TUTORIAL SOBRE PONTEIROS E ARRANJOS EM C

por Ted Jensen Versão 1.2
Setembro. 2003

Tradução e adaptação
Augusto Manzano
Abril, 2023

SUMÁRIO

[1] NT: Site não ativo, disponibilidade alternativa: *https://github.com/J-AugustoManzano/ptrtut14*.

PREFÁCIO

Este texto apresenta ponteiros para programadores iniciantes na linguagem de programação C. Ao longo de vários anos lendo e contribuindo para várias conferências sobre C, incluindo aquelas no FidoNet e UseNet, notei que grandes números de novatos em C parecem ter dificuldade em compreender os fundamentos de ponteiros. Portanto, assumi a tarefa de tentar explicá-los em linguagem simples com muitos exemplos.

A primeira versão deste documento foi distribuída na forma de domínio público, assim como este está sendo realizado. Foi adotada por Bob Stout, que o incluiu como um arquivo chamado PTR-HELP.TXT em sua coleção amplamente distribuída de SNIPPETS. Desde o lançamento original no ano de 1995, acrescentei uma quantidade significativa de conteúdo e fiz algumas correções menores no trabalho original.

Posteriormente, por volta de 1998, publiquei uma versão em HTML em meu site em:

http://pweb.netcom.com/~tjensen/ptr/cpoint.htm[2]

Após inúmeras solicitações, finalmente lancei uma versão em formato PDF que é idêntica àquela versão em HTML citada acima e que pode ser obtida no mesmo site da web[3].

Agradecimentos:

Há tantas pessoas que contribuíram involuntariamente para este trabalho devido as perguntas que foram feitas no FidoNet C Echo, ou no UseNet Newsgroup comp.lang.c, ou em várias outras conferências de outras redes, que seria impossível listá-las. Agradecimentos especiais vão para Bob Stout, que teve a gentileza de incluir a primeira versão deste material em seu arquivo SNIPPETS.

Sobre o autor:

Ted Jensen é engenheiro eletrônico aposentado tendo trabalhado como designer de hardware e gerente de designers de hardware no campo de gravação magnética. Programação tem sido seu hobby desde 1968, quando aprendeu a usar cartões perfurados para serem executados em um mainframe. (O mainframe tinha 64K de memória de núcleo magnético!).

Uso deste material:

[2] NT: No momento da tradução deste texto em abril/maio de 2023 o site do autor não se encontrava mais no ar, ficando sua indicação apenas como ilustração e respeito ao texto original.
[3] NT: Para acesso alternativo ao arquivo PDF use o link:
https://pdos.csail.mit.edu/6.828/2014/readings/pointers.pdf.

gostaria que, nessas circunstâncias, o ministrante dessa aula me enviasse nota em um dos endereços abaixo informando-me disso. Eu escrevi isso com a esperança de que seja útil para outros e, como não estou pedindo nenhuma remuneração financeira, a única maneira que eu conheço para saber que atingi, pelo menos parcialmente, esse objetivo é por meio do feedback daqueles que acham este material útil.

Aliás, não é necessário ser um instrutor ou professor para entrar em contato comigo. Eu apreciaria uma nota de qualquer pessoa que considere o material útil ou que tenha críticas construtivas a oferecer. Também estou disposto a responder perguntas enviadas por e-mail nos endereços mostrados abaixo.

Ted Jensen
Redwood City, Californiatjensen@ix.netcom.com July 1998

Sobre o tradutor:

Augusto Manzano tem formação em análise de sistemas, ciências econômicas e licenciatura em matemática. Iniciou sua jornada na área de computação em 1985 tornando-se desenvolvedor, professor e autor de diversas obras relacionadas a área. Atualmente é professor do Instituto Federal de São Paulo.

Pode ser contatado no e-mail: augusto.garcia@ifsp.edu.br.

Nota/advertência:

Entre a escrita deste artigo e sua tradução se passaram, aproximadamente, 20 anos, uma eternidade se considerada a área da computação. Alguns exemplos conceituais indicados originalmente pelo autor e outros elementos textuais necessitaram de atençãom e neste sentido algumas inserções como notas de rodapé foram implementadas neste texto. Tentou-se ao máximo manter o tom de originalidade do material. As notas inseridas estão grafadas com a rubrica NT (Nota do Tradutor). É pertinente salientar que alguns dos códigos apresentados tiveram pequenas modificações nos elementos de apresentação de dados em tela.

Uma ação de preservação do material foi colocada na plataforma Github discriminada como versões 1.3, 1.4 e 1.5 por Jay Flaherty: *https://github.com/jflaherty/ptrtut13*. Seguindo a mesma linha disponibilizo a versão 1.6 em ***https://github.com/J-AugustoManzano/ptrtut16*** como *fork* do trabalho produzido por Jay Flaherty.

INTRODUÇÃO

Se você deseja ser proficiente na escrita de código na linguagem de programação C[4], deve ter um conhecimento profundo e prático sobre uso de ponteiros. Infelizmente, ponteiros em C parecem representar um obstáculo para novatos, especialmente para aqueles que vêm de outras linguagens de computador como Fortran, Pascal ou BASIC.

Para ajudar os recém-chegados a entenderem o conceito de ponteiros, escrevi o material a seguir. Para obter o máximo benefício deste material, acredito ser importante que o usuário possa executar o código dos vários exemplos contidos no artigo. Portanto, tentei manter todo o código compatível com o padrão ANSI da linguagem, de modo que funcione a partir de qualquer compilador compatível com este padrão. Também tentei cuidadosamente dividir o código dentro do texto. Dessa forma, com a ajuda de um editor de texto padrão ASCII, você pode copiar um determinado bloco de código para um novo arquivo e compilá-lo em seu sistema. Recomendo que os leitores façam isso, pois isso ajudará na compreensão do material.

[4] NT: Considere o que aqui é apresentado como base também a linguagem C++.

CAPÍTULO 1: O que é ponteiro?

Um dos temas que iniciantes em C encontram dificuldade é o conceito de ponteiros. O objetivo deste tutorial é fornecer uma introdução aos ponteiros e seu uso para esses iniciantes.

Percebi que muitas vezes a principal razão pela qual iniciantes têm problemas com ponteiros é que eles têm um entendimento fraco ou mínimo sobre variáveis em C. Portanto, comecemos com uma discussão sobre variáveis em C de modo geral.

Uma variável em um programa é algo com um nome, cujo valor pode variar. A forma como o compilador[5] e o linker[6] lidam com isso é atribuindo um bloco específico de memória dentro do computador para armazenar o valor dessa variável. O tamanho desse bloco depende da faixa em que a variável é permitida variar. Por exemplo, nos PC's[7], o tamanho de uma variável inteira é de 2 bytes, e de um inteiro longo é de 4 bytes[8]. Em C, o tamanho de um tipo de variável, como um inteiro, não precisa ser o mesmo em todos os tipos de máquinas.

Quando declaramos uma variável, informamos ao compilador duas coisas: o tipo da variável e seu nome de identificação. Por exemplo, ao declaramos uma variável do tipo inteiro com o nome "**k**" escrevemos:

```
int k;
```

Ao ver o indicativo "**int**" desta declaração, o compilador aloca *2 bytes* de memória (em um PC[9]) para poder armazenar o valor do tipo inteiro. Ele também configura uma tabela de símbolos. Nessa tabela, adiciona-se o símbolo "**k**" e o endereço relativo de memória onde esses *2 bytes* foram reservados.

Assim, posteriormente, se escrevermos:

```
k = 2;
```

esperamos que, durante a execução deste comando em tempo de execução, o valor "**2**" seja definido na localização de memória reservada para armazenar o valor "**k**". Em C, nos referimos a uma variável "**k**" como sendo um "*objeto*"[10].

Em certo sentido, nessa definição há dois "valores" associados ao objeto "**k**". Um é o valor do inteiro armazenado lá ("**2**" no exemplo indicado) e o outro é o "*valor*" do local de memória, ou seja, o endereço da variável "**k**". Alguns textos se referem a esses dois valores com a nomenclatura ***rvalue*** (valor da direita, pronunciado "*are value*") e ***lvalue*** (valor da

[5] NT: Compilador: software que transforma o código-fonte escrito por um programador em um programa executável pelo computador.
[6] NT: Linker (ou ligador, em português): software que faz a ligação entre os diferentes módulos de um programa e cria um executável.
[7] NT: PCs = Personal Computers (Computadores Pessoais).
[8] NT: A referência aqui apontada considera PCs de 16 bits. Nos modelos de 32 bits os inteiros são de 4 bytes e os inteiros longos são de 8 bytes, assim como os modelos de 64 bits possuem inteiros de 8 bytes e inteiros longos de 16 bytes.
[9] NT: Lembre-se de que o autor está considerando uma plataforma de 16 bits.
[10] NT: Não confundir o termo *objeto* usado no paradigma imperativo orientado a objetos.

esquerda, pronunciado "*el value*"), respectivamente.

Em algumas linguagens, o *lvalue* é o valor permitido no lado esquerdo do operador de atribuição "=" (ou seja, o endereço onde o resultado da avaliação do lado direito acaba). O *rvalue* é aquele que está no lado direito da instrução de atribuição, como o valor "**2**" exemplificado. *Rvalues* não podem ser usados no lado esquerdo da instrução de atribuição. Portanto, "**2 = k"**; é considerado ilegal.

Na verdade, a definição de "*lvalue*" é um pouco modificada para C. De acordo com a referência "K&R II (page 197): [1]":

> "*An **object** is a named region of storage; an **lvalue** is an expression referring to an object*"
>
> "*Um **objeto** é uma região nomeada de armazenamento; um **lvalue** é uma expressão que se refere a um objeto*".

Entretanto, para este momento, a definição originalmente citada é suficiente. Conforme ficamos mais familiarizados com ponteiros, entraremos em mais detalhes sobre isso.

Muito bem, considere então:

```
int j, k;

k = 2;
j = 7;      <-- linha 1
k = j;      <-- linha 2
```

Neste exemplo, o compilador interpreta "**j**" na *linha 1* como sendo o endereço da variável "**j**" (seu *lvalue*) e cria um código para copiar o valor "**7**" para esse endereço. Já na *linha 2*, a variável "**j**" é interpretada como seu *rvalue* (já que está definida ao lado direito do operador de atribuição "="). Ou seja, aqui o "**j**" se refere ao valor ***armazenado*** no local de memória reservado para "**j**", nesse caso, o valor "**7**". Então, "**7**" é copiado para o endereço designado pelo *lvalue* de "**k**".

Em todos esses exemplos, estamos usando inteiros de *2 bytes*, então toda cópia de *rvalues* de um local de armazenamento para o outro local é feito copiando *2 bytes*. Se estivéssemos usando inteiros do tipo **long**, estaríamos copiando *4 bytes*.

Agora, suponha que temos uma razão para querer uma variável projetada para armazenar um *lvalue* (um endereço). O tamanho necessário para armazenar tal valor depende do sistema. Em computadores do tipo desktop mais antigos com 64K de memória total, o endereço de qualquer ponto na memória pode ser contido em *2 bytes*. Computadores com mais memória exigiriam mais bytes para manter um endereço. Alguns computadores, como o IBM PC, podem exigir tratamento especial para manter um segmento e um deslocamento em certas circunstâncias. O tamanho real necessário não é importante, desde que tenhamos uma maneira de informar ao compilador que o que queremos armazenar é um endereço.

Esse tipo de variável é chamado de ***variável ponteiro*** (por razões que esperamos que fi-

quem mais claras um pouco mais adiante). Em C, a definição de variável ponteiro é feita colocando-se um símbolo de asterisco antes do seu nome e também se atribuindo um tipo de dado ao nosso ponteiro, que neste caso se refere ao tipo de dado armazenado no endereço que estaremos associando em nosso ponteiro. Por exemplo, considere a declaração da variável:

```
int *ptr;
```

"**ptr**" é o nome da nossa variável (assim como "**k**" era o nome da nossa variável inteira). O símbolo "*****" informa ao compilador que queremos usar uma variável do tipo ponteiro, ou seja, pede-se para reservar a quantidade de bytes necessária para armazenar um endereço na memória. O "**int**" indica que pretendemos usar nossa variável do tipo ponteiro para armazenar o endereço de um dado inteiro. Tal ponteiro é chamado de "*apontar para*" um inteiro. No entanto, observe que, quando escrevemos "**int k;**", não definimos um valor para a variável "**k**". Se essa definição for feita fora de qualquer função, os compiladores compatíveis com ANSI a inicializarão como zero. Da mesma forma, "**ptr**" não tem valor, ou seja, não armazenamos um endereço nele em sua declaração. Nesse caso, novamente, se a declaração estiver fora de qualquer função, ela será inicializada com um valor que não aponte para nenhum objeto ou função C. Um ponteiro inicializado dessa maneira é chamado de "*ponteiro nulo*" ("**null**" *pointer*).

O padrão real de bits usados para um ponteiro nulo pode ou não ser avaliado como zero, uma vez que isso depende do sistema específico no qual o código é desenvolvido. Para tornar o código fonte compatível entre vários compiladores em vários sistemas usa-se uma macro para representar um *ponteiro nulo*. Essa macro é chamada de **NULL**. Assim, definir o valor de um ponteiro usando a macro **NULL**, como em uma declaração de atribuição como **ptr = NULL** garante que o ponteiro se torne um ponteiro nulo. Da mesma forma, assim como se pode testar um valor inteiro zero, como em "**if (k == 0)**", podemos testar um ponteiro nulo usando "**if (ptr == NULL)**".

Mas voltando a usar nossa variável "**ptr**". Suponha agora que queremos armazenar no ponteiro "**ptr**" o endereço da nossa variável inteira "**k**". Para fazer isso, usamos o operador unário "**&**" (e comercial) e escrevemos:

```
ptr = &k;
```

O operador "**&**" obtém o *lvalue* (endereço) da variável "**k**", mesmo que "**k**" esteja do lado direito do operador de atribuição "=", e copia isso para o conteúdo do nosso ponteiro "***ptr***". Agora, "**ptr**" é dito como "*apontar para*" o endereço de memória para a variável "**k**". Apenas mais um operador precisa ser discutido.

O "*operador de desreferenciamento*"[11] é representado pelo símbolo de asterisco sendo utilizado da seguinte forma:

```
*ptr = 7;
```

isso irá copiar o valor "**7**" para o endereço de memória apontado por "**ptr**". Portanto, se "**ptr**" aponta para o endereço da variável "**k**", essa instrução definirá o valor de "**k**" como

[11] NT: O *operador de desreferenciamento* também é conhecido como *operador de indireção*.

"**7**". Ou seja, quando usamos o "*****" dessa forma, estamos nos referindo ao valor daquilo para o qual "**ptr**" está apontando e não ao valor do próprio ponteiro.

Da mesma forma, poderíamos escrever:

```
printf("%d\n", *ptr);
```

para imprimir na tela o valor inteiro armazenado no endereço apontado por "**ptr**".

Uma maneira de entender como todas essas coisas se encaixam é executando o programa 1.1 (prog0101.c) e, em seguida, fazer uma cuidadosa análise do comportamento do código e saída apresentada.

```
/* Programa 1.1 extraído de PTRTUT10.TXT - 10/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                youtube.com/@AM-42 */

#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
  j = 1;
  k = 2;
  ptr = &k;
  printf("\n");
  printf("j tem valor %d, armazenado em %p\n", j, (void*) &j);
  printf("k tem valor %d, armazenado em %p\n", k, (void*) &k);
  printf("ptr tem valor %p, armazenado em %p\n", ptr, (void*) &ptr);
  printf("O valor do inteiro apontado para ptr e' %d\n", *ptr);
  return 0;
}
```

Nota: Ainda não discutimos os aspectos da linguagem C que requerem o uso da expressão (**void ***), usada neste exemplo. Por enquanto, não se preocupe com isso. O motivo por trás do uso dessa expressão será abordado mais adiante.

Para revisar:

- Uma variável é declarada dando-lhe um tipo e um nome (por exemplo, "**int k**;")
- Uma variável do tipo ponteiro é declarada dando-lhe um tipo e um nome (por exemplo, "**int *ptr**"), onde o asterisco informa ao compilador que a variável chamada "**ptr**" é uma variável especial para ponteiro e o tipo de dado usado nela informa ao compilador qual é o tipo que o ponteiro deve apontar (no caso do exemplo fornecido, inteiro).
- Uma vez que uma variável é declarada, podemos obter seu endereço precedendo seu nome com o operador unário "**&**", como em "**&k**".
- Podemos "desreferenciar" um ponteiro, ou seja, referir-se ao valor ao qual ele aponta, usando o operador unário "*****" como em "***ptr**".
- Um "*lvalue*" de uma variável é o valor de seu endereço, ou seja, onde a variável está

armazenada na memória. O "*rvalue*" de uma variável é o valor armazenado nessa variável, ou seja, neste endereço.

Referência para o Capítulo 1:

1. "The C Programming Language" 2nd Edition
B. Kernighan and D. RitchiePrentice Hall
ISBN 0-13-110362-8

CAPÍTULO 2: Tipos de ponteiros e arranjos

Muito bem, continuando. Consideremos que precisamos identificar o tipo de variável para o qual um ponteiro aponta, como em:

```
int *ptr;
```

Uma razão para fazer isso é para que, mais tarde, uma vez que "**ptr**" aponta para algo, se escrevermos:

```
*ptr = 2;
```

o compilador saberá quantos bytes copiar para aquele local de memória apontado por "**ptr**". Se "**ptr**" ***foi*** declarado como apontando para um inteiro, *2 bytes* serão copiados, se for um **long** *4 bytes* serão copiados. Da mesma forma, para **floats** e **doubles** o número apropriado será copiado. Mas, definir o tipo para o qual o ponteiro aponta permite uma série de outras maneiras interessantes que um compilador poder interpretar o código. Por exemplo, considere um bloco de memória consistindo em dez inteiros em sequência. Isto é, *20 bytes* de memória são reservados para armazenar *10 inteiros*.

Agora, suponha que apontemos nosso ponteiro inteiro "**ptr**" para o primeiro desses inteiros. Além disso, digamos que o inteiro está localizado no endereço de memória **100** (decimal). O que acontece quando escrevemos:

```
ptr + 1;
```

pelo fato do compilador "saber" que isso é um ponteiro (ou seja, seu valor é um endereço) e que ele aponta para um inteiro (seu endereço atual, **100**, é o endereço de um inteiro), ele adiciona "**2**" a "**ptr**" em vez de "**1**", fazendo com que o ponteiro "*aponte para*" o **próximo inteiro**, no endereço de memória **102**. Da mesma forma, se "**ptr**" fosse declarado como um ponteiro para um **long**, ele adicionaria "**4**" em vez de "**1**". O mesmo vale para outros tipos de dados, como **floats**, **doubles** ou mesmo tipos de dados definidos pelo usuário, como estruturas (**structs**). Isso obviamente não é o mesmo tipo de "*adição*" que normalmente pensamos. Em C, isso é referido como adição usando "*aritmética de ponteiros*", um termo que voltaremos a discutir mais adiante.

Da mesma forma, uma vez que "**++ptr**" e "**ptr++**" são equivalentes a "**ptr + 1**" (embora o momento em que "**ptr**" é incrementado no programa possa ser diferente), incrementar um ponteiro usando o operador unário de incremento "**++**" da forma pré-fixado ou pós-fixado incrementa o endereço que ele armazena em "**sizeof(type)**", onde "**type**" é o tipo do objeto apontado. (ou seja, "**2**" para um **int**, "**4**" para um **long**, etc.).

Já que um bloco de *10 inteiros* localizados contiguamente na memória é, por definição, um **array** (*arranjo*) de inteiros, isso nos traz uma relação interessante entre *arranjos* e *ponteiros*.

Considere o seguinte:

```
int my_array[] = {1,23,17,4,-5,100};
```

Aqui temos um arranjo contendo *6 inteiros*. Nós nos referimos a cada um desses inteiros por meio de um índice de subscrição para "**my_array**", ou seja, usando "**my_array[0]** " até "**my_array[5]** ". Mas, poderíamos acessá-los alternativamente por meio de um ponteiro da seguinte forma:

```
int *ptr;
ptr = &my_array[0];     /* apontar nosso ponteiro para o primeiro
                           inteiro em nosso arranjo */
```

E então poderíamos imprimir coleção de dados usando a notação de arranjo ou desreferenciando nosso ponteiro, como ilustrado no programa 1.2 (prog0102.c):

```
/* Programa 1.2 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                youtube.com/@AM-42 */

#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
  int i;
  ptr = &my_array[0];   /* apontar nosso ponteiro para o primeiro
                           inteiro em nosso arranjo */

  printf("\n");
  for (i = 0; i <= 5; i++)
  {
    printf("my_array[%d] = %d ", i, my_array[i]);   /*<-- A */
    printf("ptr + %d = %d\n", i, *(ptr + i));       /*<-- B */
  }
  return 0;
}
```

Compile e execute o programa acima e observe cuidadosamente o resultado apresentado a partir das linhas "A" e "B". Observe que o programa imprime os mesmos valores em ambos os casos. Note também como desreferenciamos nosso ponteiro na linha "B", ou seja, primeiro adicionamos "**i**" a ele e depois desreferenciamos o novo ponteiro. Altere a linha "B" para o seguinte:

```
printf("ptr + %d = %d\n", i, *ptr++);
```

E execute novamente... então mude para:

```
printf("ptr + %d = %d\n", i, *(++ptr));
```

e tente novamente. A cada vez, tente prever o resultado e observe cuidadosamente o resultado real.

Em C, o padrão estabelece que onde quer que possamos usar "**&var_nome[0]** ", podemos substituir isso por "**var_nome**", portanto em nosso código, onde escrevemos:

```
ptr = &my_array[0];
```

podemos reescrevê-lo como:

```
ptr = my_array;
```

Para obter o mesmo resultado.

Isso leva muitos textos a afirmar que o nome de um arranjo é um ponteiro. Eu prefiro pensar mentalmente que "*o nome do arranjo é o endereço do primeiro elemento no arranjo*". Muitos iniciantes (incluindo eu quando estava aprendendo) têm a tendência de ficarem confusos pensando nisso como um ponteiro. Por exemplo, enquanto podemos escrever:

```
ptr = my_array;
```

não podemos escrever

```
my_array = ptr;
```

A razão é que enquanto "**ptr**" é uma variável, "**my_array**" é uma constante. Ou seja, a localização em que o primeiro elemento de "**my_array**" é armazenado não pode ser alterada depois que "**my_array[]**" é declarado.

Anteriormente, ao discutir o termo "*lvalue*", citei K&R-2, onde se afirmava:

> "*An **object** is a named region of storage; an **lvalue** is an expression referring to an object*"
>
> "*Um **objeto** é uma região nomeada de armazenamento; um **lvalue** é uma expressão que se refere a um objeto*".

Isso levanta um problema interessante. Já que "**my_array**" é uma região nomeada de armazenamento, por que "**my_array**" na instrução de atribuição anterior não é um "*lvalue*"? Para resolver esse problema, alguns se referem a "**my_array**" como um "*lvalue inalterável*".

Modifique o programa de exemplo acima alterando

```
ptr = &my_array[0];
```

para

```
ptr = my_array;
```

e execute-o novamente para verificar se os resultados são idênticos.

Agora, vamos aprofundar um pouco mais na diferença entre os nomes "**ptr**" e "**my_array**" utilizados. Alguns autores referem-se ao nome de um arranjo como um ponteiro ***constante***. O que queremos dizer com isso? Bem, para entender o termo "*constante*" nesse sentido, voltemos à nossa definição do termo "*variável*". Quando declaramos uma variável, reservamos um local na memória para armazenar o valor do tipo apropriado. Uma vez feito isso, o nome da variável pode ser interpretado de duas maneiras. Quando usado no lado esquerdo do operador de atribuição, o compilador o interpreta como o local de memória para o qual deve-se mover aquele valor resultante da avaliação do lado direito do operador de atribuição. Mas, quando usado no lado direito do operador de atribuição, o nome de uma variável é interpretado como o conteúdo armazenado nesse endereço de memória reservado para armazenar o valor dessa variável.

Com isso em mente, vamos agora considerar a mais simples das constantes, como em:

```
int i, k; i = 2;
```

Aqui, enquanto "**i**" é uma variável e ocupa um espaço na parte de dados da memória, "**2**" é uma constante e, como tal, em vez de reservar uma memória no segmento de dados, ela é incorporada diretamente no segmento de código da memória. Ou seja, enquanto escrever algo como "**k = i**"; diz ao compilador para criar um código que, durante a execução, olhará para a localização da memória "**&i**" para determinar o valor a ser movido para "**k**", o código criado por "**i = 2**; " simplesmente coloca o valor "**2**" no código e não há referência ao segmento de dados. Ou seja, tanto "**k**" quanto "**i**" são objetos, mas "**2**" não é um objeto.

Da mesma forma, no exemplo anterior, como "**my_array**" é uma constante, uma vez que o compilador estabelece onde o próprio arranjo deve ser armazenado, ele "conhece" o endereço de "**my_array[0]** " e, ao ver:

```
ptr = my_array;
```

simplesmente usa este endereço como uma constante no segmento de código e não há referência do segmento de dados além disso.

Este pode ser um bom lugar para explicar ainda mais o uso da expressão " **(void *)**" indicada no Programa 1.1 do Capítulo 1. Como vimos, podemos ter ponteiros de vários tipos. Até agora, discutimos ponteiros para inteiros. Nos próximos capítulos, aprenderemos sobre ponteiros para estruturas e até mesmo ponteiros para ponteiros.

Também aprendemos que em diferentes sistemas, o tamanho de um ponteiro pode variar. Acontece também que é possível que o tamanho de um ponteiro varie dependendo do tipo de dados do objeto ao qual ele aponta. Assim, como com inteiros, onde você pode ter problemas ao tentar atribuir um *inteiro longo* a uma variável do tipo *inteiro curto*, você pode ter problemas ao tentar atribuir os valores de ponteiros de vários tipos a variáveis de ponteiro de outros tipos.

Para minimizar este problema, C fornece um ponteiro do tipo "**void**". Podemos declarar tal ponteiro escrevendo:

```
void *vptr;
```

Um ponteiro "**void**" é uma espécie de ponteiro genérico. Por exemplo, embora C não permita a comparação de um ponteiro do tipo inteiro com um ponteiro do tipo caractere, qualquer um desses tipos pode ser comparado com um ponteiro "**void**". Claro, assim como com outras variáveis, conversões podem ser usadas para converter de um tipo de ponteiro para outro sob as circunstâncias apropriadas. No Programa 1.1 do Capítulo 1, eu converti os ponteiros para inteiros em ponteiros void para torná-los compatíveis com a especificação de conversão "**%p**". Em capítulos posteriores, outras conversões serão feitas por razões definidas neles.

Bem, isso é muito técnico para digerir e eu não espero que um iniciante entenda tudo isso na primeira leitura. Com o tempo e experimentação, você vai querer voltar e reler os primeiros dois capítulos. Mas por agora, vamos seguir em frente para a relação entre ponteiros, matrizes de caracteres e cadeias.

CAPÍTULO 3: Ponteiros e cadeias

O estudo de cadeias é útil para reforçar a relação entre ponteiros e arranjos. Também torna fácil ilustrar como algumas das funções de cadeias do padrão C podem ser implementadas. Por fim, isso ilustra como e quando ponteiros podem e devem ser passados para funções.

Em C, as cadeias são arranjos de caracteres. Isso não é necessariamente verdadeiro em outras linguagens. Em BASIC, Pascal, Fortran e várias outras linguagens, uma cadeia tem seu próprio tipo de dados. Mas em C não tem. Em C, uma cadeia é um arranjo de caracteres terminado com um caractere binário zero (escrito como "**\0**"). Para começar nossa discussão, escreveremos algum código que, embora preferido para fins ilustrativos, você provavelmente nunca escreveria em um programa real. Considere, por exemplo:

```
char my_string[40];

my_string[0] = 'T';
my_string[1] = 'e';
my_string[2] = 'd':
my_string[3] = '\0';
```

Embora nunca se construa uma cadeia de caracteres dessa maneira, o resultado final é uma cadeia no sentido de que é um arranjo de caracteres **terminado com um caractere nulo**. Por definição, em C, uma cadeia é um arranjo de caracteres terminado com o caractere nulo. Esteja ciente de que "***nul*" não** é o mesmo que "***NULL***". O "**nul**" se refere a um zero conforme definido pela sequência de *escape* "**\0**". Ou seja, ocupa um byte de memória. Já o "**NULL**" é o nome da macro usada para inicializar ponteiros nulos. O "*NULL*" é definido em um arquivo de cabeçalho em seu compilador C, enquanto o "*nul*" pode não ser definido de forma alguma.

Já que escrever o código anterior seria muito demorado, C permite duas maneiras alternativas de alcançar o mesmo objetivo. Primeiro, alguém poderia escrever:

```
char my_string[40] = {'T', 'e', 'd', '\0',};
```

Mas isso também exige mais digitação do que é conveniente. Então, o C permite que seja escrito:

```
char my_string[40] = "Ted";
```

Quando as aspas inglesas (") são usadas, em vez das aspas simples (') como feito nos exemplos anteriores, o caractere nulo ("**\0**") é automaticamente adicionado ao final da cadeia.

Em todos os casos apresentados, a mesma coisa acontece. O compilador reserva um bloco contíguo de memória com 40 bytes de comprimento para armazenar caracteres e o inicializa de tal forma que os primeiros "**4**" caracteres são "**Ted\0**".

Agora, considere o programa 3.1 (prog0301.c):

```
/* Programa 3.1 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                youtube.com/@AM-42 */

#include <stdio.h>

char strA[80] = "Cadeia usada para fins de demonstracao";
char strB[80];

int main(void)
{
  char *pA;      /* um ponteiro para entrada de caractere */
  char *pB;      /* outro ponteiro para entrada de caractere */
  puts(strA);    /* apresenta cadeia A */
  pA = strA;     /* ponteiro pA para a cadeia A */
  puts(pA);      /* mostrar o que pA está apontando */
  pB = strB;     /* ponteiro pB para a cadeia B */
  putchar('\n'); /* move para baixo uma linha na tela */
  while (*pA != '\0')   /* linha A (ver texto) */
  {
    *pB++ = *pA++;      /* linha B (ver texto) */
  }
  *pB = '\0';           /* linha C (ver texto) */
  puts(strB);           /* mostra strB na tela */
  return 0;
}
```

No programa 3.1, começamos definindo dois arranjos de caracteres com 80 caracteres cada. Como eles são definidos globalmente, são inicializados primeiro com todas as posições como "**0**". Então, "**strA**" tem seus primeiros 39 caracteres inicializados com a cadeia definida entre aspas inglesas.

Agora, movendo-se para o código, declaramos dois ponteiros de caractere e mostramos a cadeia na tela. Então, "*apontamos*" o ponteiro "**pA**" para "**strA**". Ou seja, por meio da instrução de atribuição, copiamos o endereço de "**strA[0]** " em nossa variável "**pA**". Agora, usamos "**puts()**" para mostrar na tela aquilo que é apontado por "**pA**". Considere aqui que o protótipo da função para "**puts()**" é:

```
int puts(const char *s);
```

Ignorando por enquanto o "**const**", o parâmetro passado para "**puts()**" é um ponteiro, ou seja, o **valor** de um ponteiro (já que todos os parâmetros em C são passados por valor), e o valor de um ponteiro é o endereço para o qual ele aponta, ou simplesmente, um endereço. Portanto, quando escrevemos "**puts(strA);**", como vimos, estamos passando o endereço de "**strA[0]**".

Da mesma forma, quando escrevemos "**puts(pA);**" estamos passando o mesmo endereço, uma vez que definimos "**pA = strA;**".

Dado isso, acompanhe o código até a instrução "**while()**" na linha "A". A linha "A" afirma:

Enquanto o caractere apontado por "**pA**" (ou seja, "***pA**") não é um caractere nulo (ou seja, o caractere terminador "**\0**"), faça o seguinte:

A linha "B" está declarado: copie o caractere apontado por "**pA**" para o espaço apontado por "**pB**", em seguida, incremente "**pA**" para que ele aponte para o próximo caractere e "**pB**" para que ele aponte para o próximo espaço.

Quando copiamos o último caractere, "**pA**" agora aponta para o caractere nulo de término e o loop termina. No entanto, não copiamos o caractere nulo. E, por definição, uma cadeia em C **deve** ser terminada com nulo. Então, adicionamos o caractere nulo com a linha C.

É muito educativo executar este programa com seu depurador enquanto observa "**strA**", "**strB**", "**pA**" e "**pB**" e avança o programa passo a passo. É ainda mais educativo se, em vez de simplesmente definir "**strB[]**" como feito, inicializá-lo também com algo como:

```
strB[80] = "12345678901234567890123456789012345678901234567890"
```

onde o número de dígitos usados é maior do que o comprimento de "**strA**" e, em seguida, repita o procedimento de passo único enquanto observa as variáveis acima. Experimente essas coisas!

Voltando ao protótipo de "**puts()**" por um momento, o "**const**" usado como um modificador de parâmetro informa o usuário que a função não modificará a cadeia apontada por "**s**", ou seja, ela tratará essa cadeia como uma constante.

Antes de tudo, vamos revisar o programa anterior que ilustra como copiar uma cadeia. Depois de compreendermos bem o que acontece, poderemos criar nossa própria função de cópia de cadeia que substituirá a função "**strcpy()**" padrão do C. O programa pode ser parecido com o seguinte:

```
char *my_strcpy(char *destination, char *source)
{
  char *p = destination;
  while (*source != '\0')
    *p++ = *source++;
  *p = '\0';
  return destination;
}
```

Nesse caso, eu segui a prática usada na rotina padrão de retornar um ponteiro para o destino.

Mais uma vez, a função é projetada para aceitar os valores de dois ponteiros de caractere, ou seja, endereços, e assim no programa anterior poderíamos escrever:

```
int main(void)
{
  my_strcpy(strB, strA);
  puts(strB);
}
```

Eu desviei um pouco da forma usada no C padrão, que teria o protótipo:

```
char *my_strcpy(char *destination, const char *source);
```

Aqui, o modificador "**const**" é usado para garantir ao usuário que a função não modificará o conteúdo apontado pelo ponteiro de origem. Você pode provar isso modificando a função acima e seu protótipo para incluir o modificador "**const**", conforme mostrado. Em seguida, dentro da função, você pode adicionar uma instrução que tenta alterar o conteúdo do que é apontado por "**source**", como:

```
*source = 'X';
```

o que normalmente mudaria o primeiro caractere da cadeia para um "**X**". O modificador "**const**" deve fazer com que seu compilador detecte isso como um erro. Tente e veja.

Agora, vamos considerar algumas das coisas que os exemplos acima nos mostraram. Primeiro, considere o fato de que "***ptr++**" deve ser interpretado como retornando o valor apontado por "**ptr**" e, em seguida, incrementando o valor do ponteiro. Isso tem a ver com a precedência dos operadores. Se escrevêssemos " **(*ptr)++** ", incrementaríamos não o ponteiro, mas o valor apontado pelo ponteiro, ou seja, se usado no primeiro caractere da cadeia de exemplo acima, o "**T**" seria incrementado para um "**U**". Você pode escrever um código de exemplo simples para ilustrar isso.

Lembre-se novamente que uma cadeia não é mais do que um arranjo de caracteres, sendo o último caractere um "**\0**". O que fizemos foi lidar com a cópia de um arranjo. No exemplo foi usado um arranjo de caracteres, mas a técnica poderia ser aplicada a um arranjo de valores inteiros, reais (ponto flutuante), etc. Nesses casos, no entanto, não estaríamos lidando com cadeias e, portanto, o final do arranjo não seria marcado com um valor especial como o caractere nulo. Poderíamos implementar uma versão que dependesse de um valor especial para identificar o final. Por exemplo, poderíamos copiar um arranjo de inteiros positivos marcando o final com um inteiro negativo. Por outro lado, é mais comum que, ao escrever uma função para copiar um arranjo de elementos que não sejam cadeias, passemos para a função o número de elementos a serem copiados, bem como o endereço do arranjo, por exemplo, algo como o seguinte protótipo poderia indicar:

```
void int_copy(int *ptrA, int *ptrB, int nbr);
```

onde "**nbr**" é o número de inteiros a serem copiados. Você pode querer brincar com essa ideia e criar uma matriz de inteiros e ver se consegue escrever a função "**int_copy()**" e fazê-la funcionar.

Isso permite usar funções para manipular grandes matrizes. Por exemplo, se tivermos uma matriz de 5000 inteiros que desejamos manipular com uma função, precisamos apenas passar para essa função o endereço da matriz (e qualquer informação auxiliar, como "**nbr**" indicado anteriormente, dependendo do que estamos fazendo). A matriz em si **não** é passada, ou seja, a matriz inteira não é copiada e colocada na pilha antes de chamar a função, apenas seu endereço é enviado.

Isso é diferente de passar, por exemplo, um inteiro para uma função. Quando passamos um inteiro, fazemos uma cópia do inteiro, ou seja, obtemos o seu valor e o colocamos na pilha. Dentro da função, qualquer manipulação do valor passado não pode afetar o inteiro original. Mas, com matrizes e ponteiros, podemos passar o endereço da variável e, portanto, manipular os valores das variáveis originais.

CAPÍTULO 4: Mais sobre cadeias

Bem, progredimos bastante em pouco tempo! Vamos voltar um pouco e olhar para o que foi feito no Capítulo 3 sobre a cópia de cadeias, mas de uma maneira diferente. Considere a seguinte sub-rotina:

```
char *my_strcpy(char dest[], char source[])
{
  int i = 0;
  while (source[i] != '\0')
  {
    dest[i] = source[i];
    i++;
  }
  dest[i] = '\0';
  return dest;
}
```

Lembre-se que cadeias são arranjos (vetores/arrays) de caracteres. Aqui escolhemos usar a notação de arranjo em vez da notação de ponteiro para fazer a cópia real. Os resultados são os mesmos, ou seja, a cadeia é copiada com a mesma precisão que antes. Isso levanta alguns pontos interessantes que discutiremos.

Como os parâmetros são passados por valor, tanto no uso de um ponteiro de caractere quanto no nome do arranjo como indicado, o que é passado na verdade é o endereço do primeiro elemento de cada arranjo. Assim, o valor numérico do parâmetro passado é o mesmo, quer usemos um ponteiro de caractere ou usemos um nome de arranjo como parâmetro. Isso tende a implicar que de alguma forma "**source[i]**" é o mesmo que "***(p + i)**".

Onde quer que se escreva "**a[i]**", pode ser substituído por "***(a + i)**" sem nenhum problema. Na verdade, o compilador criará o mesmo código em ambos os casos. Assim, vemos que a aritmética de ponteiros é a mesma coisa que a indexação de arranjos. Qualquer uma das sintaxes produz o mesmo resultado.

Isso NÃO significa que ponteiros e arranjos são a mesma coisa, eles não são. Estamos apenas dizendo que para identificar um determinado elemento de um arranjo, temos a opção de duas sintaxes, uma usando indexação e outra usando aritmética de ponteiro, que produzem resultados idênticos.

Agora, olhando para essa última expressão, parte dela, "**(a + i)**", é uma adição simples usando o operador "+" e as regras de C afirmam que tal expressão é comutativa. Ou seja, "**(a + i)**" é idêntico a "**(i + a)**". Assim, poderíamos escrever "***(i + a)**" tão facilmente quanto "***(a + i)**".

Mas "***(i + a)**" poderia ter vindo de "**i[a]**"! De tudo isso surge a curiosa verdade de que se:

```
char a[20];
int i;
```

escrevendo isso

```
a[3] = 'x';
```

é o mesmo que escrever

```
3[a] = 'x';
```

Tente! Configure um arranjo de caracteres, inteiros ou inteiros longos, etc. e atribua o valor do terceiro ou quarto elemento usando a abordagem convencional e, em seguida, imprima esse valor para ter certeza de que está funcionando. Em seguida, inverta a notação do arranjo como eu fiz acima. Um bom compilador não terá problemas e os resultados serão idênticos. Uma curiosidade... nada mais!

Agora, olhando para nossa função, quando escrevemos:

```
dest[i] = source[i];
```

devido ao fato de que a indexação de matriz e a aritmética de ponteiros produzem resultados idênticos, podemos escrever isso como:

```
*(dest + i) = *(source + i);
```

Mas, isso requer *2 adições* para cada valor assumido por "**i**". Adições, em geral, levam mais tempo do que incrementações (como as feitas usando o operador "++" como em "**i**++"). Isso pode não ser verdade em compiladores modernos de otimização, mas nunca se pode ter certeza. Portanto, a versão do ponteiro pode ser um pouco mais rápida do que a versão do arranjo.

Outra maneira de acelerar a versão de ponteiro seria mudar:

```
while (*source != '\0')
```

para simplesmente

```
while (*source)
```

desde que o valor dentro do parêntese se tornará zero (FALSO) ao mesmo tempo em ambos os casos.

Neste ponto, você pode querer experimentar um pouco escrevendo seus próprios programas usando ponteiros. Manipular cadeias é um bom lugar para experimentar. Você pode querer escrever suas próprias versões de funções padrão como:

```
strlen();
strcat();
strchr();
```

e outros que você possa ter no seu sistema.

Voltando ao tópico de estruturas, por enquanto, vamos seguir em frente e discutir estruturas por um momento.

CAPÍTULO 5: Ponteiros e estruturas

Como você pode saber, podemos declarar a forma de um bloco de dados contendo diferentes tipos de dados por meio de uma declaração de estrutura. Por exemplo, um arquivo de pessoal pode conter estruturas que se parecem com isso:

```
struct tag {
  char lname[20]; /* last name (ultimo nome) */
  char fname[20]; /* first name (primeiro nome) */
  int age;        /* age (idade) */
  float rate;     /* exemplo: 12.75 por hora */
};
```

Digamos que temos um monte dessas estruturas em um arquivo de disco e queremos ler cada uma delas e imprimir o primeiro e o último nome de cada uma para que possamos ter uma lista das pessoas em nossos arquivos. As informações restantes não serão impressas. Queremos fazer esta impressão com uma chamada de função e passar para essa função um ponteiro para a estrutura em questão. Para fins de demonstração, usarei apenas uma estrutura por enquanto. Mas perceba que o objetivo é escrever a função, não a leitura do arquivo que, presumivelmente, sabemos como fazer.

Para revisão, lembre-se de que podemos acessar os membros de uma estrutura com o operador de ponto, como mostra o programa 5.1 (prog0501.c):

```
/* Programa 5.1 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                  youtube.com/@AM-42 */

#include <stdio.h>
#include <string.h>

struct tag {
  char lname[20]; /* last name (ultimo nome) */
  char fname[20]; /* first name (primeiro nome) */
  int age;        /* age (idade) */
  float rate;     /* exemplo: 12.75 por hora */
};

struct tag my_struct;   /* declarar a estrutura "my_struct" */

int main(void)
{
  strcpy(my_struct.lname, "Jensen");
  strcpy(my_struct.fname, "Ted");
  printf("\n%s ", my_struct.fname);
  printf("%s\n", my_struct.lname);
  return 0;
}
```

Agora, essa estrutura em particular é bastante pequena em comparação com muitas usadas em programas em C. Podemos querer adicionar ao exemplo apresentado:

```
date_of_hire;           (tipos de dados não mostrados)
date_of_last_raise;
last_percent_increase;
emergency_phone;
medical_plan; S
ocial_S_Nbr;
etc.....
```

Se tivermos um grande número de funcionários, o que queremos é manipular os dados nessas estruturas por meio de funções. Por exemplo, podemos querer uma função para imprimir o nome do funcionário listado em qualquer estrutura passada para ela. No entanto, no C original (Kernighan & Ritchie, 1ª edição), não era possível passar uma estrutura completa, apenas um ponteiro para uma estrutura poderia ser passado. No ANSI C, agora é permitido passar a estrutura completa. Mas, como nosso objetivo aqui é aprender mais sobre ponteiros, não iremos seguir essa abordagem.

De qualquer forma, se passarmos toda a estrutura, isso significa que devemos copiar o conteúdo da estrutura da função de chamada para a função chamada. Em sistemas que usam pilhas, isso é feito empurrando o conteúdo da estrutura na pilha. Com estruturas grandes, isso poderia ser um problema. No entanto, passar um ponteiro usa uma quantidade mínima de espaço de pilha.

Em todo caso, uma vez que esta é uma discussão sobre ponteiros, vamos discutir como passamos um ponteiro para uma estrutura e, em seguida, como a usamos dentro da função.

Considere o caso descrito, ou seja, queremos uma função que aceite como parâmetro um ponteiro para uma estrutura e, a partir dessa função, queremos acessar membros da estrutura. Por exemplo, queremos imprimir o nome do funcionário em nossa estrutura de exemplo.

Ok, então sabemos que nosso ponteiro apontará para uma estrutura declarada usando a *tag* "**struct**". Declaramos tal ponteiro com a declaração:

```
struct tag *st_ptr;
```

e apontamos para nossa estrutura de exemplo com:

```
st_ptr = &my_struct;
```

Agora, podemos acessar um membro específico referenciando o ponteiro. Mas como fazemos para referenciar o ponteiro de uma estrutura? Bem, considere que podemos querer usar o ponteiro para definir a idade do funcionário. Escreveríamos:

```
(*st_ptr).age = 63;
```

Olhando cuidadosamente para o código, ele diz para substituir o que está entre parênteses com o que o ponteiro "**st_ptr**" aponta, que é a estrutura "**my_struct**". Portanto, isso se resume ao mesmo que "**my_struct.age**".

No entanto, esta é uma expressão bastante usada e os projetistas da linguagem C criaram uma sintaxe alternativa com o mesmo significado, que é:

```
st_ptr->age = 63;
```

Com isso em mente, observe o programa 5.2 (prog0502.c):

```
/* Programa 5.2 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                  youtube.com/@AM-42 */

#include <stdio.h>
#include <string.h>

struct tag
{ /* o tipo de estrutura */
  char lname[20]; /* last name (ultimo nome) */
  char fname[20]; /* first name (primeiro nome) */
  int age;        /* age (idade) */
  float rate;     /* exemplo: 12.75 por hora */
};

struct tag my_struct;          /* define a estrutura */
void show_name(struct tag *p); /* prototipo da funcao */

int main(void)
{
  struct tag * st_ptr;    /* um ponteiro para uma estrutura */
  st_ptr = &my_struct;    /* apontamento do ponteiro para "my_struct" */
  strcpy(my_struct.lname, "Jensen");
  strcpy(my_struct.fname, "Ted");
  printf("\n%s ", my_struct.fname);
  printf("%s\n", my_struct.lname);
  my_struct.age = 63;
  show_name(st_ptr);      /* passa o ponteiro */
  return 0;
}

void show_name(struct tag *p)
{
  printf("\n%s ", p->fname);  /* ponto "p" para a estrutura */
  printf("%s ", p->lname);
  printf("%d\n", p->age);
}
```

Novamente, essa é muita informação para absorver de uma vez só. O leitor deve compilar e executar os vários trechos de código e, usando um depurador, monitorar coisas como "**my_struct**" e "**p**" enquanto segue passo a passo pelo código principal e segue o código até a função para entender o que está acontecendo.

CAPÍTULO 6: Mais detalhes sobre arranjos de cadeias

Bem, vamos voltar um pouco para cadeias. Em todos os casos a seguir, as atribuições devem ser entendidas como globais, ou seja, feitas fora de qualquer função, incluindo a função "**main()**".

Apontamos em um capítulo anterior que poderíamos escrever:

```
char my_string[40] = "Ted";
```

o qual alocaria espaço para um arranjo de *40 bytes* e colocaria a cadeia nos primeiros *4 bytes* (três para os caracteres entre as aspas inglesas e o quarto para lidar com o caractere nulo de terminação "**\0**").

Na verdade, se tudo o que queríamos era armazenar o nome "Ted", poderíamos escrever:

```
char my_name[] = "Ted";
```

e o compilador contaria os caracteres, deixaria espaço para o caractere nulo e armazenaria o total de quatro caracteres na memória, cuja localização seria retornada pelo nome do arranjo, neste caso "**my_name**".

Em algum código, em vez do programa apresentado, você pode ver:

```
char *my_name = "Ted";
```

que é uma abordagem alternativa. Existe diferença entre essas duas formas? A resposta é... sim. Usando a notação de arranjo, *4 bytes* de armazenamento na memória estática são usados, um para cada caractere e outro para o caractere nulo de terminação. No entanto, na notação de ponteiro, os mesmos *4 bytes* são necessários, **mais** *N bytes* para armazenar a variável ponteiro "**my_name**" (onde "**N**" depende do sistema, mas é geralmente no mínimo *2 bytes*, podendo ser *4 ou mais bytes*).

A notação de arranjo, "**my_name**" é uma abreviação para "**&myname[0]** ", que é o endereço do primeiro elemento do arranjo. Como a localização do arranjo é fixa durante o tempo de execução, isso é uma constante (e não uma variável). A notação de ponteiro, "**my_name**" é uma variável. Quanto a qual método é melhor, isso depende do que você pretende fazer no restante do programa.

Vamos agora um passo além e considerar o que acontece se cada uma dessas declarações for feita dentro de uma função, em vez de globalmente fora do escopo de qualquer função.

```
void my_function_A(char *ptr)
{
  char a[] = "ABCDE"
  .
  .
}

void my_function_B(char *ptr)
```

```
{
  char *cp = "FGHIJ"
  .
  .
}
```

No caso de "**my_function_A**", o conteúdo ou o(s) valor(es) do arranjo "**a[]**" é considerado como dados. O arranjo é dito estar inicializado com os valores *ABCDE*. No caso de "**my_function_B**", o valor do ponteiro "**cp**" é considerado como dados. O ponteiro foi inicializado para apontar para a string "**FGHIJ**". Em ambas as funções "**my_function_A**" e "**my_function_B**", as definições são variáveis locais e, portanto, a cadeia *ABCDE* é armazenada na pilha, assim como o valor do ponteiro "**cp**". A cadeia "**FGHIJ**" pode ser armazenada em qualquer lugar. No meu sistema, ela é armazenada no segmento de dados.

A propósito, a inicialização de matrizes de variáveis automáticas como fiz junto a função "**my_function_A**" era ilegal no antigo K&R C e só "amadureceu" no novo ANSI C. Um fato que pode ser importante quando se considera a portabilidade e a compatibilidade com versões anteriores.

Já que estamos discutindo a relações/diferenças entre ponteiros e arranjos, vamos falar sobre arranjos multidimensionais. Considere, por exemplo, o arranjo:

```
char multi[5][10];
```

O que isso significa? Bem, vamos considerar isso da seguinte maneira.

```
char multi[5][10];
```

Tomando a parte sublinhada como o "*nome*" de um arranjo. Em seguida, considerando o tipo **char** e a dimensão **[10]**, temos um arranjo de *10 caracteres* por linha de dados. Mas, o nome "**multi[5]** " é ele mesmo um arranjo, indicando que existem *5 elementos*, sendo cada um, um arranjo de *10 caracteres*. Portanto, temos um arranjo de 5 arranjos com 10 caracteres cada arranjo.

Vamos supor que preenchemos essa matriz bidimensional com dados de algum tipo. Na memória, ela pode parecer como se tivesse sido formada pela inicialização de 5 arranjos separados usando algo com:

```
multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

Ao mesmo tempo, os elementos individuais podem ser acessados utilizando a sintaxe como:

```
multi[0][3] = '3'
multi[1][7] = 'h'
multi[4][0] = 'J'
```

Uma vez que os arranjos são contíguos na memória, nosso bloco de memória real para o exemplo indicado deve se parecer com o seguinte:

```
0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
^
|     começando no endereço &multi[0][0]
```

Observe que eu **não** escrevi "[**multi[0] = "0123456789"**]". Se eu tivesse feito isso, um caractere de término "**\0**" teria sido implícito, uma vez que sempre que as aspas inglesas são usadas, um caractere "**\0**" é anexado aos caracteres contidos entre essas aspas. Se isso tivesse sido o caso, eu teria que reservar espaço para *11 caracteres* por linha em vez de *10*.

Meu objetivo anterior é ilustrar como a memória é organizada para arranjos bidimensionais. Ou seja, este é um arranjo bidimensional de caracteres, NÃO um arranjo de "cadeias".

Agora, o compilador sabe quantas colunas estão presentes no arranjo, então ele pode interpretar "**multi + 1**" como o endereço do **'a'** na segunda linha acima. Ou seja, ele adiciona *10*, no número de colunas, para obter essa localização. Se estivéssemos lidando com inteiros e um arranjo com a mesma dimensão, o compilador adicionaria "**10*sizeof(int)**" que, na minha máquina, seria *20*. Assim, o endereço do "**9**" na quarta linha acima seria "**&multi[3][0]**" ou "***(multi + 3)**" em notação de ponteiro. Para acessar o conteúdo do segundo elemento na quarta linha, adicionamos "**1**" a este endereço e desreferenciamos o resultado como em

```
*(*(multi + 3) + 1)
```

Com um pouco de reflexão, podemos ver que:

```
*(*(multi + row) + col)     e
multi[row][col]             produzir os mesmos resultados.
```

O programa 6.1 (prog0601.c) ilustra isso usando arranjos de inteiros em vez de arranjos de caracteres.

```
/* Programa 6.1 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 28/04/2023 */
/*                                  youtube.com/@AM-42 */

#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
  int row, col;
  for (row = 0; row < ROWS; row++)
  {
    for (col = 0; col < COLS; col++)
    {
      multi[row][col] = row * col;
```

```
    }
  }

  for (row = 0; row < ROWS; row++)
  {
    for (col = 0; col < COLS; col++)
    {
      printf("\n%d ", multi[row][col]);
      printf("%d ", *(*(multi + row) + col));
    }
  }

  return 0;
}
```

Devido à dupla desreferenciação necessária na versão do ponteiro, o nome de uma matriz bidimensional é frequentemente considerado equivalente a um ponteiro para um ponteiro. Com uma matriz tridimensional, estaríamos lidando com uma matriz de matrizes de matrizes e alguns poderiam dizer que seu nome seria equivalente a um ponteiro para um ponteiro para um ponteiro. No entanto, aqui inicialmente reservamos o bloco de memória para a matriz definindo-a usando a notação de matriz. Portanto, estamos lidando com uma constante, não com uma variável. Ou seja, estamos falando de um endereço fixo, não de um ponteiro variável. A função de desreferenciação usada nos permite acessar qualquer elemento na matriz de matrizes sem a necessidade de alterar o valor desse endereço (o endereço de "**multi[0][0]**" conforme dado pelo símbolo "**multi**").

CAPÍTULO 7: Mais detalhes sobre arranjos mutidimensionais

No capítulo anterior, notamos que, dado

```
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];
```

podemos acessar elementos individuais do arranjo "**multi**" usando tanto:

```
multi[row][col]
```

como

```
*(*(multi + row) + col)
```

Para entender melhor o que está acontecendo, vamos substituir

```
*(multi + row)
```

com "**X**" como em:

```
*(X + col)
```

Agora, a partir disso, vemos que "**X**" é como um ponteiro, uma vez que a expressão é desreferenciada e sabemos que **col** é um inteiro. Aqui, a aritmética sendo usada é de um tipo especial chamado "*aritmética de ponteiro*". Isso significa que, uma vez que estamos falando de uma matriz de inteiros, o endereço apontado (ou seja, o valor) por "**X + col + 1**" deve ser maior do que o endereço "**X + col**" por uma quantidade igual a "**sizeof(int)**".

Já que sabemos a disposição de memória para arranjos bidimensionais, podemos determinar que na expressão "**multi + row**" indicada, "**multi + row + 1**" deve aumentar o valor em certa quantidade igual ao necessário para "*apontar para*" a próxima linha, o que neste caso seria uma quantidade igual a "**COLS * sizeof(int)**".

Isso significa que se a expressão "***(*(multi + row) + col)**" deve ser avaliada corretamente em tempo de execução. O compilador deve gerar código que leve em consideração o valor de **COLS**, ou seja, a segunda dimensão. Devido à equivalência das duas formas de expressão, isso é verdadeiro tanto se estivermos usando a expressão de ponteiro como aqui ou a expressão de arranjo "**multi [row] [col]**".

Assim, para avaliar qualquer uma das expressões, um total de 5 valores devem ser conhecidos:

1. O endereço do primeiro elemento da matriz, que é retornado pela expressão "**multi**", ou seja, o nome da matriz.
2. O tamanho do tipo de elementos da matriz, neste caso "**sizeof(int)** ".
3. A segunda dimensão da matriz.

4. O valor específico do índice para a primeira dimensão, "**row**" neste caso a linha.
5. O valor específico do índice para a segunda dimensão, "**col**" neste caso a coluna.

Dado tudo isso, considere o problema de projetar uma função para manipular os valores dos elementos de uma matriz previamente declarada. Por exemplo, uma função que definiria todos os elementos da matriz "**multi**" com o valor "**1**".

```
void set_value(int m_array[][COLS])
{
  int row, col;
  for (row = 0; row < ROWS; row++)
  {
    for (col = 0; col < COLS; col++)
      m_array[row][col] = 1;
  }
}
```

E para chamar essa função, usaríamos:

```
set_value(multi);
```

Agora, dentro da função, usamos os valores definidos por **ROWS** e **COLS** que definem os limites dos laços "**for**". Mas esses "**#defines**" são apenas constantes quanto ao compilador, isto é, não há nada que os conecte ao tamanho do arranjo dentro da função. *Row* e *col* são variáveis locais, é claro. A definição do parâmetro formal permite ao compilador determinar as características associadas ao valor do ponteiro que será passado em tempo de execução. Na verdade, não precisamos da primeira dimensão e, como será visto mais tarde, há ocasiões em que preferimos não a estabelece-la na definição do parâmetro. Por hábito ou consistência, não a usei aqui. Mas, a segunda dimensão deve ser usada, como foi mostrado na expressão do parâmetro. A razão é que precisamos disso na avaliação de "**m_array[row][col]** ", como descrito anteriormente. Embora o parâmetro defina o tipo de dados ("**int**" nesse caso) e as variáveis automáticas para **row** e **col** sejam definidas nos laços "**for**", apenas um valor pode ser passado usando um único parâmetro. Nesse caso, esse é o valor de "**multi**", como observado na instrução de chamada, ou seja, o endereço do primeiro elemento, frequentemente chamado de ponteiro para o arranjo. Portanto, a única maneira que temos de informar ao compilador a segunda dimensão é incluí-la explicitamente na definição do parâmetro.

Na verdade, em geral, todas as dimensões de ordem superior a um são necessárias ao lidar com matrizes multidimensionais. Ou seja, se estivermos falando de matrizes tridimensionais, *a segunda e a terceira dimensão* devem ser especificadas na definição do parâmetro.

CAPÍTULO 8: Ponteiros para arranjos

Ponteiros, obviamente, podem "*apontar para*" para qualquer tipo de objeto de dados, incluindo arranjos. Embora isso tenha sido evidente quando discutimos o programa 3.1, é importante expandir sobre como fazemos isso quando se trata de matrizes multi-dimensionais.

Para revisar, no Capítulo 2, afirmamos que, dado um arranjo de inteiros, poderíamos apontar um ponteiro de inteiro para esse arranjo usando a seguinte expressão:

```
int *ptr;
ptr = &my_array[0];     /* aponte nosso ponteiro para o
                           primeiro inteiro em nossa matriz */
```

Como afirmamos anteriormente, o tipo de dado indicado em variável ponteiro deve corresponder ao tipo de dado do primeiro elemento do arranjo.

Além disso, podemos usar um ponteiro como um parâmetro formal de uma sub-rotina (procedimento/função) que é projetada para manipular uma matriz. Por exemplo.

Dado:

```
int array[3] = {'1', '5', '7'};
void a_func(int *p);
```

Alguns programadores preferem escrever o protótipo da sub-rotina como:

```
void a_func(int p[]);
```

o que tende a informar a outros que podem usar esta sub-rotina que o procedimento projetado para manipular os elementos de uma matriz. Claro que, em ambos os casos, o que realmente é passado é o valor de um ponteiro para o primeiro elemento da matriz, independente de qual notação é usada no protótipo ou definição da sub-rotina. Observe que, se a notação da matriz for usada, não há necessidade de passar a dimensão real da matriz, já que não estamos passando toda a matriz, apenas o endereço do primeiro elemento.

Agora vamos abordar o problema do arranjo bidimensional. Como dito no último capítulo, a linguagem C interpreta um arranjo bidimensional como um arranjo de arranjos unidimensionais. Sendo assim, o primeiro elemento de um arranjo bidimensional de inteiros é um arranjo unidimensional de inteiros. E um ponteiro para um arranjo bidimensional de inteiros deve ser um ponteiro para esse tipo de dado. Uma forma de fazer isso é por meio do uso da palavra-chave "*typedef*", em que "**typedef**" atribui um novo nome a um tipo de dado especificado. Por exemplo:

```
typedef unsigned char byte;
```

isso faz com que o nome "**byte**" signifique o tipo "**unsigned char**". Portanto

```
byte b[10];       seria uma matriz de caracteres não sinalizados.
```

Observe que na declaração do *typedef*, a palavra "**byte**" substitui o que normalmente seria o nome do nosso "**unsigned char**". Ou seja, a regra para usar o "**typedef**" é que o novo nome para o tipo de dados é o nome usado na definição do tipo de dados. Portanto, em:

```
typedef int Array[10];
```

"*Array*" se torna um tipo de dado para um arranjo de 10 inteiros. Ou seja, "**Array my_arr;**" declara "**my_arr**" como um arranjo de 10 inteiros e "**Array arr2d[5];**" torna "**arr2d**" um arranjo de 5 arranjos com *10 inteiros* cada.

Também observe que "**Array *p1d;** " faz de "**p1d**" um ponteiro para um arranjo de *10 inteiros*. Como "***p1d**" aponta para o mesmo tipo que "**arr2d**". Atribuir o endereço do arranjo bidimensional "**arr2d**" a "**p1d**" para um arranjo unidimensional de *10 inteiros* é aceitável. Ou seja, "**p1d = &arr2d[0];** " ou "**p1d = arr2d**; " são ambos corretos.

Uma vez que o tipo de dado que usamos para nosso ponteiro é um arranjo de *10 inteiros*, esperaríamos que incrementar "**p1d**" em "**1**" mudaria seu valor por "**10*sizeof(int)** ", o que efetivamente acontece. Ou seja, "**sizeof(*p1d)**" é "**20**". Você pode provar isso escrevendo e executando um programa simples e curto.

Agora, enquanto o uso de "**typedef**" torna as coisas mais claras para o leitor e mais fáceis para o programador, isso não é realmente necessário. O que precisamos é de uma maneira de declarar um ponteiro como "**p1d**" sem a necessidade da palavra-chave "**typedef**". Acontece que isso pode ser feito como

```
int (*p1d)[10];
```

sendo essa declaração apropriada, ou seja, "**p1d**" é um ponteiro para um arranjo de *10 inteiros*, assim como era na declaração usando o tipo "**Array**". Note que isso é diferente de

```
int *p1d[10];
```

o que faria de "**p1d**" o nome de um arranjo de *10 ponteiros* para o tipo "**int**".

CAPÍTULO 9: Ponteiros e alocação dinâmica de memória

Há momentos em que é conveniente alocar memória em tempo de execução usando "**malloc()**","**calloc()**" ou outras funções de alocação. Usando essa abordagem, é possível adiar a decisão sobre o tamanho do bloco de memória necessário para armazenar um arranjo, por exemplo, até o momento da execução. Ou ele permite usar uma seção de memória para o armazenamento de um arranjo de inteiros em um determinado momento e, em seguida, quando essa memória não é mais necessária, ela pode ser liberada para outros usos, como o armazenamento de um arranjo de estruturas.

Quando a memória é alocada, a função de alocação (como "**malloc()**","**calloc()**", etc.) retorna um ponteiro. O tipo desse ponteiro depende se você está usando um compilador K&R antigo ou o compilador ANSI mais recente. Com o compilador antigo, o tipo do ponteiro retornado é "**char**", com o compilador ANSI é "**void**".

Se você estiver usando um compilador mais antigo e desejar alocar memória para uma matriz de inteiros, será necessário converter o ponteiro "**char**" retornado em um ponteiro "**int**". Por exemplo, para alocar espaço para *10 inteiros*, poderíamos escrever:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)

{ .. ROTINA DE ERRO VAI AQUI .. }
```

Se você estiver usando um compilador compatível com ANSI, "**malloc()**" retorna um ponteiro "**void**" e, como um ponteiro "**void**" pode ser atribuído a uma variável ponteiro de qualquer tipo de objeto, a conversão forçada **(int *)** mostrado acima não é necessária. A dimensão do arranjo pode ser determinada em tempo de execução e não é necessária em tempo de compilação. Ou seja, o número "**10**" indicado poderia ser uma variável lida de um arquivo de dados ou teclado, ou calculada com base em alguma necessidade, em tempo de execução.

Devido à equivalência entre a notação de arranjo e ponteiro, uma vez que "**iptr**" tenha sido atribuído como indicado, pode-se usar a notação de arranjo. Por exemplo, pode-se escrever:

```
int k;
for (k = 0; k < 10; k++)
  iptr[k] = 2;
```

para definir os valores de todos os elementos como "**2**".

Mesmo com uma compreensão razoavelmente boa de ponteiros e matrizes, um lugar onde o iniciante em C provavelmente irá tropeçar no início é na alocação dinâmica de matrizes multidimensionais. Em geral, gostaríamos de ser capazes de acessar elementos de tais matrizes usando a notação de matriz, não a notação de ponteiro, sempre que possível. Dependendo da aplicação, podemos ou não saber ambas as dimensões em tempo de compilação. Isso leva a uma variedade de maneiras de abordar nossa tarefa.

Como vimos, ao alocar dinamicamente um arranjo unidimensional, sua dimensão pode ser determinada em tempo de execução. Agora, ao usar alocação dinâmica de arranjos de ordem superior, nunca precisaremos saber a primeira dimensão em tempo de compilação. Aqui, discutirei alguns métodos de alocação dinamicamente de espaço para a definição de arranjos bidimensionais de inteiros.

Primeiro, vamos considerar casos em que a segunda dimensão é conhecida em tempo de compilação.

MÉTODO 1:

Uma maneira de lidar com o problema é por meio do uso da palavra-chave "**typedef**". Para alocar uma matriz bidimensional de inteiros lembre-se de que ambas notações a seguir resultam no mesmo código de objeto sendo gerado:

```
multi[row][col] = 1;   | ou |   *(*(multi + row) + col) = 1;
```

É também verdade que ambas notações seguintes geram o mesmo código:

```
multi[row]   | ou |   *(multi + row)
```

Já que a notação à direita avalia para um ponteiro, a notação de arranjo à esquerda também avalia para um ponteiro. De fato, "**multi[0]**" retornará um ponteiro para o primeiro inteiro na primeira linha, "**multi[1]**" um ponteiro para o primeiro inteiro da segunda linha, etc. Na verdade, "**multi[n]**" avalia para um ponteiro para esse arranjo de inteiros que compõem a n-ésima linha do nosso arranjo bidimensional. Ou seja, "**multi**" pode ser considerado como um arranjo de arranjos e "**multi[n]**" como um ponteiro para o n-ésimo arranjo desse arranjo de arranjos. Aqui, a palavra *ponteiro* está sendo usada para representar um valor de endereço. Embora tal uso seja comum na literatura, ao ler tais declarações, é preciso ter cuidado em distinguir entre o endereço constante de um arranjo e uma variável de ponteiro que é um objeto de dado em si.

Agora considere o programa 9.1 (prog0901.c):

```
/* Programa 9.1 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 02/05/2023 */
/*                                  youtube.com/@AM-42 */

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

#define COLS 5

typedef int RowArray[COLS];
RowArray *rptr;

int main(void)
{
  int nrows = 10;
  int row, col;
```

```
  rptr = malloc(nrows * COLS * sizeof(int));
  for (row = 0; row < nrows; row++)
  {
    for (col = 0; col < COLS; col++)
    {
      rptr[row][col] = 17;
    }
  }

  return 0;
}
```

Aqui assumi um compilador padrão ANSI, então a conversão "**void**" no ponteiro retornado pelo **malloc()** não é necessária. Se você estiver usando um compilador padrão K&R mais antigo, precisará fazer o cast usando:

```
rptr = (RowArray *)malloc(..., etc.
```

Usando essa abordagem, "**rptr**" tem todas as características de um nome de arranjo (exceto que "**rptr**" é modificável), e a notação de arranjo pode ser usada em todo o restante do programa. Isso também significa que, se você pretende escrever uma função para modificar o conteúdo do arranjo, deve usar "**COLS**" como parte do parâmetro formal nessa sub-rotina, assim como feito ao discutir a passagem de matrizes bidimensionais para uma sub-rotina.

MÉTODO 2:

No *MÉTODO 1* "**rptr**" ficou sendo um ponteiro do tipo "*vetor unidimensional de inteiros de COLS*". Acontece que existe uma sintaxe que pode ser usada para esse tipo sem a necessidade de "**typedef**". Se escrevermos:

```
int (*xptr)[COLS];
```

a variável "**xptr**" terá todas as mesmas características que a variável "**rptr**" no *MÉTODO 1* e não será preciso usar a palavra-chave "**typedef**". Aqui, "**xptr**" é um ponteiro para uma matriz de inteiros e o tamanho dessa matriz é dado por "**#define COLS**". A colocação dos parênteses faz com que a notação de ponteiro predomine, mesmo que a notação de matriz tenha precedência mais alta, isto é, se tivéssemos escrito

```
int *xptr[COLS];
```

definiríamos "**xptr**" como um arranjo de ponteiros contendo o número de ponteiros igual ao definido por "**#define COLS**". Isso não é a mesma coisa. No entanto, arranjos de ponteiros têm sua utilidade na alocação dinâmica de arranjos bidimensionais, como será visto nos próximos dois métodos.

MÉTODO 3:

Considere o caso em que não sabemos o número de elementos em cada linha em tempo de compilação, ou seja, tanto o número de linhas quanto o número de colunas devem ser determinados em tempo de execução. Uma maneira de fazer isso seria criar um arranjo de

ponteiros para o tipo "**int**" e, em seguida, alocar espaço para cada linha e apontar esses ponteiros para cada linha. Considere então o programa 9.2 (prog0902.c):

```
/* Programa 9.2 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 02/05/2023 */
/*                                youtube.com/@AM-42 */

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

int main(void)
{
  int nrows = 5;  /* Ambas as linhas e colunas podem ser avaliadas */
  int ncols = 10; /* ou lidas em tempo de execucao */
  int row;
  int **rowptr;
  rowptr = malloc(nrows * sizeof(int *));
  if (rowptr == NULL)
  {
    puts("\nFalha ao alocar espaco para ponteiros de linha.\n");
    exit(0);
  }

  printf("\n\n\nIndice Pont.(hex)       Pont.(dec)  Difereca.(dec)");

  for (row = 0; row < nrows; row++)
  {
    rowptr[row] = malloc(ncols * sizeof(int));
    if (rowptr[row] == NULL)
    {
      printf("\nFalha ao alocar para linha[%d]\n", row);
      exit(0);
    }
    printf("\n%d      %p %d", row, rowptr[row], rowptr[row]);
    if (row > 0)
      printf("    %d", (int)(rowptr[row] - rowptr[row-1]));
  }

  return 0;
}
```

No código anterior, "**rowptr**" é um ponteiro para ponteiro do tipo "**int**". Nesse caso, ele aponta para o primeiro elemento de um arranjo de ponteiros do tipo "**int**". Considere o número de chamadas para "**malloc()**":

```
Para obter o array de ponteiros      1   chamada
Para obter espaço para as linhas     5   chamadas
                                   -----
                     Total           6   chamadas
```

Se você escolher esta abordagem, note que, embora possa usar a notação de arranjo para acessar elementos individuais do arranjo, por exemplo, "**rowptr[row][col] = 17;** ", não significará que os dados na "*matriz bidimensional*" são contíguos na memória.

Você pode, no entanto, usar a notação de arranjo como se fosse um bloco contínuo de memória. Por exemplo, você pode escrever:

```
rowptr[row][col] = 176;
```

como se "**rowptr**" fosse o nome de uma matriz bidimensional criada em tempo de compilação. Claro que a linha e a coluna devem estar dentro dos limites da matriz que você criou, assim como em uma matriz criada em tempo de compilação.

Se você deseja ter um bloco contíguo de memória dedicado ao armazenamento dos elementos na matriz, pode fazê-lo da seguinte maneira:

MÉTODO 4:

Neste método, alocamos primeiro um bloco de memória para conter todo o arranjo. Em seguida, criamos um arranjo de ponteiros para apontar para cada linha. Assim, mesmo que o arranjo de ponteiros esteja sendo usado, o arranjo real na memória é contíguo. O código do programa 9.3 (prog0903.c) seguinte fica assim:

```
/* Programa 9.3 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 02/05/2023 */
/*                                youtube.com/@AM-42 */

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

int main(void)
{
  int **rptr;
  int *aptr;
  int *testptr;
  int k;
  int nrows = 5; /* Ambas as linhas e colunas podem ser avaliadas */
  int ncols = 8; /* ou lidas em tempo de execucao */
  int row, col;

  /* agora alocamos a memória para o arranjo */

  aptr = malloc(nrows * ncols * sizeof(int));
  if (aptr == NULL)
  {
    puts("\nFalha ao alocar espaco para o arranjo");
    exit(0);
  }

  /* em seguida, alocamos espaco para os ponteiros das linhas */

  rptr = malloc(nrows * sizeof(int*));
  if (rptr == NULL)
  {
    puts("\nFalha ao alocar espaco para ponteiros");
```

```
    exit(0);
  }

  /* e agora os ponteiros sao "apontados" */

  for (k = 0; k < nrows; k++)
  {
    rptr[k] = aptr + (k * ncols);
  }

  /* Agora ilustramos como os ponteiros de linha sao incrementados */

  printf("\n\nIlustrando como os ponteiros de linha sao incrementados ");
  printf("\n\nIndice Point.(hex)      Diferenca.(dec)");

  for (row = 0; row < nrows; row++)
  {
    printf("\n%d      %p", row, rptr[row]);
    if (row > 0)
      printf(" %d", (rptr[row] - rptr[row - 1]));
  }

  printf("\n\nE agora imprimimos o arranjo\n");
  for (row = 0; row < nrows; row++)
  {
    for (col = 0; col < ncols; col++)
    {
      rptr[row][col] = row + col;
      printf("%2d ", rptr[row][col]);
    }

    putchar('\n');
  }

  puts("\n");

  /* e aqui ilustramos que estamos, de fato, lidando com uma matriz */
  /* bidimensional em um bloco contiguo de memoria.                 */

  printf("E agora demonstramos que eles sao contiguos na memoria \n");

  testptr = aptr;
  for (row = 0; row < nrows; row++)
  {
    for (col = 0; col < ncols; col++)
    {
      printf("%2d ", *(testptr++));
    }

    putchar('\n');
  }

  return 0;
}
```

Considere novamente, o número de chamadas para "**malloc()**":

```
Para obter espaço para o próprio arranjo        1   chamada
Para obter espaço para a matriz de ponteiros    1   chamada
                                              -----
                                Total           2   chamadas
```

Agora, cada chamada para "**malloc()**" cria uma sobrecarga de espaço adicional, já que "**malloc()**" é geralmente implementado pelo sistema operacional formando uma lista ligada que contém dados sobre o tamanho do bloco. No entanto, mais importante do que isso, com matrizes grandes (várias centenas de linhas), manter o controle do que precisa ser liberado quando chegar a hora pode ser mais complicado. Isso, combinado com a continuidade do bloco de dados que permite a inicialização de todos os zeros usando "**memset()**", parece tornar a segunda alternativa a preferida.

Como exemplo final sobre matrizes multidimensionais, ilustraremos a alocação dinâmica de uma matriz tridimensional. Este exemplo irá ilustrar mais uma coisa a se observar ao fazer esse tipo de alocação. Por razões citadas anteriormente, usaremos a abordagem descrita na alternativa dois. Considere o seguinte código para o programa 9.4 (prog0904.c):

```
/* Programa 9.4 extraído de PTRTUT10.TXT - 13/06/1997 */
/*              adaptado por AM-42       - 02/05/2023 */
/*                                  youtube.com/@AM-42 */

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

int X_DIM = 16;
int Y_DIM = 5;
int Z_DIM = 3;

int main(void)
{
  char *space;
  char ***Arr3D;
  int y, z;
  ptrdiff_t diff;

  /* primeiro reservamos espaco para o proprio arranjo */

  space = malloc(X_DIM * Y_DIM * Z_DIM * sizeof(char));

  /* em seguida, alocamos espaco para uma matriz de ponteiros,
     cada um para eventualmente apontar para o primeiro elemento
     de uma matriz bidimensional de ponteiros para ponteiros */

  Arr3D = malloc(Z_DIM * sizeof(char **));

  /* e para cada um deles atribuimos um ponteiro a um arranjo
     recem alocado de ponteiros para uma linha */

  for (z = 0; z < Z_DIM; z++)
```

```
  {
    Arr3D[z] = malloc(Y_DIM * sizeof(char*));

    /* e para cada espaco neste arranjo colocamos um ponteiro para o
       primeiro elemento de cada linha no espaco do arranjo
       originalmente alocado */

    for (y = 0; y < Y_DIM; y++)
    {
      Arr3D[z][y] = space + (z * (X_DIM *Y_DIM) + y * X_DIM);
    }
  }

  /* E, agora, verificamos cada endereco em nossa matriz 3D para ver
     se a indexacao do ponteiro Arr3d conduz dados de forma continua */

  for (z = 0; z < Z_DIM; z++)
  {
    printf("Locacao do arranjo %d e' %p\n", z, *Arr3D[z]);
    for (y = 0; y < Y_DIM; y++)
    {
      printf(" Arranjo %d com linha %d em %p", z, y, Arr3D[z][y]);
      diff = Arr3D[z][y] - space;
      printf(" diferenca = %3d", diff);
      printf(" z = %d y = %d\n", z, y);
    }
  }

  return 0;
}
```

Se você seguiu este tutorial até este ponto, não terá problema em decifrar o código indicado anteriormente com base apenas em seus comentários. No entanto, há alguns pontos que devem ser destacados. Vamos começar com a linha que diz:

```
Arr3D[z][y] = space + (z * (X_DIM * Y_DIM) + y * X_DIM);
```

Observe que "**space**" é um ponteiro de caracteres, que é do mesmo tipo que "**Arr3D[z][y]**". É importante que, ao adicionar um inteiro, como o obtido pela avaliação da expressão "**(z * (X_DIM * Y_DIM) + y * X_DIM)**" a um ponteiro, o resultado seja um novo valor de ponteiro. E ao atribuir valores de ponteiro a variáveis de ponteiro, os tipos de dados do valor e da variável devem ser compatíveis.

CAPÍTULO 10: Ponteiros e sub-rotinas

Até agora, estivemos discutindo ponteiros para objetos de dados. C também permite a declaração de ponteiros para sub-rotinas (tanto tratadas como procedimento, quanto para funções[12]). Ponteiros para sub-rotinas possuem uma variedade de usos, dos quais alguns são aqui discutidos.

Considere o seguinte problema real. Você quer escrever uma sub-rotina capaz de classificar virtualmente qualquer coleção de dados que possa ser armazenada em um arranjo. Isso pode ser um arranjo de cadeias (**char[n]**), inteiros (**int**), flutuantes (**float** ou **double**) ou até mesmo estruturas (**struct**). O algoritmo de classificação pode ser o mesmo para todos. Por exemplo, pode ser um algoritmo como o *bubble sort* ou um algoritmo mais complexo como o *shell sort* ou *quick sort*. Usaremos um *bubble sort* para fins de demonstração.

Sedgewick [1] descreveu o *bubble sort* usando código em C, criando uma sub-rotina em esatilo procedimento que ao receber um ponteiro para o arranjo o classifica. Se considerarmos essa sub-rotina com nome "**bubble()**" podemos ter o programa "bubble_1.c", descrito a seguir:

```
/* Programa 'bubble_1.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h>

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

void bubble(int a[], int N);

int main(void)
{
  int i;
  putchar('\n');
  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr, 10);
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  return 0;
}

void bubble(int a[], int N)
{
```

[12] NT: Observação indicada pelo tradutor.

```
  int i, j, t;
  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)

    {
      if (a[j - 1] > a[j])
      {
        t = a[j - 1];
        a[j - 1] = a[j];
        a[j] = t;
      }
    }
  }
}
```

O *bubble sort* é um dos algoritmos de classificação mais simples eistentes. Esse algoritmo percorre o arranjo do segundo ao último elemento, comparando cada elemento com o que o precede. Se o elemento que precede for maior do que o elemento atual, os dois são trocados, de forma que o maior esteja mais próximo do final do arranjo. Na primeira passagem, isso resulta no maior elemento terminando no final do arranjo. O arranjo é agora limitado a todos os elementos, exceto o último, e aquele que o processo é repetido. Isso coloca o próximo maior elemento em um ponto anterior ao maior elemento. O processo é repetido um número de vezes igual ao número de elementos *menos 1*. O resultado final é um arranjo classificado.

Aqui nossa sub-rotina é projetada para classificar um arranjo de inteiros. Portanto, na linha 1, estamos comparando inteiros e, nas linhas 2 a 4, estamos usando armazenamento temporário de inteiro para armazenar inteiros. O que queremos fazer agora é ver se podemos converter este código para que possamos usar qualquer tipo de dado, ou seja, não ser restrito a inteiros.

Ao mesmo tempo, não queremos ter que analisar nosso algoritmo e o código associado a ele toda vez que o usarmos. Começamos removendo a comparação de dentro da sub-rotina de classificação "**bubble()**", para facilitar a modificação da sub-rotina de comparação sem ter que reescrever partes relacionadas ao algoritmo real. Isso resulta no código "bubble_2.c":

```
/* Programa 'bubble_2.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

/* Separando a sub-rotina de comparacao */

#include <stdio.h>

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

void bubble(int a[], int N);
int compare(int m, int n);

int main(void)
{
```

```
  int i;
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  bubble(arr, 10);
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  return 0;
}

void bubble(int a[], int N)

{
  int i, j, t;
  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (compare(a[j - 1], a[j]))
      {
        t = a[j - 1];
        a[j - 1] = a[j];
        a[j] = t;
      }
    }
  }
}

int compare(int m, int n)
{
  return (m > n);
}
```

Se o nosso objetivo é tornar nossa rotina de classificação independente do tipo de dados, uma maneira de fazer isso é usar ponteiros do tipo "**void**" para apontar para os dados em vez de usar o tipo de dados inteiro. Como um começo nessa direção, vamos modificar algumas coisas do programa anterior para que ponteiros possam ser usados. Para começar, vamos manter ponteiros do tipo inteiro no código "bubble_3.c":

```
/* Programa 'bubble_3.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h>

int arr[10] = {3,6,1,2,3,8,4,1,7,2};
```

```
void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
  int i;
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  bubble(arr, 10);
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  return 0;
}

void bubble(int *p, int N)
{
  int i, j, t;
  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (compare(&p[j - 1], &p[j]))
      {
        t = p[j - 1];
        p[j - 1] = p[j];
        p[j] = t;
      }
    }
  }
}

int compare(int *m, int *n)
{
  return (*m > *n);
}
```

Veja as mudanças. Agora estamos passando um ponteiro para um inteiro (ou um arranjo de inteiros) para a sub-rotina "**bubble()**". E dentro de *bubble* estamos passando ponteiros para os elementos do arranjo que queremos comparar à nossa função de comparação. E, é claro, estamos referenciando esses ponteiros na nossa sub-rotina "**compare()**" para fazer a comparação real. Nosso próximo passo será converter os ponteiros de "**bubble()**" em ponteiros do tipo "**void**" para que essa sub-rotina se torne mais insensível ao tipo. Isso é mostrado junto ao programa "bubble_4.c":

```c
/* Programa 'bubble_4.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h>

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

void bubble(int *p, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  bubble(arr, 10);
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  return 0;
}

void bubble(int *p, int N)
{
  int i, j, t;
  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (compare((void*) &p[j - 1], (void*) &p[j]))
      {
        t = p[j - 1];
        p[j - 1] = p[j];
        p[j] = t;
      }
    }
  }
}

int compare(void *m, void *n)
{
  int *m1, *n1;
  m1 = (int*) m;
```

```
  n1 = (int*) n;
  return (*m1 > *n1);
}
```

Observe que, ao fazer isso, em "**compare()**" tivemos que introduzir a conversão dos tipos de ponteiros "**void**" passados para o tipo real que está sendo classificado. Mas, como veremos mais tarde, isso está correto. E, como o que está sendo passado para "**bubble()**" ainda é um ponteiro para um arranjo de inteiros, tivemos que converter esses ponteiros em ponteiros para "**void**" ao passá-los como parâmetros em nossa chamada para "**compare()**".

Agora abordamos o problema do que passamos para "**bubble()**". Queremos que o primeiro parâmetro dessa função também seja um ponteiro "**void**". Mas isso significa que, dentro de "**bubble()**", precisamos fazer algo com a variável "**t**", que atualmente é um inteiro. Além disso, onde usamos "**t = p[j - 1];**" o tipo de "**p[j - 1]**" precisa ser conhecido para saber quantos bytes copiar para a variável t (ou qualquer que seja a variável que substituirá t).

Atualmente, no programa "*bubble_4.c*", o conhecimento dentro de "**bubble()**" sobre o tipo de dados sendo classificados (e, portanto, o tamanho de cada elemento individual) é obtido do fato de que o primeiro parâmetro é um ponteiro para inteiro. Se quisermos ser capazes de usar "**bubble()**" para classificar qualquer tipo de dados, precisamos tornar esse ponteiro um ponteiro para "**void**". Mas, ao fazer isso, vamos perder informações sobre o tamanho dos elementos individuais dentro do arranjo. Então, em "*bubble_5.c*", adicionaremos um parâmetro separado para lidar com essa informação de tamanho.

Essas mudanças, de "*bubble4.c*" para "*bubble5.c*", são talvez um pouco mais extensas do que as que fizemos anteriormente. Então, compare cuidadosamente os dois módulos em busca de diferenças. Veja o código para o programa "bubble_5.c":

```
/* Programa 'bubble_5.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h>
#include <string.h>

long arr[10] = {3,6,1,2,3,8,4,1,7,2};

void bubble(void *p, size_t width, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');

  for (i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }

  bubble(arr, sizeof(long), 10);
  putchar('\n');
```

```
  for (i = 0; i < 10; i++)
  {
    printf("%ld ", arr[i]);
  }

  return 0;
}

void bubble(void *p, size_t width, int N)
{
  int i, j;
  unsigned char buf[4];
  unsigned char *bp = p;

  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (compare((void*)(bp + width *(j - 1)),
          (void*)(bp + j * width))) /* 1 */
      {
        /*  t = p[j - 1];     */
        memcpy(buf, bp + width * (j - 1), width);
        /*  p[j - 1] = p[j];  */
        memcpy(bp + width *(j - 1), bp + j * width, width);
        /*  p[j] = t;         */
        memcpy(bp + j * width, buf, width);
      }
    }
  }
}

int compare(void *m, void *n)
{
  long *m1, *n1;
  m1 = (long*) m;
  n1 = (long*) n;
  return (*m1 > *n1);
}
```

Observe que eu mudei o tipo de dados do arranjo de "**int**" para "**long**" para ilustrar as mudanças que são necessárias na função "**compare()**". Dentro de "**bubble()**", eu eliminei a variável "**t**" (que teríamos que mudar de "**int**" para "**long**"). Eu adicionei um *buffer* de *4 caracteres sem sinal*, que é o tamanho necessário para armazenar um "**long**" (isso mudará novamente em modificações futuras deste código). O ponteiro de caractere sem sinal "***bp**" é usado para apontar para a base do arranjo a ser classificado, ou seja, para o primeiro elemento desse arranjo.

Também tivemos que modificar o que passamos para "**compare()**" e como fazemos a troca de elementos indicados pela comparação. O uso de "**memcpy()**" e notação de ponteiro em vez de notação de matriz trabalham para essa redução de sensibilidade ao tipo.

Novamente, comparar cuidadosamente o código "*bubble5.c*" com o "*bubble4.c*" pode resultar em uma melhor compreensão do que está acontecendo e por quê.

Nós agora vamos para o "*bubble6.c*" onde usamos a mesma função "**bubble()**" que usamos no "*bubble5.c*" para classificar cadeias em vez de "*long integers*". Claro que temos que mudar a função de comparação, já que a forma de comparar cadeias é diferente da forma de comparar *long integers*. E no programa "*bubble6.c*", nós deletamos as linhas dentro da subrotina "**bubble()**" que estavam comentadas no "*bubble5.c*". Veja o código para o programa "bubble_6.c":

```
/* Programa 'bubble_6.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h >
#include <string.h>

#define MAX_BUF 256

char arr2[7][20] = { "Mickey Mouse",
                     "Donald Duck",
                     "Minnie Mouse",
                     "Goofy",
                     "Ted Jensen",
                     "Jay Flaherty",
                     "Augusto Manzano"
                   };

void bubble(void *p, int width, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');

  for (i = 0; i < 7; i++)
  {
    printf("%s\n", arr2[i]);
  }

  bubble(arr2, 20, 7);
  putchar('\n\n');

  for (i = 0; i < 7; i++)
  {
    printf("%s\n", arr2[i]);
  }

  return 0;
}

void bubble(void *p, int width, int N)
{
```

```
  int i, j, k;
  unsigned char buf[MAX_BUF];
  unsigned char *bp = p;

  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      k = compare((void*)(bp + width * (j - 1)),
          (void*)(bp + j * width));
      if (k > 0)
      {
        memcpy(buf, bp + width *(j - 1), width);
        memcpy(bp + width *(j - 1), bp + j *width, width);
        memcpy(bp + j *width, buf, width);
      }
    }
  }
}

int compare(void *m, void *n)
{
  char *m1 = m;
  char *n1 = n;
  return (strcmp(m1, n1));
}
```

Mas o fato de "**bubble()**" permanecer inalterado em relação à "*bubble5.c*" indica que essa sub-rotina é capaz de classificar uma ampla variedade de tipos de dados. O que resta a fazer é passar para "**bubble()**" o nome da sub-rotina de comparação que queremos usar, para que ela possa ser verdadeiramente universal. Assim como o nome de um arranjo é o endereço do primeiro elemento do arranjo no segmento de dados, o nome de uma sub-rotina decai no endereço dessa sub-rotina no segmento de código. Portanto, precisamos usar um ponteiro para uma função. Neste caso, a função de comparação.

Ponteiros para sub-rotinas devem corresponder às sub-rotinas apontadas no número e nos tipos dos parâmetros e também no tipo do valor de retorno. No nosso caso, declaramos nosso ponteiro de sub-rotina como:

```
int (*fptr)(const void *p1, const void *p2);
```

Observe que se escrevêssemos:

```
int *fptr(const void *p1, const void *p2);
```

teríamos um protótipo de função que retorna um ponteiro para o tipo "**int**". Isso ocorre porque em C o operador de parênteses "**()**" tem uma precedência mais alta do que o operador de ponteiro "*****". Ao colocar os parênteses ao redor da cadeia "**(*fptr)**" indicamos que estamos declarando um ponteiro para sub-rotina.

Agora modificamos nossa declaração de "**bubble()**" adicionando, como seu quarto parâmetro, um ponteiro de função do tipo apropriado. Seu protótipo de função fica assim:

```
void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
```

Quando chamamos "**bubble()**", inserimos o nome da função de comparação que queremos usar. O programa "*bubble7.c*" ilustra como essa abordagem permite o uso da mesma sub-rotina "**bubble()**" para ordenar diferentes tipos de dados. Veja o código para o programa "bubble_7.c":

```
/* Programa 'bubble_7.c' extraído de PTRTUT10.TXT - 13/06/1997 */
/*                       adaptado por AM-42       - 02/05/2023 */
/*                                              youtube.com/@AM-42 */

#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

long arr[10] = {3,6,1,2,3,8,4,1,7,2};

char arr2[7][20] = { "Mickey Mouse",
                     "Donald Duck",
                     "Minnie Mouse",
                     "Goofy",
                     "Ted Jensen",
                     "Jay Flaherty",
                     "Augusto Manzano"
                   };

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
int compare_string(const void *m, const void *n);
int compare_long(const void *m, const void *n);

int main(void)
{
  int i;
  puts("\nAntes da classificacao:\n");

  for (i = 0; i < 10; i++) /* mostra os inteiros longos */
  {
    printf("%ld ", arr[i]);
  }

  puts("\n");

  for (i = 0; i < 7; i++) /* mostra as cadeias */
  {
    printf("%s\n", arr2[i]);
  }

  bubble(arr, 4, 10, compare_long);    /* classifica inteiros longos */
  bubble(arr2, 20, 7, compare_string); /* classifica cadeias */
  puts("\n\Depois da classificacao:\n");

  for (i = 0; i < 10; i++) /* mostra inteiros longos classificados */
```

```
  {
    printf("%d ", arr[i]);
  }

  puts("\n");

  for (i = 0; i < 7; i++) /* mostra cadeias classificadas */
  {
    printf("%s\n", arr2[i]);
  }

  return 0;
}

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *))
{
  int i, j, k;
  unsigned char buf[MAX_BUF];
  unsigned char *bp = p;

  for (i = N - 1; i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      k = fptr((void*)(bp + width * (j - 1)),
          (void*)(bp + j * width));

      if (k > 0)
      {
        memcpy(buf, bp + width *(j - 1), width);
        memcpy(bp + width *(j - 1), bp + j *width, width);
        memcpy(bp + j *width, buf, width);
      }
    }
  }
}

int compare_string(const void *m, const void *n)
{
  char *m1 = (char*) m;
  char *n1 = (char*) n;
  return (strcmp(m1, n1));
}

int compare_long(const void *m, const void *n)
{
  long *m1, *n1;
  m1 = (long*) m;
  n1 = (long*) n;
  return (*m1 > *n1);
}
```

Referência para o Capítulo 10:

"Algorithms in C" Robert Sedgewick Addison-Wesley ISBN 0-201-51425-7

EPÍLOGO

Escrevi este material para fornecer uma introdução a ponteiros para aqueles que estão começando a aprender a linguagem C. Em C, quanto mais se entende sobre ponteiros, maior flexibilidade se tem na escrita de códigos. O material apresentado expande meu primeiro esforço nesse sentido, que foi intitulado "*ptr_help.txt*" e encontrado em uma versão anterior da coleção de código C de Bob Stout, *SNIPPETS*. O conteúdo nesta versão foi atualizado em relação ao *PTRTUTOT.ZIP* incluído no *SNIP9510.ZIP*.

Estou sempre disposto a aceitar críticas construtivas sobre este material ou pedidos de revisão para adicionar outros assuntos relevantes. Portanto, se você tiver dúvidas, comentários, críticas, etc., sobre o que foi apresentado, eu ficaria muito grato se entrasse em contato comigo por meio do *e-mail* **tjensen@ix.netcom.com**[13].

[13] NT: O e-mail aqui indicado, bem como o site do autor original deste artigo não estão mais acessíveis.