundo" no ponteiro `p`. Por exemplo, o seguinte programa é perfeitamente válido:

```
#include <stdio.h>
#include <string.h>
char *p="Alô mundo";
main()
{
    register int i;
    /* imprime a string normal e inversa*/
    printf(p);
    for(t=strlen(p)-1; t>-1; t--) printf("%c",p[t]);
}
```

8 PROBLEMAS COM PONTEIROS:

Um problema com um ponteiro é difícil de ser encontrado. O ponteiro por si só não é um problema; o problema é que, cada vez que for realizada uma operação usando-o, pode-se estar lendo ou escrevendo para algum lugar desconhecido da memória. Se você ler, o pior que lhe pode acontecer é ler informação inválida, "lixo". Entretanto, se você escrever para o ponteiro, estará escrevendo sobre outros pedaços de código ou dados. Isso pode não ser mostrado mais tarde na execução do programa, levando-o a procurar pelo erro em lugar errado. Pode ser pouco ou não certo sugerir que o ponteiro é um problema. Esse tipo de erro tem feito com que programadores percam muitas noites de sono.

Tendo em vista que erros com ponteiros podem se transformar em pesadelos, você deve fazer o máximo para nunca gerar um! Alguns dos erros

mais comuns são discutidos aqui, começando com o exemplo clássico de erro com ponteiro: o ponteiro-não-inicializado. Considere o seguinte:

```
/* Este programa está errado. Não o execute.*/
main()
{
    int x, *p;
    x = 10;
    *p = x;
}
```

Esse programa atribui o valor 10 a alguma localização desconhecida na memória. Ao ponteiro `p` nunca foi dado um valor; portanto ele contém um valor, "lixo". Embora o Turbo C emita uma mensagem de aviso neste exemplo, o mesmo tipo de problema surge quando um ponteiro está simplesmente apontado para um lugar indevido. Por exemplo, você pode acidentalmente atribuir um ponteiro para um endereço errado. Esse tipo de problema frequentemente passa despercebido quando o seu programa é muito pequeno, por causa da probabilidade de `p` conter um endereço "seguro" - um endereço que não esteja no seu código, na área de dados ou no sistema operacional. Entretanto, à medida que o seu programa cresce, a probabilidade de `p` apontar para algum lugar fundamental aumenta. Eventualmente, o seu programa para de funcionar. Para evitar esse tipo de problema, certifique-se sempre de que um ponteiro esteja apontado para alguma posição válida antes que seja usado.

Um segundo erro é causado pelo mal-entendido sobre como usar um ponteiro. Considere o seguinte:

```
#include <stdio.h>
/* Este programa está incorreto. Não o execute. */
main()
{
    int x, *p;
    x = 10;
    p = x;
    printf("%d", *p);
}
```

A chamada à função `printf()` não imprimirá o valor de `x`, que é 10, na tela. Imprimirá algum valor desconhecido. O motivo para isso é que a atribuição

```
p = x;
```

está errada. Aquela declaração atribui o valor 10 para o ponteiro `p`, que supostamente contém um endereço, não um valor. Para tornar o programa correto, você deve escrever:

```
p = &x;
```

Neste exemplo, o Turbo C avisará você sobre o erro no programa. Entretanto, nem todos os erros desse tipo geral podem ser verificados pelo compilador.

9 EXERCÍCIOS RESOLVIDOS

1. Escrever um programa para alocar dinamicamente um vetor de inteiros, preencher este vetor com valores lidos do teclado e, por último, escrever o vetor.

```
#include
#include
#include
#include

void main() {
    int tam, * pt_aux, * vet;
    int i;
    char aux [ 10];

    printf ( "Entre com o numero de elementos do vetor:
");
    gets ( aux);
    tam = atoi ( aux);
    vet = ( int *) malloc ( tam * sizeof ( int));
    pt_aux = vet;
    for ( i = 0; i < tam; i ++ ) {
        printf ( "Entre com o valor do elemento %d: ", i);
        gets ( aux);
        * pt_aux ++ = atoi ( aux);
    }
    printf ( "\n");

    pt_aux = vet;
    for ( i = 0; i < tam; i ++ ) {
        printf("O valor do elemento %d eh: %d\n", i,
*pt_aux++);
    }
    free ( vet);

    getch ();
}
```

2. Escrever um programa para imprimir, do último para o primeiro, cada um dos elementos do vetor `X = {1.0, 3.4, 5.7, 9.8, 3.3}`, através de um ponteiro.

3. Escrever um programa para ler uma frase qualquer do teclado e imprimir, esta mesma frase, um caractere por vez.

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

void main() {
    char frase [ 255], * aux;

    printf( "Entre com uma frase qualquer: ");
    gets ( frase);
    aux = frase;
    while ( * aux) printf ( "%c", * aux ++);
    printf ( "\n");
    getch ( );
}
```

10 EXERCÍCIOS PROPOSTOS

- Completar o programa do exercício resolvido 1 (sempre usando ponteiros) para informar ao usuário:
 - o maior valor no vetor;
 - o menor valor no vetor;
 - o valor médio armazenado no vetor;
 - e o número de valores negativos no vetor.
- Completar programa do exercício resolvido 3 (sempre usando ponteiros) para:
 - imprimir a frase ao contrário;
 - contar o número de espaços em branco na frase.
- Escrever um programa para imprimir as matrizes declaradas abaixo utilizando ponteiros.

```
#include <stdio.h>

main() {
    int vetor[5] = {0, 1, 2, 3, 4};
    int matriz[5][5] = {
        0, 0, 0, 0, 0,
        1, 1, 1, 1, 1,
        2, 2, 2, 2, 2,
        3, 3, 3, 3, 3,
        4, 4, 4, 4, 4
    }
    char string [ 10] = "BICAMPEAO";
```

- Escrever um programa para alocar dinamicamente uma área de memória para N double's, preencher esta área com valores lidos do teclado e, por último, proporcionar uma consulta aos dados gravados. A consulta deve receber um número, que corresponde a uma das posições da área alocada, e o valor double armazenado naquela posição deve ser apresentado ao usuário. O programa deve fazer uma consistência no número informado, correspondente à posição, e apresentar uma mensagem de erro se ele estiver aquém ou além dos limites da área de memória.
- Escrever um programa para alocar uma área de memória para armazenar 3 valores float e 3 caracteres, preencher esta área com dados provenientes do teclado e, por último, apresentá-los do último até o primeiro, isto é, de maneira inversa àquela do armazenamento.
- Escrever um programa que recebe um nome completo e, posteriormente, armazena-o numa área de memória alocada dinamicamente. O tamanho desta área deve ser exatamente o necessário para armazenar corretamente o nome informado. Por último, o nome armazenado na área alocada deve ser impresso (usar %s no printf).
- Escrever um programa que recebe o nome de uma disciplina do curso de Informática e um caractere avulso. O programa deve informar o endereço da primeira ocorrência deste caractere no nome da disciplina (usar %x no printf). Igualmente, deve ser dada uma notificação caso o caractere não seja encontrado.

11 TERMINAMOS

Terminamos por aqui. O que está esperando, saia do Dev-C++ e corra para pegar o próximo tutorial em <http://flavioaf.blogspot.com>. Siga o blog, assim você fica sabendo das novidades no momento em que forem publicadas. Seguindo o blog você se mantém sempre atualizado de qualquer lançamento novo.