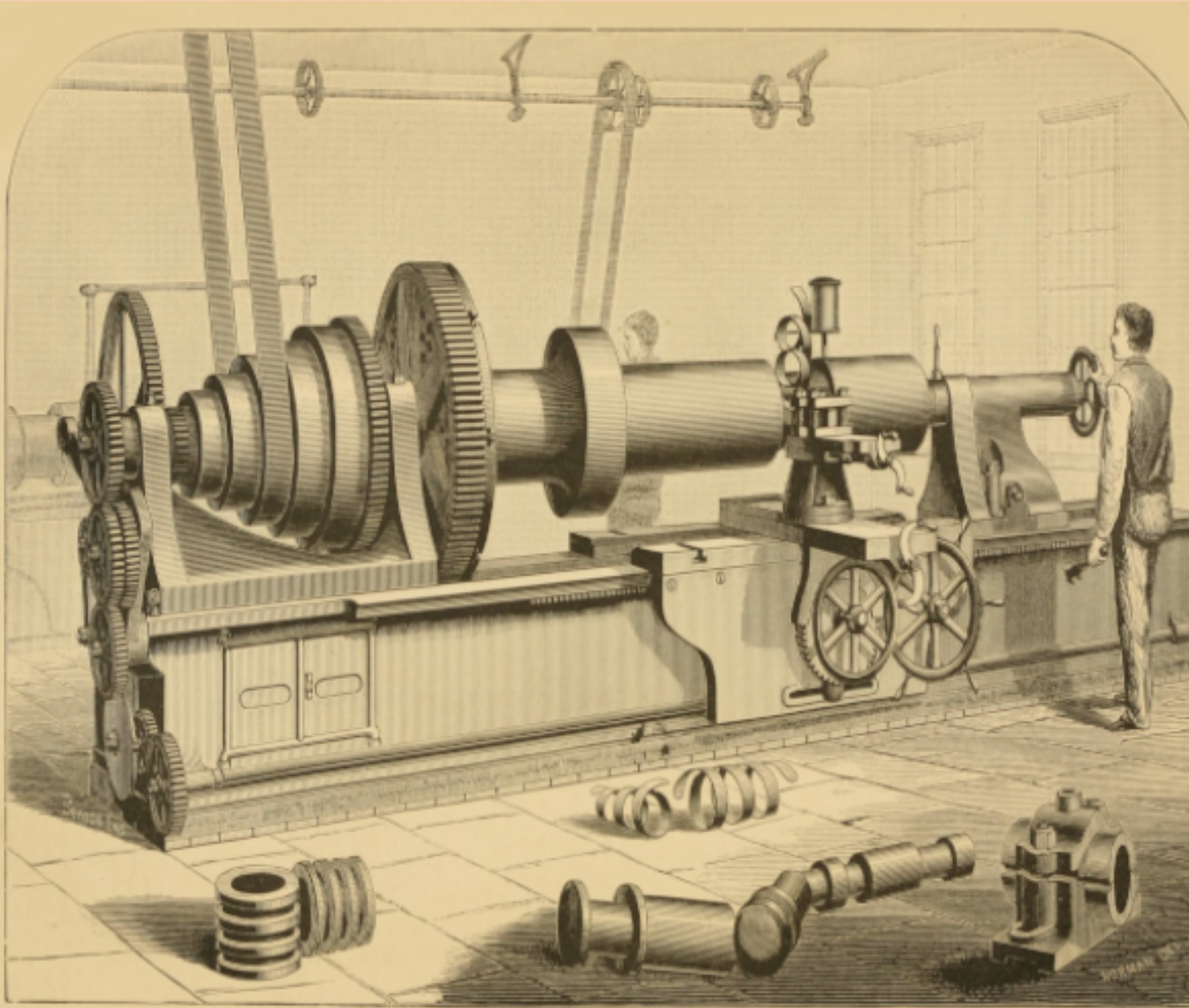
Ponteiros e Arrays em C

Tutorial definitivo

Texto Original: Ted Jensen
Adaptação/atualização: João Araujo



PUBLICADO POR JOÃO ARAUJO

WWW.ARAUJO.ENG.UERJ.BR

Primeira versão, Outubro de 2021

Conteúdo

Prefácio

Este texto nasceu da necessidade de ensinar ponteiros em C. O tópico de ponteiros sempre foi o grande terror de estudantes iniciais da linguagem de programação C e este tema não era bem esclarecido nos principais livros disponíveis. Apesar do medo dos estudantes, o tema não é complexo, mas exige muita atenção e uma compreensão muito boa sobre o hardware e como funcionam os computadores.

Sempre pensei em escrever um texto como esse, mas esta tarefa sempre foi deixada para depois, até que me deparei com o excelente texto de **Ted Jensen**. Felizmente ele colocou o texto em domínio público e assim pude aproveitá-lo sem restrições. No início seria apenas uma tradução, porém a última revisão deste texto foi feita há mais de 20 anos. Na época que Ted escreveu este livro e o disponibilizou gratuitamente, a licença *Creative Commons* estava iniciando. Assim, em vez de distribuir como domínio público, escolhi distribuí-la com uma licença mais moderna que mantém o material disponível para todos e pede apenas uma declaração de atribuição.

Livros excelentes se perdem por questões de copyright. Os autores perdem o interesse no tema (ou coisa pior) e bons textos não podem ser atualizados para aproveitar os avanços na computação.

Antes de disponibilizá-lo tentei contato com Ted, mas todas as tentativas foram infrutíferas. O site original não funciona mais e o e-mail não responde. A última atualização do site foi em 2003 e em 2018 ele saiu do ar.

De qualquer modo, mesmo sem conseguir contactá-lo, deixo aqui meu profundo agradecimento a Ted pela sua generosidade em distribuir material de alta qualidade. A maior parte do texto aqui apresentado é minha tradução do texto original de Ted.

Fiz algumas adaptações para modernizar o texto. Na primeira versão do livro existia apenas o C ANSI, depois tivemos algumas pequenas modificações na linguagem que tentei incorporar ao texto original de Ted.

Chegou um momento que deixou de ser apenas uma tradução, mas uma co-autoria, mas mantenho a gratuidade do texto para todos que queiram utilizá-lo. Assim, se você quiser usar este texto, peço apenas a citação deste material

O arquivo fonte em Latex deste texto pode ser obtido em
`https://github.com/jaraujouerj/Ponteiros-em-C`

e o arquivo em pdf pode ser baixado do meu site pessoal:
`http://araujo.eng.uerj.br`

Sobre mim

Meu nome é João Araujo e sou professor do Departamento de Engenharia de Sistemas e Computação da Faculdade de Engenharia da Universidade do Estado do Rio de Janeiro desde 1990. Trabalho há anos ensinando programação principalmente com a linguagem C. Aprendi C ainda estudante, na UFRJ, nos anos 1980, na versão original do livro K&R e acompanhei a transição de C por todas as suas versões: ANSI, C99 e C11.

Uso deste Material

 Assim como Ted, ‘‘*a única coisa que peço é que se este material for usado como um auxiliar de ensino em uma aula, eu agradeceria se fosse distribuído na íntegra, ou seja, incluindo todos os capítulos, o prefácio e a introdução. Eu também agradeceria se, em tais circunstâncias, o instrutor de tal classe me deixasse uma nota em um dos endereços abaixo informando-me sobre isso. Escrevi isso com a esperança de que seja útil a outras pessoas e, como não estou pedindo nenhuma remuneração financeira, a única maneira de saber que atingi, pelo menos parcialmente, esse objetivo é por meio do feedback de quem considera este material útil.*’

Instrutor ou não, se este material foi útil para você, me envie uma mensagem em araujo@eng.uerj.br. Críticas, correções e sugestões também são bem-vindas.

`João Araujo Ribeiro`
`Departamento de Engenahria de Sistemas e Computação`
`Faculdade de Engenharia`
`Universidade do Estado do Rio de Janeiro`
`Outubro de 2021`

Prefácio original de Ted Jensen

Este documento tem como objetivo apresentar dicas para programadores iniciantes na linguagem de programação C. Ao longo de vários anos lendo e contribuindo para várias conferências sobre C, incluindo as da *FidoNet* e *UseNet*, observei que um grande número de novatos em C parece ter dificuldade em compreender os fundamentos dos ponteiros. Portanto, empreendi a tarefa de tentar explicá-los em linguagem simples, com muitos exemplos.

A primeira versão deste documento foi colocada em domínio público. Ele foi escolhido por *Bob Stout*, que o incluiu como um arquivo chamado **PTR-HELP.TXT** em sua coleção amplamente distribuída de **SNIPPETS**. Desde aquele lançamento original de 1995, adicionei uma quantidade significativa de material e fiz algumas pequenas correções no trabalho original.

Posteriormente, postei uma versão HTML por volta de 1998 em meu site em:
http://pweb.netcom.com/ tjensen/ptr/cpoint.htm

Depois de inúmeros pedidos, finalmente saí com esta versão em PDF que é idêntica à versão em HTML citada acima e que pode ser obtida no mesmo site.

Agradecimentos

Há tantas pessoas que, sem saber, contribuíram para este trabalho por causa das perguntas que colocaram no *FidoNet C Echo*, ou no *UseNet Newsgroup comp.lang.c*, ou em várias outras conferências

em outras redes, que seria impossível listar todas elas. Agradecimentos especiais a *Bob Stout*, que teve a gentileza de incluir a primeira versão deste material em seu arquivo **SNIPPETS**.

Sobre o Autor

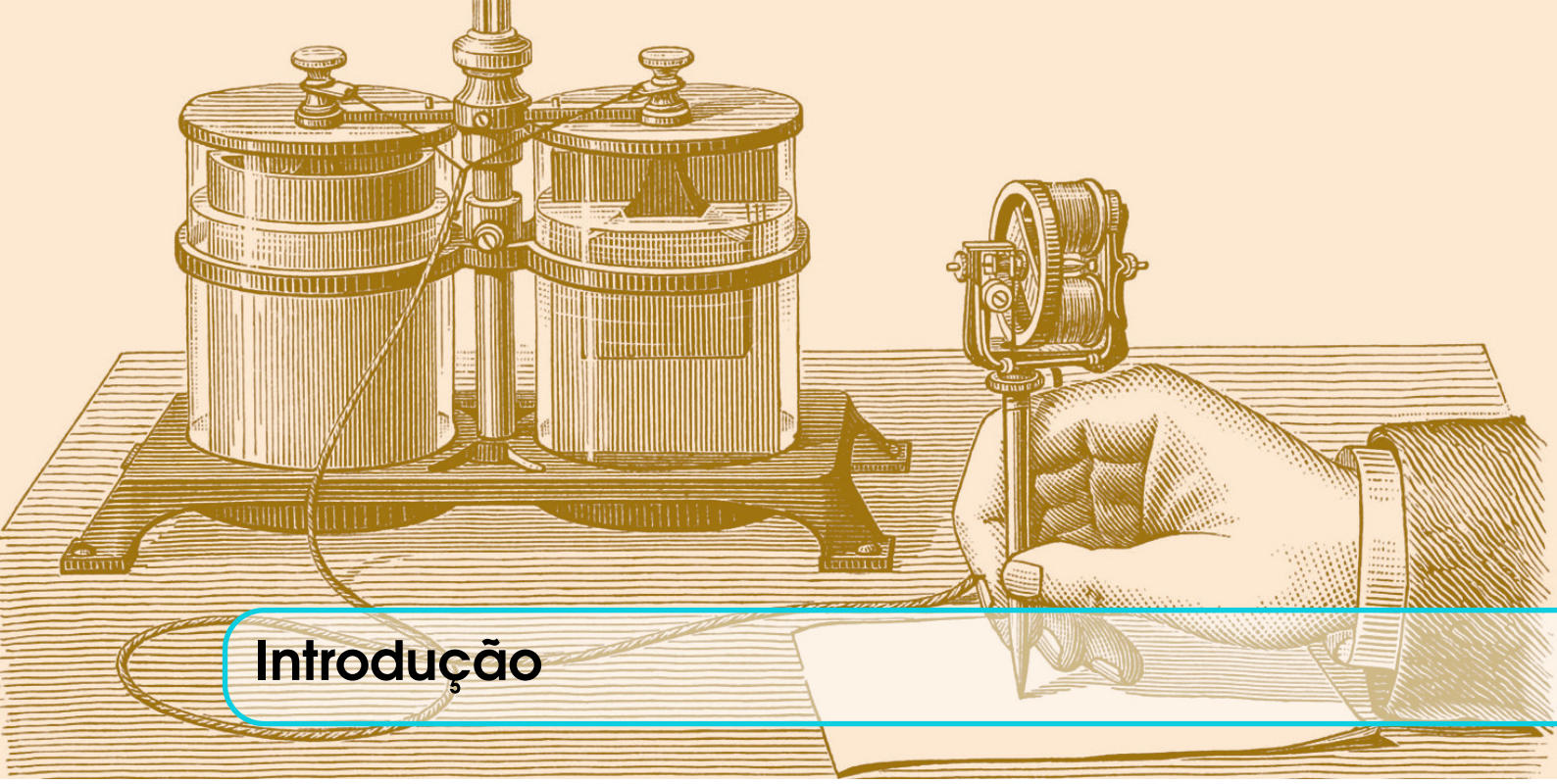
Ted Jensen é um engenheiro eletrônico aposentado que trabalhou como designer de hardware ou gerente de designers de hardware na área de gravação magnética. Programar tem sido um hobby seu, intermitentemente, desde 1968, quando ele aprendeu a perfurar cartões para enviá-los a um mainframe. (O mainframe tinha 64K de memória de núcleo magnético!).

Uso deste Material

Tudo aqui contido é liberado para o domínio público. Qualquer pessoa pode copiar ou distribuir este material da maneira que desejar. A única coisa que peço é que se este material for usado como um auxiliar de ensino em uma aula, eu agradeceria se fosse distribuído na íntegra, ou seja, incluindo todos os capítulos, o prefácio e a introdução. Eu também agradeceria se, em tais circunstâncias, o instrutor de tal classe me deixasse uma nota em um dos endereços abaixo informando-me sobre isso. Escrevi isso com a esperança de que seja útil a outras pessoas e, como não estou pedindo nenhuma remuneração financeira, a única maneira de saber que atingi, pelo menos parcialmente, esse objetivo é por meio do feedback de quem considera este material útil.

A propósito, você não precisa ser um instrutor ou professor para entrar em contato comigo. Agradeço uma nota de qualquer pessoa que considere o material útil ou que tenha uma crítica construtiva a oferecer. Também estou disposto a responder a perguntas enviadas por e-mail nos endereços abaixo.

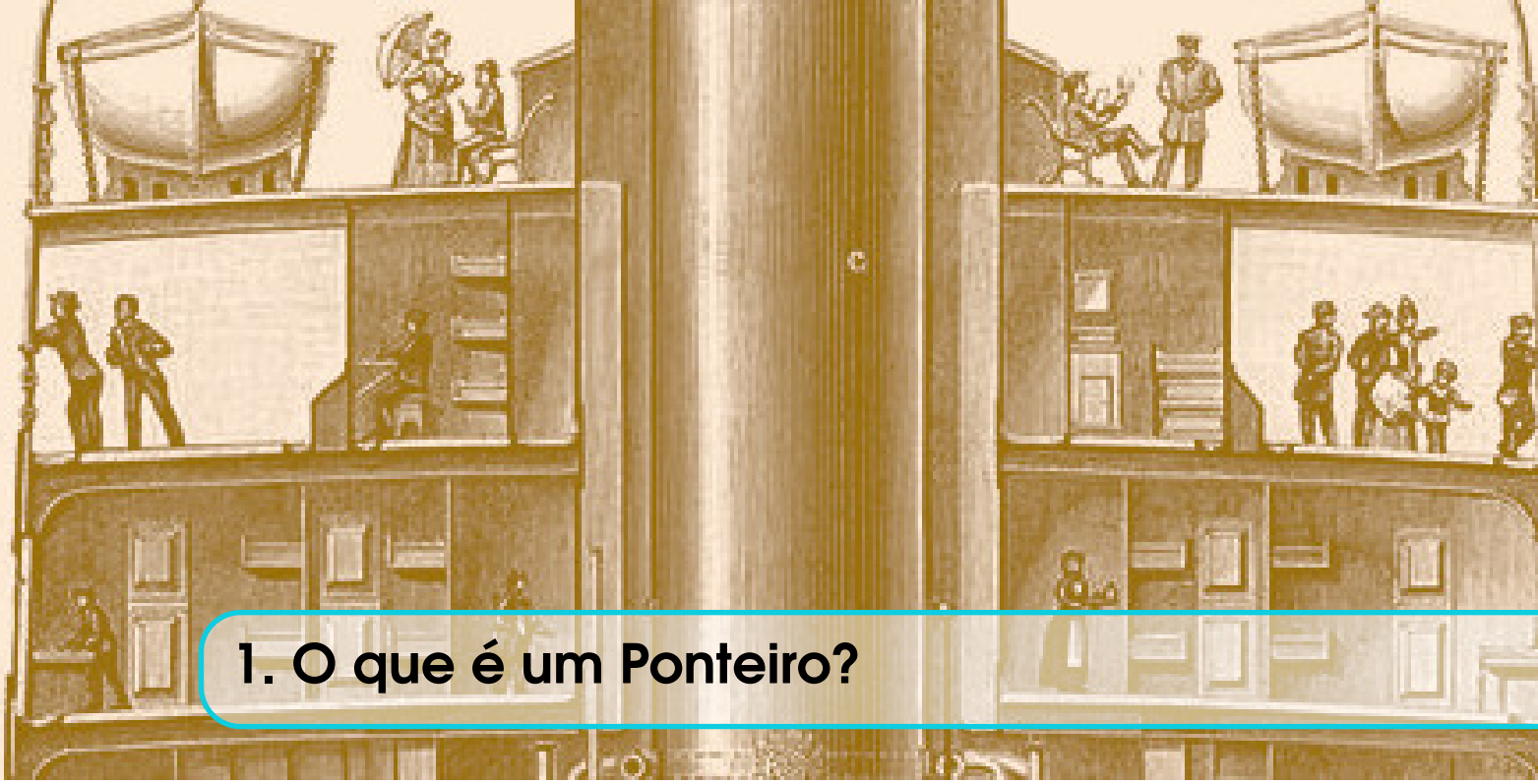
Ted Jensen
Redwood City, California
tjensen@ix.netcom.com
July 1998



Introdução

Se você deseja ser proficiente na escrita de código na linguagem de programação C, deve ter um conhecimento prático completo de como usar ponteiros. Infelizmente, os ponteiros C parecem representar um obstáculo para os recém-chegados, particularmente aqueles vindos de outras linguagens de computador, como Python, Java ou Javascript. Muitas dessas linguagens dizem não usar ponteiros, porém, por debaixo dos panos, tudo são ponteiros. A compreensão dos mecanismos dos ponteiros em C permitem um melhor entendimento dos mecanismos e limitações de linguagens como Python e Java, nas quais os ponteiros não são visíveis, mas estão lá, na infraestrutura dessas linguagens, prestando seu serviço.

Para ajudar esses recém-chegados a compreender as dicas, escrevi o seguinte material. Para obter o máximo benefício deste material, considero importante que o usuário seja capaz de executar o código nas várias listagens contidas no artigo. Tentei, portanto, manter todo o código compatível com ANSI para que funcione com qualquer compilador compatível com ANSI. Também tentei bloquear cuidadosamente o código dentro do texto. Dessa forma, com a ajuda de um editor de texto 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 ajudará na compreensão do material.



1. O que é um Ponteiro?

Uma daquelas coisas que os iniciantes em C acham difícil é o conceito de ponteiros. O objetivo deste texto é fornecer uma introdução aos ponteiros e seu uso para esses iniciantes.

Eu descobri que muitas vezes o principal motivo pelo qual os iniciantes têm problemas com ponteiros é que eles têm uma percepção fraca ou mínima das variáveis (como são usadas em C). Assim, começamos com uma discussão das variáveis C em geral.

Uma **variável** em um programa é algo com um nome, cujo valor pode variar. A maneira como o compilador e o *linker* lidam com isso é que ele atribui um bloco específico de memória dentro do computador para armazenar o valor daquela variável. O tamanho desse bloco depende do intervalo no qual a variável pode variar. Por exemplo, em PCs, o tamanho de uma variável inteira é de 4 bytes e o de um inteiro longo é de 8 bytes. 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 nome da variável e o tipo da variável. Por exemplo, declaramos uma variável do tipo inteiro com o nome *k* escrevendo:

```
int k;
```

Ao ver a parte "int" dessa instrução, o compilador reserva 4 bytes de memória (em um PC de 32 bits) para armazenar o valor do inteiro. Também configura uma tabela de símbolos. Nessa tabela, ele adiciona o símbolo *k* e o endereço relativo na memória onde esses 4 bytes foram reservados.

Então, se mais tarde escrevermos:

```
k = 2;
```

esperamos que, em tempo de execução quando esta instrução for executada, o valor 2 seja colocado naquele local de memória reservado para o armazenamento do valor de **k**. Em C, nos referimos a uma variável como o inteiro **k** como um "objeto".

Em certo sentido, existem dois 'valores" associados ao objeto **k**. Um é o valor do número inteiro armazenado lá (2 no exemplo acima) e o outro é o "valor" da localização da memória, ou seja, o endereço de **k**. Alguns textos referem-se a esses dois valores com a nomenclatura ***rvalue*** (valor à direita, pronunciado "ar value") e ***lvalue*** (valor à esquerda, pronunciado "el value"), respectivamente.

Em alguns linguagens, o *lvalue* é o valor permitido no lado esquerdo do operador de atribuição '=' (ou seja, o endereço onde termina o resultado da avaliação do lado direito). O *rvalue* é aquele que está no lado direito da instrução de atribuição, o 2 acima. *Rvalues* não podem ser usados no lado esquerdo da instrução de atribuição. Assim: **2 = k**; é ilegal.

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

> "Um **objeto** é uma região nomeada de armazenamento; um **lvalue** é uma expressão que se refere a um objeto"'

No entanto, neste ponto, a definição originalmente citada acima é suficiente. À medida que nos familiarizamos com os ponteiros, entraremos em mais detalhes sobre isso.

Ok, agora considere:

```
1  int j, k;
2
3  k = 2;
4  j = 7;
5  k = j;
```

Acima, o compilador interpreta o **j** na linha 4 como o endereço da variável **j** (seu *lvalue*) e cria o código para copiar o valor 7 para esse endereço. Na linha 5, entretanto, **j** é interpretado como seu *rvalue* (uma vez que está no lado direito do operador de atribuição '='). Ou seja, aqui **j** se refere ao valor armazenado no local da memória reservado para **j**, neste caso 7. Assim, o 7 é copiado para o endereço designado pelo *lvalor* de **k**.

Em todos esses exemplos, estamos usando inteiros de 4 bytes, portanto, todas as cópias de *rvalues* de um local de armazenamento para o outro são feitas copiando 4 bytes. Se estivéssemos usando números inteiros longos, estaríamos copiando 8 bytes.

Agora, digamos que temos um motivo para querer uma variável projetada para conter um *lvalue* (um endereço). O tamanho necessário para armazenar tal valor depende do sistema. Em computadores de mesa mais antigos com 64 K de memória total, o endereço de qualquer ponto da memória pode estar contido em 2 bytes. Os computadores com mais memória exigiriam mais bytes para armazenar um endereço. O tamanho real necessário não é muito importante, desde que tenhamos uma maneira de informar ao compilador que o que queremos armazenar é um endereço.

Essa variável é chamada de **ponteiro** (por razões que, esperamos, ficarão mais claras um pouco mais tarde). Em C, quando definimos uma variável como ponteiro, fazemos isso precedendo seu nome com um asterisco. Em C, também damos ao nosso ponteiro um tipo que, neste caso, se refere ao tipo de dados armazenados no endereço que iremos armazenar 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 '*' informa ao compilador que queremos uma variável de ponteiro, ou seja, separar quantos bytes forem necessários para armazenar um endereço na memória. O **int** diz que pretendemos usar nossa variável de ponteiro para armazenar o endereço de um inteiro. Diz-se que esse ponteiro "aponta para" um número inteiro. No entanto, observe que quando escrevemos **int k;** não atribuímos um valor a **k**. Se essa definição for feita fora de qualquer função, os compiladores compatíveis com ANSI irão inicializá-la com zero. Da mesma forma, **ptr** não tem valor, ou seja, não armazenamos um endereço nele na declaração acima. Nesse caso, novamente se a declaração estiver fora de qualquer função, ela é inicializada com um valor garantido de forma que não aponte para nenhum objeto ou função C. Um ponteiro inicializado dessa maneira é chamado de ponteiro "nulo".

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

Mas, de volta ao uso de nossa nova variável **ptr**. Suponha agora que queremos armazenar em **ptr** o endereço de nossa variável inteira **k**. Para fazer isso, usamos o operador **&** unário e escrevemos:

```
ptr = &k;
```

O que o operador **&** faz é recuperar o *lvalue* (endereço) de **k**, embora **k** esteja no lado direito do operador de atribuição '=', e o copia para o conteúdo do nosso ponteiro **ptr**. Agora, diz-se que **ptr** "aponta para" **k**. Tenha paciência conosco agora, há apenas mais um operador que precisamos discutir.

O "operador de desreferenciação" é o asterisco e é usado da seguinte forma:

```
*ptr = 7;
```

irá copiar 7 para o endereço apontado por **ptr**. Assim, se **ptr** "aponta para" (contém o endereço de) **k**, a instrução acima irá definir o valor de **k** para 7. Ou seja, quando usamos o '*' desta forma estamos nos referindo ao valor para o qual **ptr** está apontando, não o 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 ver como tudo isso se encaixa seria executar o programa a seguir e, em seguida, revisar o código e a saída com cuidado.

Programa 1.1: Variáveis e ponteiros

```
1  #include <stdio.h>
2
3  int j, k;
4  int* ptr;
5
6  int main(void) {
7      j = 1;
8      k = 2;
9      ptr = &k;
10     printf("j tem valor %d e está armazenado em %p\n", j, (void*)&j);
11     printf("k tem valor %d e está armazenado em %p\n", k, (void*)&k);
12     printf("ptr tem valor %p e está armazenado em %p\n",
13           (void*)ptr, (void*)&ptr);
14     printf("O valor do inteiro apontado por ptr é %d\n", *ptr);
15     return 0;
16 }
```

O programa 1.1 foi compilado com o compilador gnu gcc, com as seguintes opções:

```
gcc -Wall -std=c11 -pedantic -o prog1-1 prog1-1.c
```

O parâmetro `Wall` faz com que o compilador emita todos os *warnings*; `-std=c11` diz que ele deve seguir a versão **c11** da linguagem; e finalmente, `-pedantic` diz que ele deve seguir estritamente o padrão da linguagem, ignorando possíveis extensões do compilador.

Um resultado possível, executando em um computador de 64 bits, seria:

```
j tem valor 1 e está armazenado em 0x562dcb413018
k tem valor 2 e está armazenado em 0x562dcb413028
ptr tem valor 0x562dcb413028 e está armazenado em 0x562dcb413020
O valor do inteiro apontado por ptr é 2
```

N Ainda temos que discutir os aspectos de C que requerem o uso da expressão **(void *)** para fazer a conversão de ponteiros para **int** para ponteiros para **void**, necessárias aqui para a impressão com **%p**. Por enquanto, ignoramos este aspecto e nosso código é compilado com a opção `-pedantic`, que causa diversos *warnings* caso não seja feita a conversão com **(void*)**. Voltaremos a este assunto na Seção 2.1.

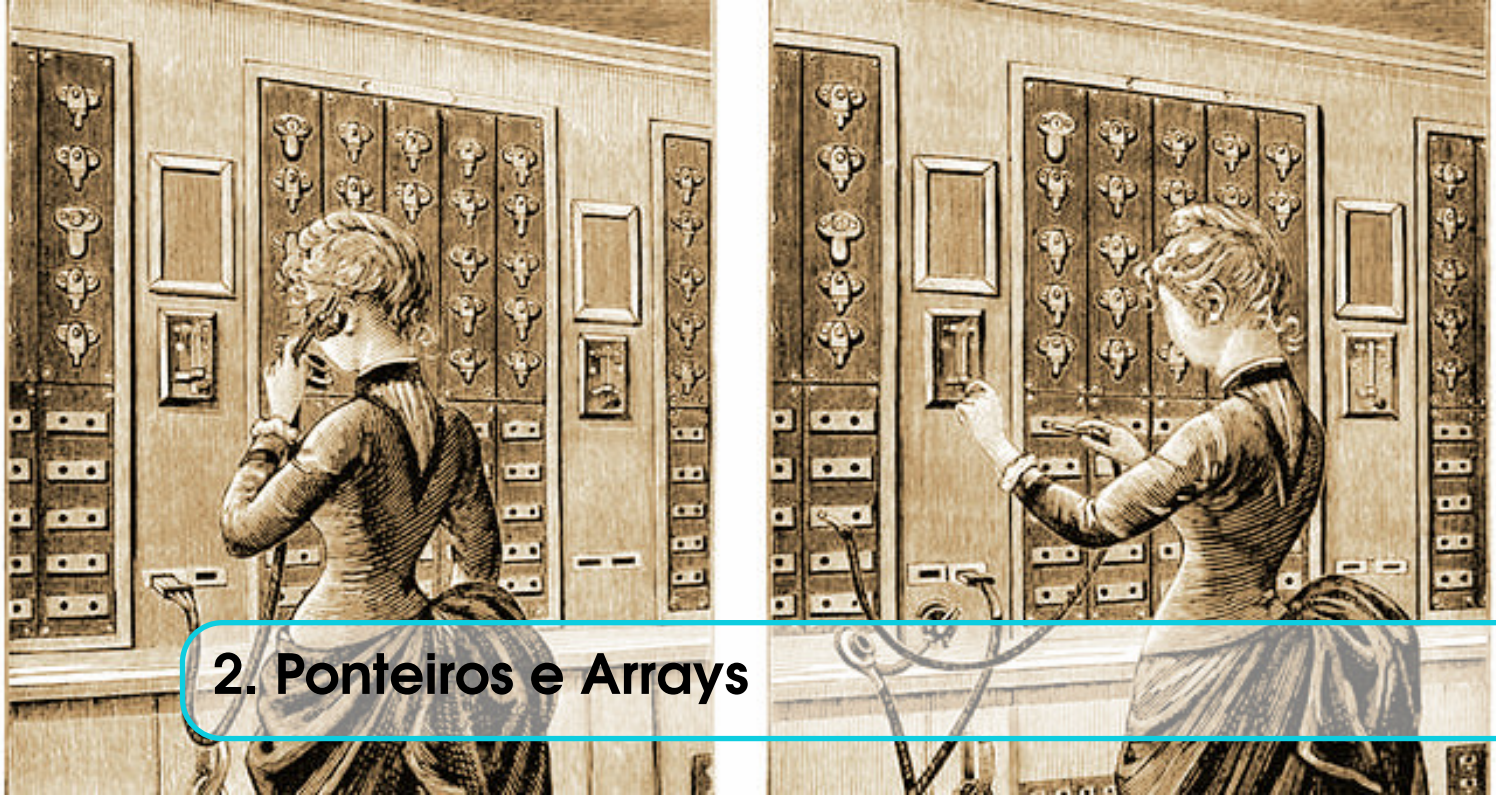
A Figura 6 representa esquematicamente a memória para este arranjo de variáveis. Lembre-se que neste caso estamos trabalhando com um computador de 64 bits, assim os endereços têm 8 bytes (dos quais escrevemos 6, por simplicidade).

Endereço	Memória	Variável
	⋮	
0x562dcb413018	1	j
0x562dcb413020	0x562dcb413028	ptr
0x562dcb413028	2	k
	⋮	

Figura 1.1: Endereços e memória.

Para revisar:

- Uma variável é declarada dando-lhe um tipo e um nome (por exemplo, **int k;**)
- Uma variável de ponteiro é declarada dando a ela um tipo e um nome (por exemplo, **int * ptr**) onde o asterisco diz ao compilador que a variável chamada **ptr** é uma variável de ponteiro e o tipo diz ao compilador para qual tipo o ponteiro deve apontar (inteiro nesse caso).
- 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 para o qual ele aponta, usando o operador unário '*' como em ***ptr**.
- Um "*lvalue*" de uma variável é o valor do seu endereço, ou seja, onde está armazenado na memória. O "*rvalue*" de uma variável é o valor armazenado nessa variável (naquele endereço).



2. Ponteiros e Arrays

Ok, vamos em frente. Vamos considerar porque precisamos identificar o tipo de variável para a qual um ponteiro aponta, como em:

```
int *ptr;
```

Uma razão para fazer isso é 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** for declarado apontando para um inteiro, 4 bytes serão copiados, se for longo, 8 bytes serão copiados. Da mesma forma, para *floats* e *doubles*, o número apropriado será copiado. Porém, definir o tipo para o qual o ponteiro aponta permite uma série de outras maneiras interessantes que um compilador pode interpretar o código. Por exemplo, considere um bloco na memória consistindo em dez inteiros em uma linha. Ou seja, 40 bytes de memória são reservados para conter 10 inteiros.

Agora, digamos que apontamos nosso ponteiro de inteiro **ptr** para o primeiro desses inteiros. Além disso, digamos que o inteiro esteja localizado na posição de memória 100 (decimal). O que acontece quando escrevemos:

```
ptr + 1;
```

Porque o compilador "sabe" que este é um ponteiro (ou seja, seu valor é um endereço) e que aponta para um inteiro (seu endereço atual, 100, é o endereço de um inteiro), ele adiciona 4 a **ptr** em vez de 1, então o ponteiro "aponta para" o **próximo inteiro**, no local da memória 104 (Figura 2.1). Da mesma forma, se o **ptr** fosse declarado como um ponteiro para um *long*, ele adicionaria 8 a ele em vez de 1. O mesmo vale para outros tipos de dados, como floats, doubles ou até mesmo tipos de dados definidos pelo usuário, como estruturas. Obviamente, esse não é o mesmo tipo de "adição" em que normalmente pensamos. Em C, é referido como adição usando "aritmética de ponteiros", um termo ao qual voltaremos mais tarde.

Da mesma forma, uma vez que **++ptr** e **ptr++** são ambos equivalentes a **ptr + 1** (embora o ponto no programa quando **ptr** é incrementado possa ser diferente), incrementar um ponteiro usando o operador unário **++**, seja pré- ou pós-, incrementa o endereço que ele armazena pela

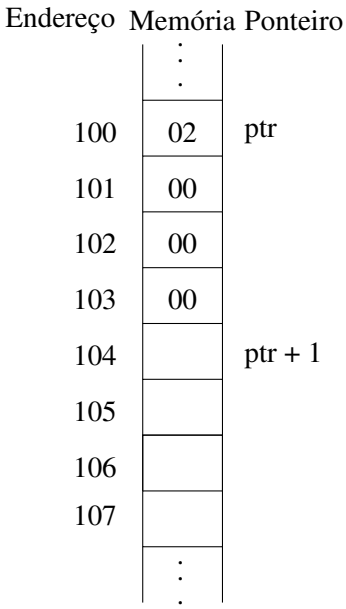


Figura 2.1: Array de inteiros, cada posição com 1 byte.

quantidade *sizeof(tipo)* onde "tipo" é o tipo do objeto apontado. (ou seja, 4 para um número inteiro, 8 para um longo, etc.).

Como um bloco de 10 inteiros localizado contiguamente na memória é, por definição, um *array* de inteiros, isso traz uma relação interessante entre *arrays* e ponteiros.

Considere o seguinte:

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

Aqui temos um array contendo 6 inteiros. Referimo-nos a cada um desses inteiros por meio de um subscrito para **meu_array**, ou seja, usando de **meu_array[0]** até **meu_array[5]**. Mas, podemos acessá-los alternativamente por meio de um ponteiro da seguinte maneira:

```
int *ptr;
ptr = &meu_array[0]; /* o ponteiro aponta para o primeiro
                        inteiro em nosso array */
```

E então poderíamos imprimir nosso *array* usando a notação de *array* ou desreferenciando nosso ponteiro. O código a seguir ilustra isso:

```
#include <stdio.h>

int meu_array[] = {1, 23, 17, 4, -5, 100};
#define N sizeof(meu_array) / sizeof(meu_array[0])
int* ptr;

int main(void) {
    ptr = &meu_array[0]; /* aponta nosso ponteiro para o primeiro
                            elemento do array */

    for (int i = 0; i < N; i++) {
        printf("meu_array[%d] = %d\t", i, meu_array[i]); /*<-- A */
        printf("ptr + %d = %d\n", i, *(ptr + i));        /*<-- B */
    }
```

```
16      return 0;
17  }
```

Programa 2.1: Ponteiros e arrays

Compile e execute o programa 2.1 e observe cuidadosamente as linhas A e B e que o programa imprime os mesmos valores em ambos os casos. Observe também como desreferenciamos nosso ponteiro na linha B, ou seja, primeiro adicionamos **i** a ele e, em seguida, desreferenciamos o novo ponteiro. Se você digitou o programa corretamente, sua saída deve ser algo como:

```
meu_array[0] = 1   ptr + 0 = 1
meu_array[1] = 23  ptr + 1 = 23
meu_array[2] = 17  ptr + 2 = 17
meu_array[3] = 4   ptr + 3 = 4
meu_array[4] = -5  ptr + 4 = -5
meu_array[5] = 100 ptr + 5 = 100
```

Mude a linha B para:

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

e execute-o novamente ... depois mude para:

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

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

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

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

podemos escrever:

```
ptr = meu_array;
```

para alcançar o mesmo resultado.

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

```
ptr = meu_array;
```

mas não podemos escrever

```
meu_array = ptr;
```

O motivo é que, embora **ptr** seja uma variável, **meu_array** é uma constante. Ou seja, o local em que o primeiro elemento de **meu_array** será armazenado não pode ser alterado depois que **meu_array[]** for declarado.

A Figura 2.2 mostra uma possível configuração dessas variáveis.

Anteriormente, ao discutir o termo “lvalue”, citei K&R-2, onde afirmava:

> “Um **objeto** é uma região nomeada de armazenamento; um **lvalue** é uma expressão que se refere a um objeto”.

Isso levanta um problema interessante. Visto que **meu_array** é uma região nomeada de armazenamento, por que **meu_array** na instrução de atribuição acima não é um *lvalue*? Para resolver esse problema, alguns se referem a **meu_array** como um “*lvalue* não modificável”.

Modifique o programa de exemplo anterior mudando

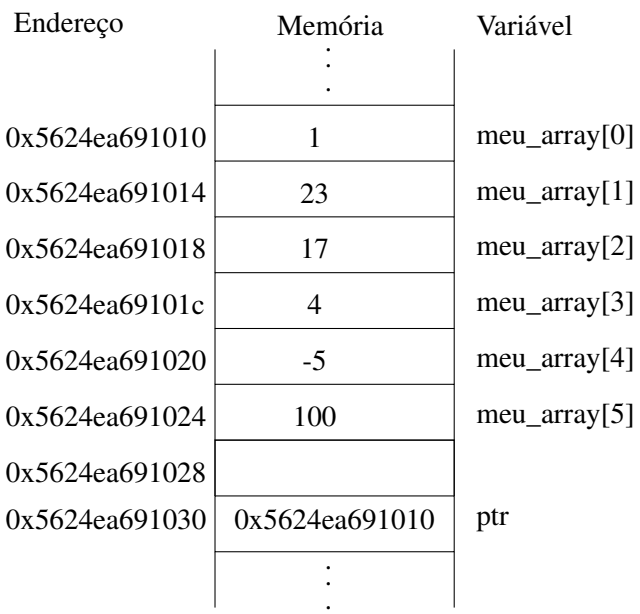


Figura 2.2: Ponteiros e arrays.

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

para

```
ptr = meu_array;
```

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

Agora, vamos nos aprofundar um pouco mais na diferença entre os nomes **ptr** e **meu_array** como usados acima. Alguns escritores se referem ao nome de um array como um ponteiro constante. O que queremos dizer com isso? Bem, para entender o termo "***constante***" neste sentido, vamos voltar à nossa definição do termo "variável". Quando declaramos uma variável, reservamos um ponto na memória para armazenar o valor do tipo apropriado. 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 da memória para o qual 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 significando o conteúdo armazenado naquele endereço de memória reservado para conter 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 então ocupa espaço na porção de dados da memória, 2 é uma constante e, como tal, em vez de reservar memória no segmento de dados, é embutido diretamente no segmento de código da memória. Ou seja, ao escrever algo como **k = i;** dizemos ao compilador para criar um código que, em tempo de execução, examinará a localização da memória **&i** para determinar o valor a ser movido para **k**, o código criado por **i = 2;** simplesmente coloca o 2 no código e não há referência ao segmento de dados. Ou seja, **k** e **i** são objetos, mas 2 não é um objeto.

Da mesma forma, no exemplo acima, uma vez que **meu_array** é uma constante, uma vez que o compilador estabelece onde o próprio *array* deve ser armazenado, ele "sabe" o endereço de **meu_array[0]** e ao ver:

```
ptr = meu_array;
```

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

2.1 Expressão (void*)

No Programa 1.1 do Capítulo 1 usamos a expressão **(void *)** sem muitas explicações. Vamos detalhar melhor seu uso. Como vimos, podemos ter ponteiros de vários tipos. Até agora, discutimos ponteiros para inteiros e ponteiros para caracteres. 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 que também é possível que o tamanho de um ponteiro possa variar dependendo do tipo de dados do objeto para o qual ele aponta. Assim, como acontece com inteiros nos quais 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 esse 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, enquanto C não permite a comparação de um ponteiro para um tipo inteiro com um ponteiro para um tipo caractere, por exemplo, qualquer um deles pode ser comparado a um ponteiro **void**. Claro, como com outras variáveis, os *casts* podem ser usados para converter de um tipo de ponteiro para outro nas circunstâncias adequadas. No Programa 1.1 do Capítulo 1, converto os ponteiros para inteiros em ponteiros vazios para torná-los compatíveis com a especificação de conversão **%p**. Em capítulos posteriores, outros *casts* serão feitos pelas razões aqui definidas.

Bem, isso é muito material técnico para digerir e não espero que um iniciante entenda tudo isso na primeira leitura. Com o tempo e experimentação, você vai querer voltar e reler os primeiros 2 capítulos. Mas, por enquanto, vamos prosseguir para a relação entre ponteiros, arrays de caracteres e strings.

3. Ponteiros e Strings

O estudo de strings é útil para vincular ainda mais a relação entre ponteiros e *arrays*. Também torna fácil ilustrar como algumas das funções de string C padrão podem ser implementadas. Finalmente, ilustra como e quando os ponteiros podem e devem ser passados para funções.

Em C, strings são *arrays* de caracteres. Isso não é necessariamente verdade em outras linguagens. Em **Java**, **Python**, **PHP** e várias outras linguagens, uma string tem seu próprio tipo de dados. Mas em C isso não acontece. Em C, uma string é um *array* de caracteres terminado com um caractere binário zero (escrito como '\0'). Para começar nossa discussão, escreveremos um código que, embora seja útil 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 construísse uma string como essa, o resultado final é uma string no sentido de que é um *array* de caracteres **terminado com um caractere nulo**. Por definição, em C, uma string é um *array* de caracteres terminado com o caractere nulo. Esteja ciente de que "nulo" **não é** o mesmo que "**NULL**". O nulo se refere a um zero conforme definido pela sequência de escape '**\0**'. Ou seja, ocupa um byte de memória. NULL, por outro lado, é o nome da macro usada para inicializar ponteiros nulos. NULL é definido (com #define) em um arquivo de cabeçalho em seu compilador C, o caractere nulo não pode ser definido por #define.

Uma vez que escrever o código acima consumiria muito tempo, C permite duas maneiras alternativas de obter a mesma coisa. Primeiro, pode-se escrever:

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

Mas isso também exige mais digitação do que o necessário. Então, C permite:

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

Quando as aspas duplas são usadas, em vez das aspas simples como foi feito nos exemplos anteriores, o caractere nulo (’**\0**’) é automaticamente anexado ao final da string.

Em todos os casos acima, acontece a mesma coisa. O compilador separa um bloco contíguo de memória de 40 bytes para conter caracteres e o inicializa de forma que os primeiros 4 caracteres sejam **Ted\0**.

Agora, considere o seguinte programa:

```
1  #include <stdio.h>
2
3  char strA[80] = "Uma string usada para demonstração";
4  char strB[80];
5
6  int main(void) {
7      char* pA;    // um ponteiro do tipo caractere
8      char* pB;    // outro ponteiro do tipo caractere
9      puts(strA);  // mostra a string A
10     pA = strA;   // aponta pA para a string A
11     puts(pA);    // mostra o que pA está apontando
12     pB = strB;   // aponta pB para a string B
13     putchar('\n');  // pula uma linha na tela
14
15     while(*pA != '\0') { // linha A (veja texto)
16         *pB++ = *pA++;   // linha B (veja texto)
17     }
18
19     *pB = '\0';  // linha C (veja texto)
20     puts(strB);  // mostra strB na tela
21     return 0;
22 }
```

Programa 3.1: Ponteiros e strings

No Programa 3.1, começamos definindo dois *arrays* de caracteres de 80 caracteres cada. Como eles são definidos globalmente, eles são inicializados com todos os ’\0’s primeiro. Então, **strA** tem os primeiros 36 caracteres inicializados para a string entre aspas (cada caractere acentuado ou cedilha ocupa o espaço de dois caracteres).

Agora, entrando no código, declaramos dois ponteiros de caracteres e mostramos a string na tela. Em seguida, “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 o que é apontado por **pA** na tela. Considere aqui que o protótipo de função para **puts()** é:

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

Por enquanto, ignore o **const**. O parâmetro passado para **puts()** é um ponteiro, isto é, 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 . Assim, 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, pois definimos pA = strA;

Dado isso, siga o código até a instrução **while()** na linha A. A linha A declara:

*Enquanto o caractere apontado por **pA** (ou seja, ***pA**) não for um caractere nulo (ou seja, a terminação ‘**\0**’), faça a linha B.*

A linha B declara:

*Copie o caractere apontado por **pA** para o espaço apontado por **pB**, então incremente **pA** de forma que aponte para o próximo caractere e **pB** para que aponte para o próximo espaço.*

Depois de copiar o último caractere, **pA** agora aponta para o caractere *nulo* de terminação e o loop termina. No entanto, não copiamos o caractere nulo. E, por definição, uma string em C deve ter terminação nula. Então, adicionamos o caractere nulo com a linha C.

É muito educativo executar este programa com seu *debugger* enquanto acompanha **strA**, **strB**, **pA** e **pB** e avança passo a passo pelo programa. É ainda mais educacional se, em vez de simplesmente definir **strB[]** como foi feito acima, inicialize-o 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 isso!

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

Claro, o que o programa acima ilustra é uma maneira simples de copiar uma string. Depois de brincar com os itens acima até que você tenha uma boa compreensão do que está acontecendo, podemos prosseguir com a criação de nosso próprio substituto para o **strcpy()** padrão que vem com C. Pode ser parecido com:

```
1  char* meu_strcpy(char* destino, char* fonte) {
2      char* p = destino;
3
4      while (*fonte != '\0') {
5          *p++ = *fonte++;
6      }
7
8      *p = '\0';
9      return destino;
10 }
```

Programa 3.2: Meu strcpy()

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

Novamente, a função é projetada para aceitar os valores de dois ponteiros de caracteres, ou seja, endereços e, portanto, no programa anterior, poderíamos escrever:

```
1  int main(void) {
2      meu_strcpy(strB, strA);
3      puts(strB);
4  }
```

Programa 3.3: Main de meu_strcpy()

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

```
char *meu_strcpy(char *destino, const char *fonte);
```

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. Então, dentro da função, você pode adicionar uma instrução que tenta alterar o conteúdo do que é apontado pela fonte, como:

```
*fonte = 'X';
```

que normalmente mudaria o primeiro caractere da string para um X. O modificador *const* deve fazer com que seu compilador pegue isso como um erro. Experimente e veja.

Agora, vamos considerar algumas das coisas que os exemplos acima nos mostraram. Em primeiro lugar, 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)++**, não incrementaríamos o ponteiro, mas aquilo para o qual o ponteiro aponta! Ou seja, se usado no primeiro caractere da string de exemplo acima, o 'T' seria incrementado para um 'U'. Você pode escrever alguns códigos simples de exemplo para ilustrar isso.

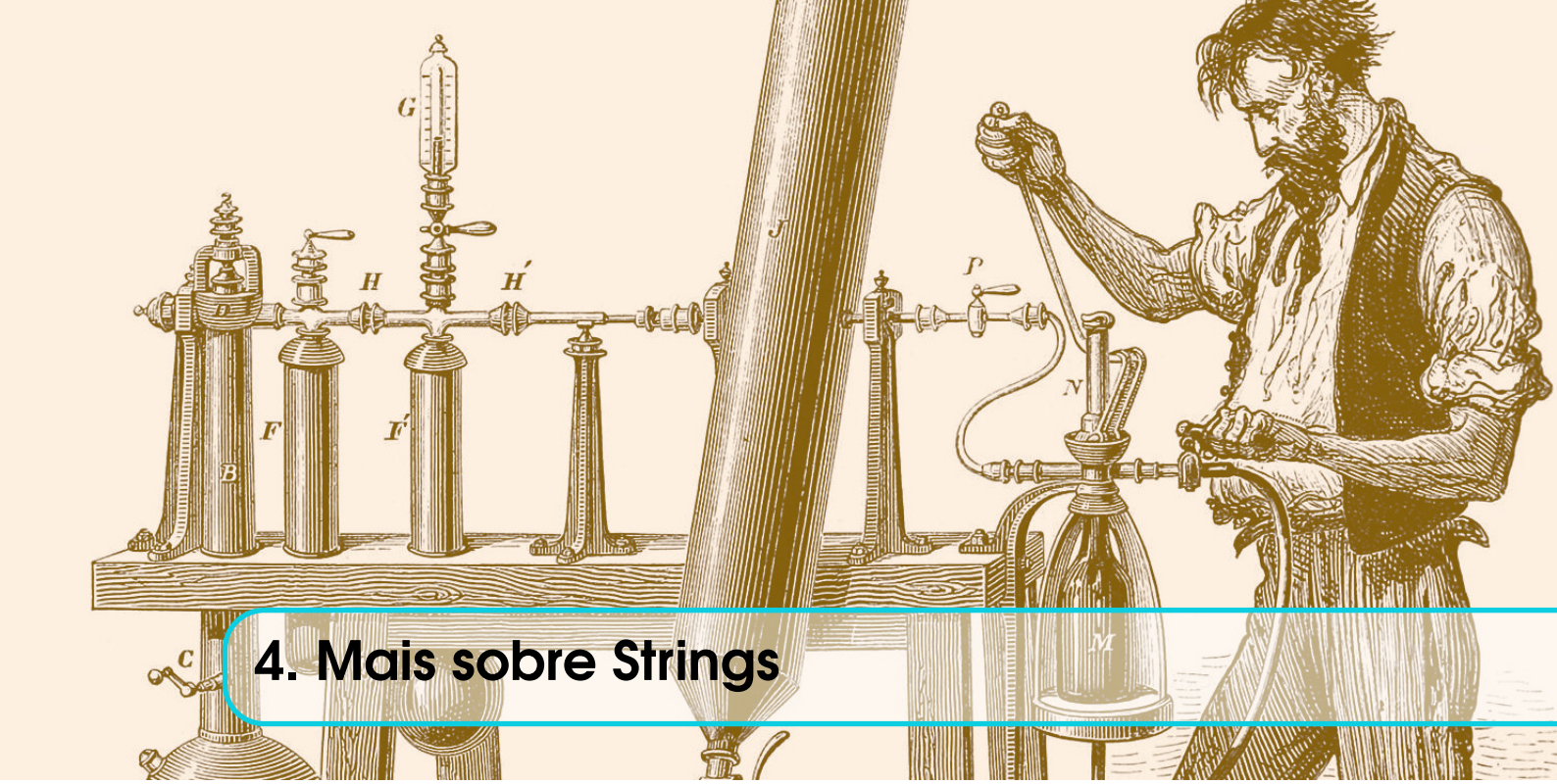
Lembre-se novamente de que uma string nada mais é do que um *array* de caracteres, com o último caractere sendo '**\0**'. O que fizemos acima foi lidar com a cópia de um *array*. Acontece que é um *array* de caracteres, mas a técnica poderia ser aplicada a um *array* de inteiros, doubles, etc. Nesses casos, no entanto, não estaríamos lidando com strings e, portanto, o final do *array* não seria marcado com um valor especial como o caractere nulo. Poderíamos implementar uma versão que contasse com um valor especial para identificar o fim. Por exemplo, podemos copiar um *array* de inteiros positivos marcando o final com um inteiro negativo. Por outro lado, é mais comum que, quando escrevemos uma função para copiar um *array* de itens que não sejam strings, passemos à função o número de itens a serem copiados, bem como o endereço do *array*, por exemplo, algo como o seguinte protótipo pode indicar:

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

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

Isso permite o uso de funções para manipular grandes *arrays*. Por exemplo, se temos um *array* de 5000 inteiros que queremos manipulá-lo com uma função, precisamos apenas passar para essa função o endereço do *array* (e qualquer informação auxiliar como **n** acima, dependendo do que estamos fazendo). O *array* em si não é passado, ou seja, o *array* de inteiros não é copiado e colocado na pilha antes de chamar a função, apenas seu endereço é enviado.

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



4. Mais sobre Strings

Bem, nós progredimos bastante em pouco tempo! Vamos recuar um pouco e ver o que foi feito no Capítulo 3 sobre cópia de strings, mas sob uma luz diferente. Considere a seguinte função:

```
1  char* meu_strcpy(char dest[], char fonte[]) {
2      int i = 0;
3
4      while (fonte[i] != '\0') {
5          dest[i] = fonte[i];
6          i++;
7      }
8
9      dest[i] = '\0';
10     return dest;
11 }
```

Programa 4.1: meu_strcpy() com arrays

Lembre-se de que as strings são *arrays* de caracteres. Aqui, escolhemos usar a notação de *array* em vez da notação de ponteiro para fazer a cópia real. Os resultados são os mesmos, ou seja, a string é copiada usando essa notação com a mesma precisão de antes. Isso levanta alguns pontos interessantes que discutiremos.

Como os parâmetros são passados por valor, tanto na passagem de um ponteiro de caractere quanto no nome do *array* como acima, o que realmente é passado é o endereço do primeiro elemento de cada *array*. Assim, o valor numérico do parâmetro passado é o mesmo se usarmos um ponteiro de caractere ou um nome de *array* como parâmetro. Isso tenderia a implicar que de alguma forma **fonte[i]** é o mesmo que ***(p + i)**.

De fato, isso é verdade, ou seja, sempre que alguém escreve um **a[i]**, ele pode ser substituído por ***(a + i)** sem problemas. Na verdade, o compilador criará o mesmo código em ambos os casos. Portanto, vemos que a aritmética de ponteiro é a mesma coisa que a indexação de *array*. Qualquer sintaxe produz o mesmo resultado.

Isso NÃO quer dizer que ponteiros e *arrays* sejam a mesma coisa, eles não são. Estamos apenas dizendo que, para identificar um determinado elemento de um *array*, temos a opção de duas

sintaxes, uma usando indexação de *array* e a outra usando aritmética de ponteiros, que produzem resultados idênticos.

4.1 O curioso caso de 3(a)

Agora, olhando para esta 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 vem a curiosa verdade que se:

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

escrever

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

é o mesmo que escrever

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

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

4.2 Quem é mais rápido?

Agora, olhando para a nossa função, quando escrevemos:

```
dest[i] = fonte[i];
```

devido ao fato de que a indexação de *array* e a aritmética de ponteiro produzem resultados idênticos, podemos escrever isso como:

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

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

Outra forma de acelerar a versão do ponteiro seria alterar:

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

para simplesmente

```
while (*fonte)
```

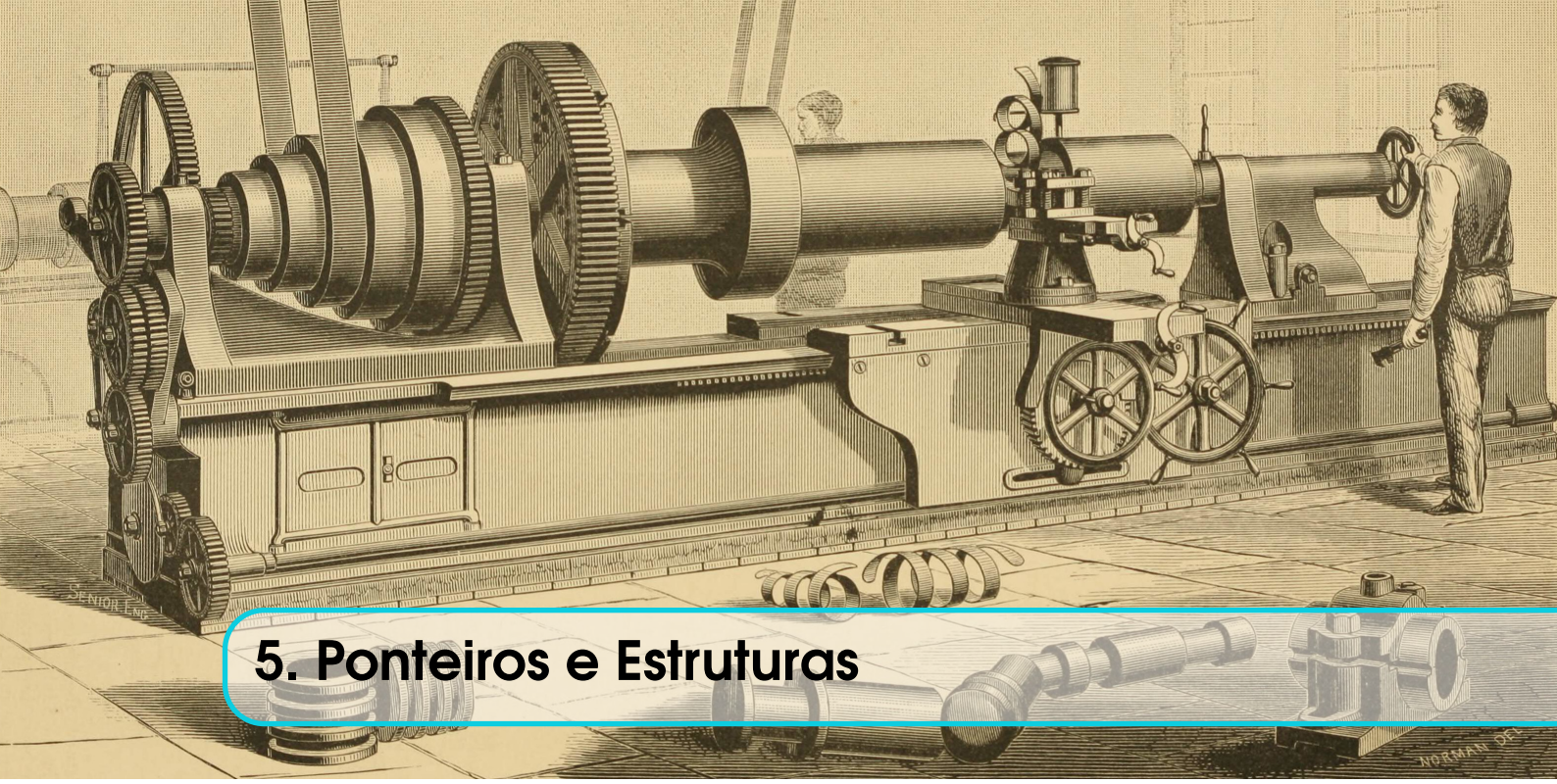
já que o valor entre parênteses irá para zero (FALSO) ao mesmo tempo em ambos os casos.

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

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

e quaisquer outras que você possa ter em seu sistema.

Voltaremos às strings e sua manipulação por meio de ponteiros em um capítulo futuro. Por enquanto, vamos prosseguir e discutir um pouco as estruturas.



5. Ponteiros e Estruturas

Como você deve saber, podemos declarar 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:

```
struct tag {
  char sobrenome[20];
  char nome[20];
  int idade;
  float salario;
};
```

Digamos que temos várias dessas estruturas em um arquivo em disco, e queremos ler e imprimir o nome e o sobrenome de cada registro para que possamos ter uma lista das pessoas em nossos arquivos. As informações restantes não serão impressas. Queremos fazer essa 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 a escrita da função, não a leitura do arquivo que, presumivelmente, sabemos fazer.

Para revisão, lembre-se de que podemos acessar membros da estrutura com o operador ponto como em:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct tag {
5      char sobrenome[20];
6      char nome[20];
7      int idade;
8      float salario; // por ex.  12.5 por hora
9  };
10
11 struct tag minha_struct;
12
13 int main(void) {
```

```
14      strcpy(minha_struct.sobrenome, "Jensen");
15      strcpy(minha_struct.nome, "Ted");
16      printf("\n%s ", minha_struct.nome);
17      printf("%s\n", minha_struct.sobrenome);
18      return 0;
19  }
```

Programa 5.1: Estruturas

Agora, esta estrutura em particular é bem pequena comparada àquelas normalmente usadas em programas C. À estrutura acima, podemos adicionar:

```
data_de_admissao;
data_do_ultimo_aumento;
ultimo_percentual_de_aumento;
telefone_de_emergencia;
plano_de_saude;
numero_de_previdencia;
etc.....
```

Se temos um grande número de funcionários, o que queremos fazer é manipular os dados dessas estruturas por meio de funções. Por exemplo, podemos querer que uma função imprima o nome do funcionário listado em qualquer estrutura passada a ela. Porém, no C original (*Kernighan & Ritchie*, 1ª Edição) não era possível passar uma estrutura, apenas um ponteiro para uma estrutura poderia ser passado. Desde **ANSI C**, é permitido passar a estrutura completa. Mas, como nosso objetivo aqui é aprender mais sobre ponteiros, não vamos nos estender nisso.

De qualquer forma, se passarmos a estrutura inteira, isso significa que devemos copiar o conteúdo da estrutura da função que chama para a função que é chamada. Em sistemas que usam pilhas, isso é feito fazendo um *push* do conteúdo da estrutura para a pilha. Com grandes estruturas, isso pode ser um problema. No entanto, passar um ponteiro usa uma quantidade mínima de espaço de pilha.

Em qualquer caso, como esta é uma discussão sobre ponteiros, discutiremos como passamos um ponteiro para uma estrutura e então o usamos dentro da função.

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

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

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

Olhe com atenção. Ele diz: substitua o que está entre parênteses pelo que **st_ptr** aponta, que é a estrutura **minha_struct**. Portanto, é o mesmo que **minha_struct.idade**.

No entanto, esta é uma expressão usada com bastante frequência e os designers de C criaram uma sintaxe alternativa com o mesmo significado, que é:

```
st_ptr->idade = 63;
```

Com isso em mente, observe o seguinte programa:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct tag {
5      char sobrenome[20];
6      char nome[20];
7      int idade;
```

```
8       float salario;
9   };

11  struct tag minha_struct;
12  void mostra_nome(struct tag* p);

14  int main(void) {
15      struct tag* st_ptr; /* um ponteiro para uma estrutura */
16      st_ptr = &minha_struct; /* aponta para minha_struct */
17      strcpy(minha_struct.sobrenome, "Jensen");
18      strcpy(minha_struct.nome, "Ted");
19      printf("\n%s ", minha_struct.nome);
20      printf("%s\n", minha_struct.sobrenome);
21      minha_struct.idade = 63;
22      mostra_nome(st_ptr); // passa o ponteiro
23      return 0;
24  }

26  void mostra_nome(struct tag* p) {
27      printf("\n%s ", p->nome);
28      printf("%s ", p->sobrenome);
29      printf("%d\n", p->idade);
30  }
```

Programa 5.2: Ponteiros e Estruturas

Novamente, essa é uma grande quantidade de informações para absorver de uma vez. O leitor deve compilar e executar os vários trechos de código e usar um depurador para monitorar coisas como **minha_struct** e **p** enquanto percorre o **main** e segue o código para dentro da função para ver o que está acontecendo.

6. Mais sobre Strings e Arrays de Strings

Bem, vamos voltar um pouco às strings. A seguir, todas as atribuições devem ser entendidas como globais, ou seja, feitas fora de qualquer função, incluindo *main()*.

Indicamos em um capítulo anterior que poderíamos escrever:

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

que alocaria espaço para um *array* de 40 bytes e colocaria a string nos primeiros 4 bytes (três para os caracteres entre aspas e um quarto para lidar com a terminação '**\0**').

Na verdade, se tudo o que quiséssemos fazer fosse armazenar o nome "Ted", poderíamos escrever:

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

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

Em algum código, em vez do descrito acima, você verá:

```
char *meu_nome = "Ted";
```

que é uma abordagem alternativa. Existe alguma diferença entre eles? A resposta é... sim. Usando a notação de *array*, 4 bytes de armazenamento no bloco de memória estática são ocupados, um para cada caractere e um para o caractere nulo de terminação. Mas, na notação de ponteiro, os mesmos 4 bytes necessários, mais **N** bytes para armazenar a variável de ponteiro **meu_nome** (onde **N** depende do sistema, mas geralmente tem um mínimo de 2 bytes e pode ser 4 ou mais). Veja a Figura 6.1.

Na notação de *array*, **meu_nome** é a abreviação de **&meu_nome[0]**, que é o endereço do primeiro elemento do *array*. Como a localização do *array* é fixada durante o tempo de execução, isso é uma constante (não uma variável). Na notação de ponteiro, **meu_nome** é uma variável. Quanto a qual é o **melhor** método, isso depende do que você vai fazer no resto do programa.

Vamos agora dar um passo adiante e considerar o que acontece se cada uma dessas declarações for feita dentro de uma função, em oposição a globalmente fora dos limites de qualquer função.

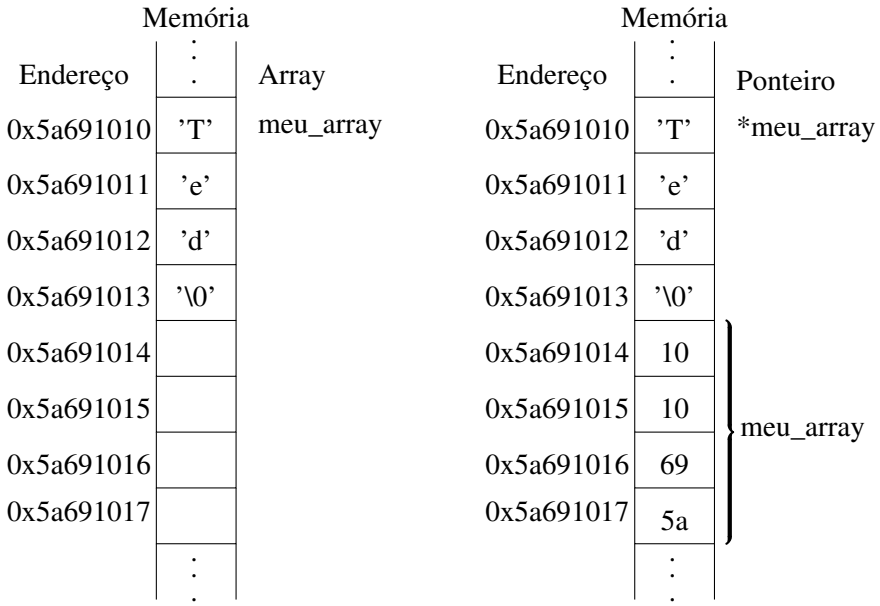


Figura 6.1: Uso da memória com Array e Ponteiro (cada posição 1 byte).

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

void minha_funcao_B(char *ptr)
  {
  char *cp = "FGHIJ";
  .
  .
  }
```

No caso de **minha_funcao_A**, o conteúdo, ou valor(es) do *array* **a[]** é considerado como sendo os dados. Diz-se que o *array* foi inicializado com os valores **ABCDE**. No caso de **minha_funcao_B**, o valor do ponteiro **cp** é considerado como sendo o dado. O ponteiro foi inicializado para apontar para a string **FGHIJ**. Tanto em **minha_funcao_A** quanto em **minha_funcao_B** as definições são variáveis locais e, portanto, a string **ABCDE** é armazenada na pilha, assim como o valor do ponteiro **cp**. A string **FGHIJ** pode ser armazenada em qualquer lugar. No meu sistema, ele fica armazenado no segmento de dados, também conhecido como *heap*. Nas Figuras 6.2 e 6.3 podemos ver como esses elementos são alocados na pilha (stack) ou na memória de dados (heap) conforme a forma de sua declaração.

A propósito, a inicialização do *array* de variáveis automáticas como fiz em **minha_funcao_A** era ilegal no livro *K&R* C antigo e apenas "atingiu a maioridade" a partir do ANSI C. Um fato que pode ser importante quando se considera portabilidade e compatibilidade com versões anteriores.

Já que estamos discutindo o relacionamento/diferenças entre ponteiros e *arrays*, vamos prosseguir para os *arrays* multidimensionais. Considere, por exemplo, o *array*:

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

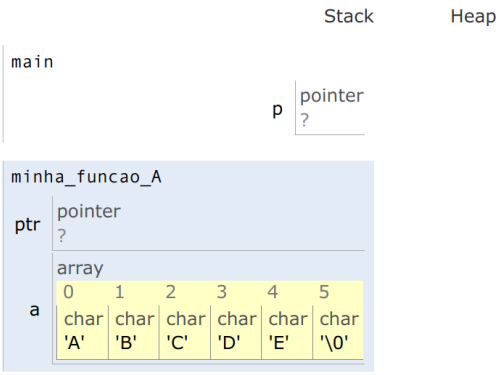


Figura 6.2: Declaração de string com Array.

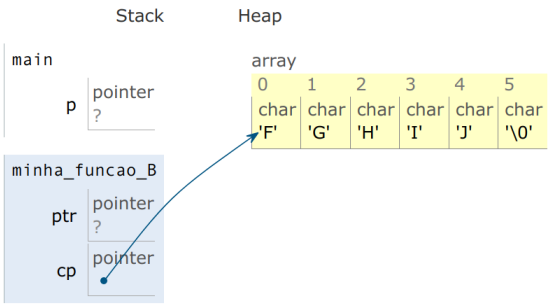


Figura 6.3: Declaração de string com Ponteiro.

O que isso significa? Bem, vamos considerar uma declaração com a seguinte configuração: char multi[5][10];

Vamos considerar a parte sublinhada como o "nome" de um *array*. Depois, acrescentando **char** e o **[10]**, temos um *array* de 10 caracteres. Mas, o próprio nome **multi[5]** é um *array* que indica que existem 5 elementos, cada um sendo um *array* de 10 caracteres. Portanto, temos um *array* de 5 *arrays* de 10 caracteres cada.

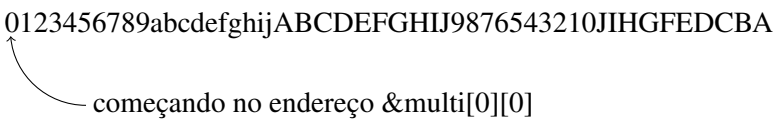
Suponha que preenchemos este *array* bidimensional com algum tipo de dado. Na memória, pode parecer que foi formado pela inicialização de 5 *arrays* separados usando algo como:

```
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, elementos individuais podem ser endereçados usando uma sintaxe como:

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

Como os *arrays* são contíguos na memória, nosso bloco de memória real para o caso acima deve ser semelhante a:



Observe que eu não escrevi **multi[0] = "0123456789"**. Se eu tivesse feito isso, uma terminação '**\0**' estaria implícita, pois sempre que as aspas duplas são usadas, um caractere '**\0**' é anexado aos caracteres contidos nessas aspas. Se fosse esse o caso, eu teria que reservar um espaço para 11 caracteres por linha em vez de 10.

Meu objetivo aqui é ilustrar como a memória é configurada para *arrays* bidimensionais. Ou seja, este é um *array* bidimensional de caracteres, NÃO um *array* de "strings".

Agora, o compilador sabe quantas colunas estão presentes no *array* para que possa interpretar **multi + 1** como o endereço do 'a' na 2ª linha. Ou seja, ele adiciona 10, o número de colunas, para obter essa localização. Se estivéssemos lidando com inteiros e um *array* com a mesma dimensão, o compilador adicionaria **10 * sizeof(int)** que, na minha máquina, seria 40. Assim, o endereço do 9 na 4ª linha acima seria &multi[3][0] ou *(multi + 3) em notação de ponteiros. Para obter o conteúdo do 2º elemento na 4ª 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] levam aos mesmo resultado.
```

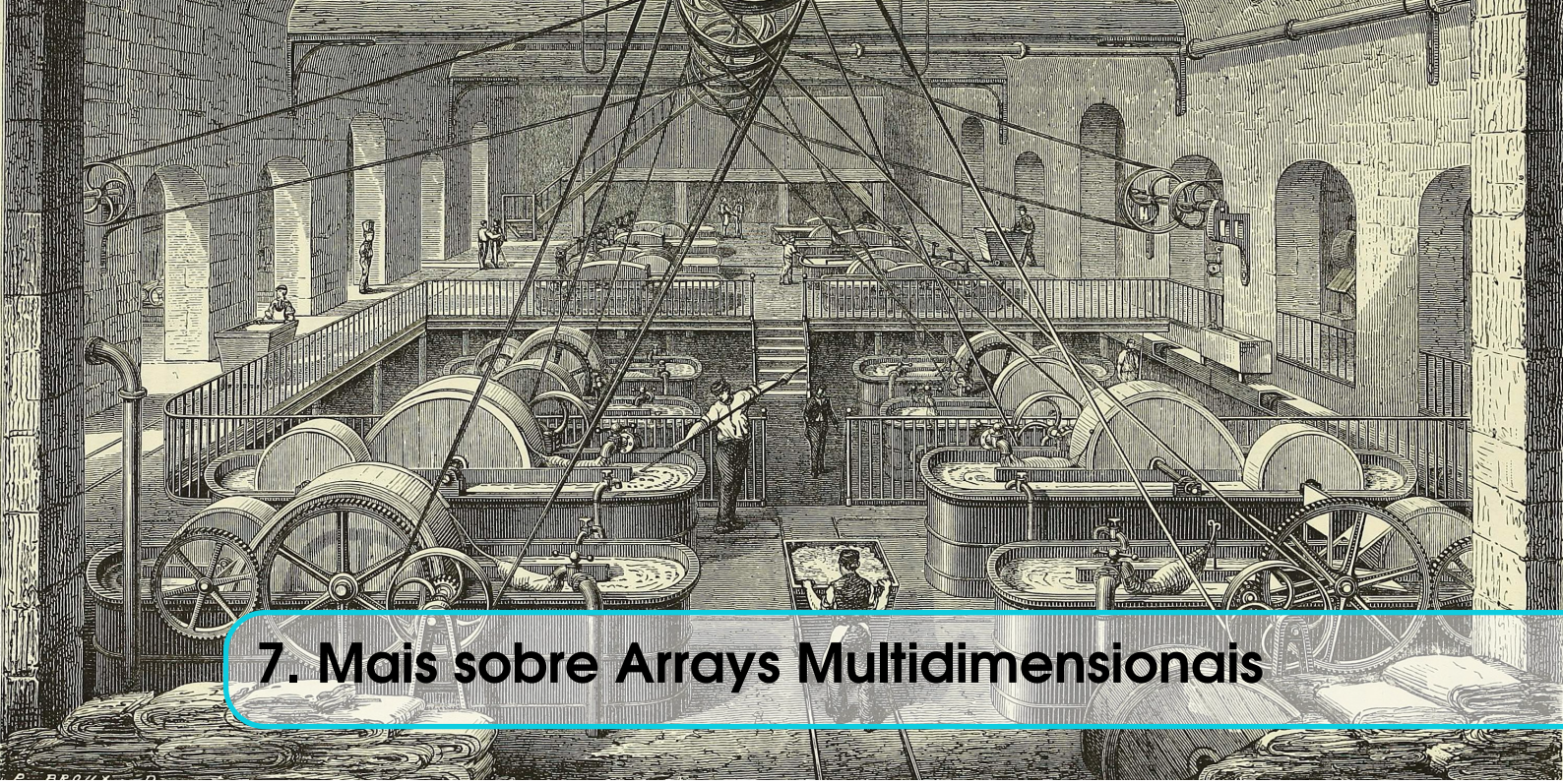
O programa a seguir ilustra isso usando *arrays* de inteiros em vez de *arrays* de caracteres.

```
1  #include <stdio.h>
2  #define LINHAS 5
3  #define COLUNAS 10
4
5  int multi[LINHAS][COLUNAS];
6
7  int main(void) {
8
9      for (int lin = 0; lin < LINHAS; lin++) {
10         for (int col = 0; col < COLUNAS; col++) {
11             multi[lin][col] = lin * col;
12         }
13     }
14
15     for (int lin = 0; lin < LINHAS; lin++) {
16         for (int col = 0; col < COLUNAS; col++) {
17             printf("\n%d ", multi[lin][col]);
18             printf("%d ", *(*(multi + lin) + col));
19         }
20     }
21
22     return 0;
23 }
```

Programa 6.1: Array bidimensional e ponteiros

Por causa da dupla desreferência necessária na versão do ponteiro, o nome de um *array* bidimensional é frequentemente considerado equivalente a um ponteiro para um ponteiro. Com um *array* tridimensional estaríamos lidando com um *array* de *arrays* de *arrays* e alguns podem 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 o *array*, definindo-o usando a notação de *array*. Portanto, estamos lidando com uma constante, não uma variável. Ou seja, estamos falando de um endereço fixo, não de um ponteiro variável. A função de desreferenciação usada acima nos permite acessar qualquer elemento no *array* de *arrays* sem a necessidade de alterar o valor desse endereço (o endereço de **multi[0][0]** conforme fornecido pelo símbolo **multi**).



7. Mais sobre Arrays Multidimensionais

No capítulo anterior, notamos que dado

```
#define LINHAS 5
#define COLUNAS 10

int multi[LINHAS][COLUNAS];
```

podemos acessar elementos individuais do *array* **multi** usando:

```
multi[lin][col]
```

ou

```
*(*(multi + lin) + col)
```

Para entender mais completamente o que está acontecendo, vamos substituir

```
*(multi + lin)
```

por **X**, como em:

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

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

Uma vez que conhecemos o layout da memória para *arrays* bidimensionais, podemos determinar que na expressão **multi + lin** como usada acima, **multi + lin + 1** deve aumentar por valor um valor igual ao necessário para "apontar para" a próxima linha, que neste caso, seria um valor igual a **COLUNAS * sizeof(int)**.

Isso significa que se a expressão ***(*(multi + lin) + col)** deve ser avaliada corretamente em tempo de execução, o compilador deve gerar o código que leva em consideração o valor de **COLUNAS**, ou seja, a 2ª dimensão. Por causa da equivalência das duas formas de expressão,

isso é verdade quer estejamos usando a expressão de ponteiro como aqui ou a expressão de *array* **multi[linha][col]**.

Assim, para avaliar qualquer uma das expressões, um total de 5 valores deve ser conhecido:

1. O endereço do primeiro elemento do array, que é retornado pela expressão **multi**, ou seja, o nome do **array**.
2. O tamanho do tipo dos elementos do *array*, neste caso **sizeof(int)**.
3. A 2ª dimensão do *array*.
4. O valor de índice específico para a primeira dimensão, **lin** neste caso.
5. O valor de índice específico para a segunda dimensão, **col** neste caso.

Dado tudo isso, considere o problema de projetar uma função para manipular os valores dos elementos de um *array* previamente declarado. Por exemplo, um que definiria todos os elementos do *array* **multi** para o valor 1.

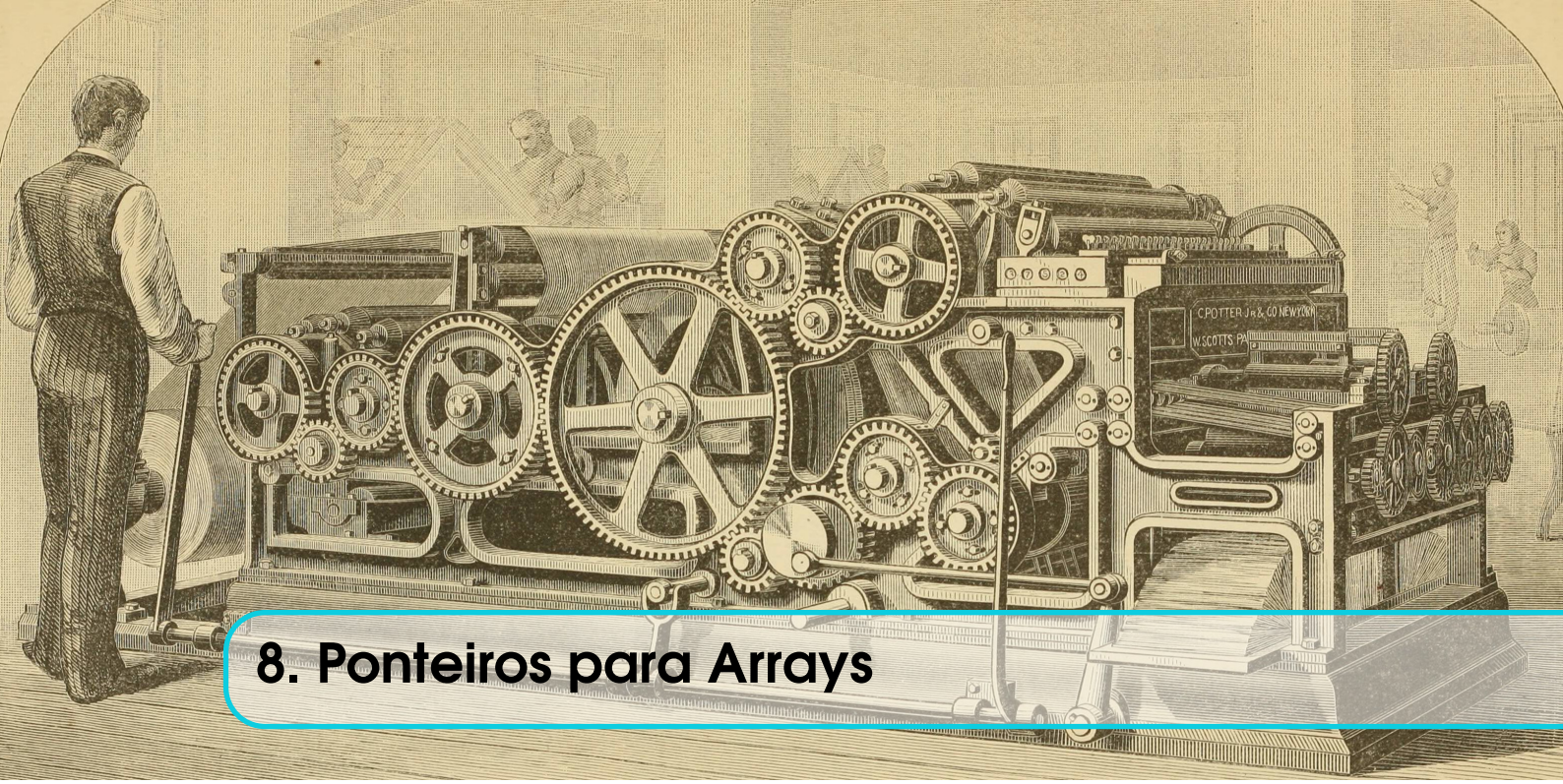
```
void define_valor(int m_array[][COLUNAS]) {
  for (int lin = 0; lin < LINHAS; lin++){
    for (int col = 0; col < COLUNAS; col++){
      m_array[lin][col] = 1;
    }
  }
}
```

E para chamar essa função, usaríamos:

```
define_valor(multi);
```

Agora, dentro da função, usamos os valores *#defined* por LINHAS e COLUNAS que definem os limites dos *loops* **for**. Mas, esses *#defines* são apenas constantes no que diz respeito ao compilador, ou seja, não há nada que os conecte ao tamanho do *array* dentro da função. **lin** e **col** são variáveis locais, é claro. A definição formal do parâmetro permite ao compilador determinar as características associadas ao valor do ponteiro que será passado em tempo de execução. Realmente não precisamos da primeira dimensão e, como será visto mais tarde, há ocasiões em que preferiríamos não defini-la dentro da 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[lin][col]** como foi descrito. Enquanto o parâmetro define o tipo de dados (**int** neste caso) e as variáveis automáticas para linha e coluna são definidas nos *loops for*, apenas um valor pode ser passado usando um único parâmetro. Nesse caso, esse é o valor de **multi** conforme observado na instrução de chamada, ou seja, o endereço do primeiro elemento, muitas vezes referido como um ponteiro para o *array*. Assim, a única forma que temos de informar o compilador da 2ª dimensão é incluindo explicitamente na definição dos parâmetros.

Na verdade, em geral, todas as dimensões de ordem superior à primeira são necessárias ao lidar com *arrays* multidimensionais. Ou seja, se estamos falando de *arrays* tridimensionais, a 2ª e a 3ª dimensões devem ser especificadas na definição do parâmetro.



8. Ponteiros para Arrays

Os ponteiros, é claro, podem ser "apontados para" qualquer tipo de objeto de dados, incluindo *arrays*. Embora isso tenha ficado evidente quando discutimos o Programa 3.1, é importante expandir como fazemos isso quando se trata de *arrays* multidimensionais.

Para revisar, no Capítulo 2, afirmamos que, dado um *array* de inteiros, poderíamos apontar um ponteiro de inteiro para esse *array* usando:

```
int *ptr;
ptr = &meu_array[0]; // Aponta nosso ponteiro para o
                     // primeiro inteiro de nosso array
```

Como afirmamos lá, o tipo da variável de ponteiro deve corresponder ao tipo do primeiro elemento do *array*.

Além disso, podemos usar um ponteiro como um parâmetro formal de uma função projetada para manipular um *array*. por exemplo.

Dado:

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

Alguns programadores podem preferir escrever o protótipo da função como:

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

que tenderia a informar outras pessoas que poderiam usar esta função que a função foi projetada para manipular os elementos de um *array*. Obviamente, em ambos os casos, o que realmente é passado é o valor de um ponteiro para o primeiro elemento do *array*, independente de qual notação é usada no protótipo ou definição da função. Observe que, se a notação de *array* for usada, não há necessidade de passar a dimensão real do *array*, pois não estamos passando o *array* inteiro, apenas o endereço do primeiro elemento.

Agora nos voltamos para o problema do *array* bidimensional. Conforme declarado no último capítulo, C interpreta um *array* bidimensional como um *array* de *arrays* unidimensionais. Sendo esse o caso, o primeiro elemento de um *array* bidimensional de inteiros é um *array* unidimensional de inteiros. E um ponteiro para um *array* bidimensional de inteiros deve ser um ponteiro para

esse tipo de dados. Uma forma de o conseguir é através da utilização da palavra-chave "**typedef**". **typedef** atribui um novo nome a um tipo de dados especificado. Por exemplo:

```
typedef unsigned char byte;
```

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

```
byte b[10];
```

seria um *array* de caracteres sem sinal.

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

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

Array se torna um tipo de dados para um *array* de 10 inteiros. ou seja, **Array meu_arr;** declara **meu_arr** como um *array* de 10 inteiros e **Array arr2d[5];** torna **arr2d** um *array* de 5 *arrays* de 10 inteiros cada.

Observe também que **Array *p1d;** torna **p1d** um ponteiro para um *array* de 10 inteiros. Como ***p1d** aponta para o mesmo tipo que **arr2d**, atribuir o endereço do *array* bidimensional **arr2d** a **p1d**, o ponteiro para um *array* unidimensional de 10 inteiros é aceitável. ou seja,

```
p1d = &arr2d[0];
```

ou

```
p1d = arr2d;
```

estão ambos corretos.

Como o tipo de dados que usamos para nosso ponteiro é um *array* de 10 inteiros, esperaríamos que incrementar **p1d** em 1 mudaria seu valor em **10 * sizeof(int)**, o que realmente acontece. Ou seja, **sizeof(*p1d)** é 40 (em um computador com inteiros de 4 bytes). Você pode provar isso para si mesmo escrevendo e executando um programa curto simples.

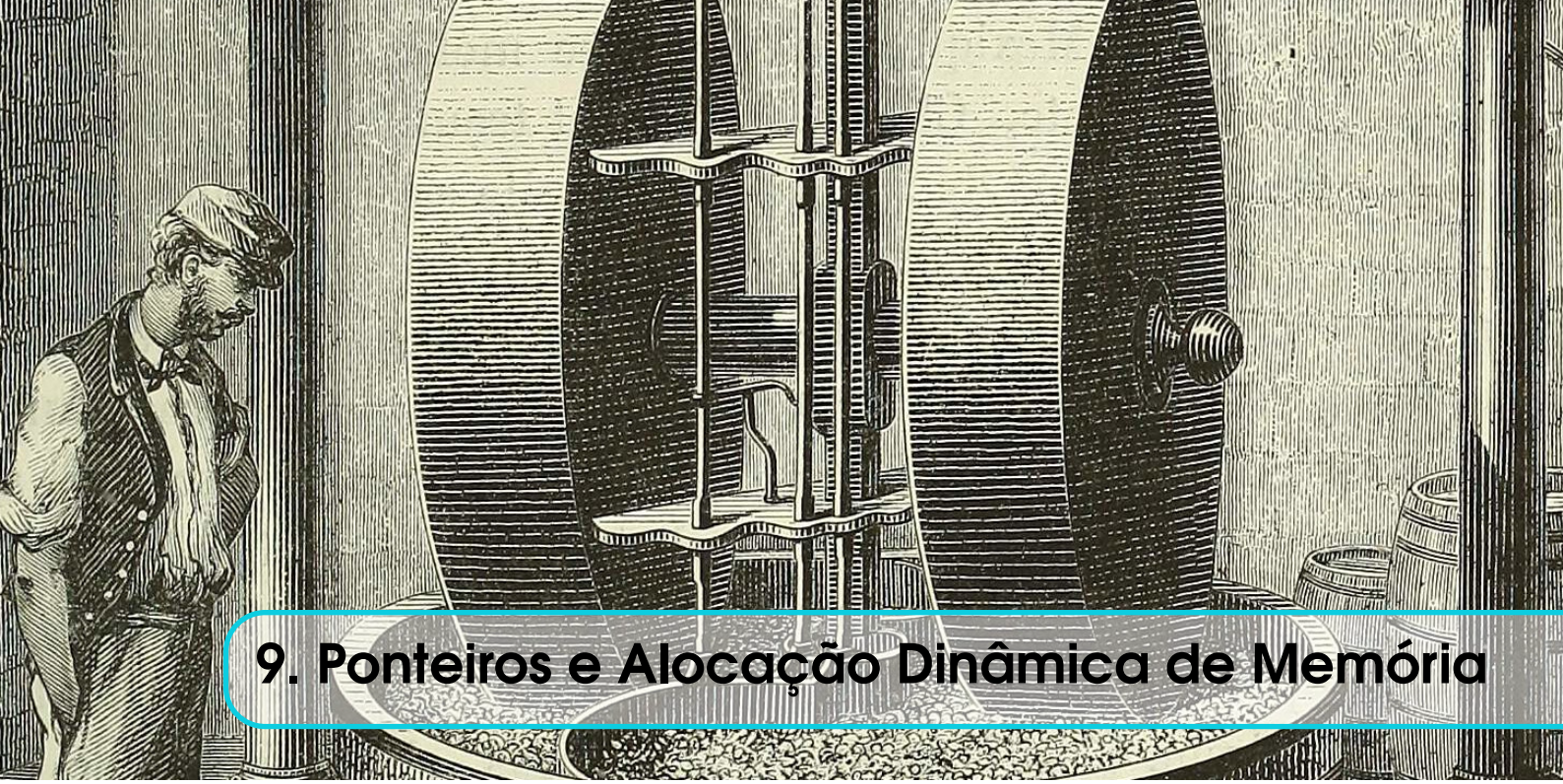
Agora, embora o uso de **typedef** torne as coisas mais claras para o leitor e mais fáceis para o programador, não é realmente necessário. O que precisamos é uma maneira de declarar um ponteiro como **p1d** sem a necessidade da palavra-chave **typedef**. Acontece que isso pode ser feito e que

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

é a declaração apropriada, ou seja, **p1d** aqui é um ponteiro para um *array* de 10 inteiros, exatamente como estava na declaração usando o tipo **Array**. Observe que isso é diferente de

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

o que tornaria **p1d** o nome de um *array* de 10 ponteiros para o tipo **int**.



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. O uso dessa abordagem permite adiar a decisão sobre o tamanho do bloco de memória necessário para armazenar um *array*, por exemplo, até o tempo de execução. Ou permite usar uma seção de memória para o armazenamento de um *array* de inteiros em um ponto no tempo, e então quando essa memória não for mais necessária, ela pode ser liberada para outros usos, como o armazenamento de um *array* de estruturas.

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

Se você estiver usando um compilador mais antigo e quiser alocar memória para um *array* de inteiros, terá que converter o ponteiro **char** retornado para um ponteiro inteiro. Por exemplo, para alocar espaço para 10 inteiros, podemos escrever:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)
{ .. A ROTINA DE ERROS VAI AQUI .. }
```

Se você estiver usando um compilador atual, **malloc()** retorna um ponteiro **void** e como um ponteiro **void** pode ser atribuído a uma variável de ponteiro de qualquer tipo de objeto, o *cast* **(int *)** mostrado acima não é necessário:

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

A dimensão do *array* pode ser determinada em tempo de execução e não é necessária em tempo de compilação. Ou seja, o 10 acima pode ser uma variável lida de um arquivo de dados ou teclado, ou calculada com base em alguma necessidade, em tempo de execução.

Por causa da equivalência entre a notação de *array* e ponteiro, uma vez que **iptr** tenha sido atribuído como acima, pode-se usar a notação de *array*. Por exemplo, pode-se escrever:

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

para definir os valores de todos os elementos para 2.

Mesmo com um entendimento razoavelmente bom de ponteiros e *arrays*, um lugar que o novato em C provavelmente tropeçará no início é na alocação dinâmica de *arrays* multidimensionais. Em geral, gostaríamos de poder acessar elementos de tais *arrays* usando notação de *array*, não notação de ponteiro, sempre que possível. Dependendo da aplicação, podemos ou não saber as duas dimensões no momento da compilação. Isso nos leva a diversas maneiras de realizar nossa tarefa.

Como vimos, ao alocar dinamicamente um *array* unidimensional, sua dimensão pode ser determinada em tempo de execução. Agora, ao usar a alocação dinâmica de *arrays* de ordem superior, nunca precisamos saber a primeira dimensão em tempo de compilação. Se precisamos saber as dimensões superiores depende de como vamos escrever o código. Aqui, discutirei vários métodos de alocação dinâmica de espaço para *arrays* bidimensionais de inteiros.

MÉTODO 1:

Uma maneira de lidar com o problema é usando a palavra-chave **typedef**. Para alocar um *array* bidimensional de inteiros, lembre-se de que as duas notações a seguir resultam na geração do mesmo código de objeto:

```
multi[lin][col] = 1;          // Usando array
*(*(multi + lin) + col) = 1; // Usando ponteiro
```

Também é verdade que as duas notações a seguir geram o mesmo código:

```
multi[lin];     // Usando array
*(multi + lin); // Usando ponteiro
```

Visto que a segunda linha deve ser avaliada como um ponteiro, a notação de *array* da primeira linha também deve ser avaliada como um ponteiro. Na realidade, **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 como um ponteiro para aquele *array* de inteiros que constituem a enésima linha de nosso array bidimensional. Ou seja, **multi** pode ser pensado como um *array* de *arrays* e **multi[n]** como um ponteiro para o enésimo *array* deste *array* de *arrays*. Aqui, a palavra ponteiro está sendo usada para representar um valor de endereço. Embora tal uso seja comum na literatura, ao ler tais declarações deve-se ter o cuidado de distinguir entre o endereço constante de um *array* e um ponteiro de variável, que é um objeto de dados em si.

Considere agora:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define COLS 5
5
6  typedef int ArrayLinha[COLS];
7  ArrayLinha* lptr;
8
9  int main(void) {
10     int nlinhas = 10;
11     lptr = malloc(nlinhas * COLS * sizeof(int));
12
13     for (int lin = 0; lin < nlinhas; lin++) {
14         for (int col = 0; col < COLS; col++) {
```

```
15              lptr[lin][col] = 42;
16          }
17      }

19      return 0;
20  }
```

Programa 9.1: Ponteiros e malloc

Aqui, assumi um compilador atual, portanto, uma conversão no ponteiro **void** retornado por **malloc()** não é necessária. Se você estivesse usando um compilador **K&R** mais antigo, teria que converter usando:

```
lptr = (ArrayLinha *)malloc(nlinhas * COLS * sizeof(int));
```

Usando esta abordagem, **lptr** tem todas as características de um nome de *array*, (exceto que **lptr** é modificável), e a notação de *array* pode ser usada em todo o resto do programa. Isso também significa que, se você pretende escrever uma função para modificar o conteúdo do *array*, deve usar **COLS** como parte do parâmetro formal dessa função, assim como fizemos ao discutir a passagem de *arrays* bidimensionais para uma função.

MÉTODO 2:

No MÉTODO 1, **lptr** acabou sendo um ponteiro para tipo "um array unidimensional de COLS inteiros". 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 características da variável **lptr** no **MÉTODO 1**, e não precisamos usar a palavra-chave *typedef*. Aqui **xptr** é um ponteiro para um *array* de inteiros e o tamanho desse *array* é dado por **#define COLS**. O posicionamento dos parênteses faz com que a notação de ponteiro predomine, embora a notação de *array* tenha precedência mais alta. ou seja, se houvéssemos escrito

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

teríamos definido **xptr** como um *array* de ponteiros contendo o número de ponteiros igual ao **#definido** por **COLS**. Isso não é a mesma coisa de forma alguma. No entanto, *arrays* de ponteiros têm seu uso na alocação dinâmica de *arrays* bidimensionais, como será visto nos próximos 2 métodos.

MÉTODO 3:

Considere o caso em que não sabemos o número de elementos em cada linha em tempo de compilação, ou seja, o número de linhas e o número de colunas devem ser determinados em tempo de execução. Uma maneira de fazer isso seria criar um *array* de ponteiros para o tipo **int** e então alocar espaço para cada linha e apontar esses ponteiros para cada linha. Considere:

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

4  int main(void) {
5      int nlinhas = 5; /* Ambas nlinhas e ncols devem ser avaliadas */
6      int ncols = 10;  /* ou lidas em tempo de execução */
7      int** linhaptr;
8      linhaptr = malloc(nlinhas * sizeof(int*));
```

```
10      if (linhaptr == NULL) {
11          puts("\nFalha ao alocar memória para ponteiros de linha.\n");
12          exit(0);
13      }

15      printf("\n\n\nIndice\t Ponteiro(hex)\t Ponteiro(dec)\t Dif.(dec)");

17      for (int linha = 0; linha < nlinhas; linha++) {
18          linhaptr[linha] = malloc(ncols * sizeof(int));

20          if (linhaptr[linha] == NULL) {
21              printf("\nFalha ao alocar memória para linha[%d]\n", linha);
22              exit(0);
23          }

25          printf("\n%d\t %p\t %ld", linha, linhaptr[linha], (long)linhaptr[linha]);

27          if (linha > 0) {
28              printf("\t %d", (int)(linhaptr[linha] - linhaptr[linha - 1]));
29          }
30      }

32      return 0;
33  }
```

Programa 9.2: Método 3

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

Para obter o array de ponteiros	1 chamada
Para obter memória para as linhas	5 chamadas
Total	6 chamadas

Se você optar por usar esta abordagem, observe que, embora você possa usar a notação de *array* para acessar elementos individuais do *array*, por exemplo, **ptrlinha[lin][col] = 42;**, isso não significa que os dados no "*array* bidimensional" são contíguos na memória.

Você pode, entretanto, usar a notação de *array* como se fosse um bloco contínuo de memória. Por exemplo, você pode escrever:

```
ptrlinha[lin][col] = 176;
```

exatamente como se **ptrlinha** fosse o nome de um *array* bidimensional criado em tempo de compilação. É claro que **lin** e **col** devem estar dentro dos limites do *array* que você criou, assim como com um *array* criado em tempo de compilação.

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

MÉTODO 4:

Neste método, alocamos um bloco de memória para armazenar primeiro o *array* inteiro. Em seguida, criamos um *array* de ponteiros para apontar para cada linha. Portanto, mesmo que o *array* de ponteiros esteja sendo usado, o *array* real na memória é contíguo. O código é parecido com este:

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

```c
int main(void) {
    int** lptr;
    int* aptr;
    int* testptr;
    int k;
    int nlinhas = 5; /* Ambas nlinhas e ncols devem ser avaliadas */
    int ncols = 8;   /* ou lidas em tempo de execução */

    /* agora alocamos a memória para o array */
    aptr = malloc(nlinhas * ncols * sizeof(int));

    if (aptr == NULL) {
        puts("\nFalha ao alocar memória para o array");
        exit(0);
    }

    /* em seguida, alocamos espaço para os ponteiros para as linhas */
    lptr = malloc(nlinhas * sizeof(int*));

    if (lptr == NULL) {
        puts("\nFalha ao alocar memória para ponteiros");
        exit(0);
    }

    /* e agora nós 'apontamos' os ponteiros */
    for (k = 0; k < nlinhas; k++) {
        lptr[k] = aptr + (k * ncols);
    }

    /* Agora ilustramos como os ponteiros de linha são incrementados */
    printf("\n\nIlustrando como os ponteiros de linha são incrementados");
    printf("\n\nIndice\t Ponteiro(hex)\t Dif.(dec)");

    for (int lin = 0; lin < nlinhas; lin++) {
        printf("\n%d\t %p", lin, lptr[lin]);

        if (lin > 0) {
            printf("\t %ld", (lptr[lin] - lptr[lin - 1]));
        }
    }

    printf("\n\nE agora imprimimos o array\n");

    for (int lin = 0; lin < nlinhas; lin++) {
        for (int col = 0; col < ncols; col++) {
            lptr[lin][col] = lin + col;
            printf("%3d", lptr[lin][col]);
        }

        putchar('\n');
    }

    puts("\n");
    /* e aqui ilustramos que estamos, de fato, lidando com um array
     * bidimensional em um bloco contíguo de memória. */
```

```
59      printf("E agora demonstramos que eles são contíguos na memória\n");
60      testptr = aptr;
61
62      for (int lin = 0; lin < nlinhas; lin++) {
63          for (int col = 0; col < ncols; col++) {
64              printf("%3d", *(testptr++));
65          }
66
67          putchar('\n');
68      }
69
70      return 0;
71  }
```

Programa 9.3: Método 4.

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

Para obter memória para o próprio array	1 chamada
Para obter memória para o array 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 encadeada que contém dados relativos ao tamanho do bloco. Mas, mais importante, com grandes arrays (várias centenas de linhas), controlar o que precisa ser liberado quando chega a hora pode ser mais complicado. Isso, combinado com a contiguidade do bloco de dados que permite a inicialização para todos os zeros usando **memset()**, parece tornar a segunda alternativa a preferida.

Como um exemplo final em *arrays* multidimensionais, ilustraremos a alocação dinâmica de um *array* tridimensional. Este exemplo ilustrará mais uma coisa a se observar ao fazer esse tipo de alocação. Pelas razões citadas anteriormente, usaremos a abordagem descrita na alternativa dois. Considere o seguinte código:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stddef.h>
4
5   const int X_DIM = 16;
6   const int Y_DIM = 5;
7   const int Z_DIM = 3;
8
9   int main(void) {
10      /* primeiro separamos a memória para o próprio array */
11      char* espaco = malloc(X_DIM * Y_DIM * Z_DIM * sizeof(char));
12
13      /* em seguida, alocamos espaco de uma matriz de ponteiros,
14       * cada um para eventualmente apontar para o primeiro
15       * elemento de um array bidimensional de ponteiros */
16      char*** Arr3D = malloc(Z_DIM * sizeof(char**));
17
18      /* e para cada um deles atribuímos um ponteiro para um
19       * array recém-alocado de ponteiros para uma linha */
20      for (int z = 0; z < Z_DIM; z++) {
21          Arr3D[z] = malloc(Y_DIM * sizeof(char*));
22
23          /* e para cada espaco neste array colocamos um
24           * ponteiro para o primeiro elemento de cada
```

```
25          * linha no espaço do array originalmente alocado */

27         for (int y = 0; y < Y_DIM; y++) {
28             Arr3D[z][y] = espaco + (z * (X_DIM * Y_DIM) + y * X_DIM);
29         }
30     }

32     /* E, agora verificamos cada endereço em nosso array 3D
33      * para ver se a indexação do ponteiro Arr3d é conduzida
34      * de maneira contínua */
35     for (int z = 0; z < Z_DIM; z++) {
36         printf("Localização do array %d é %p\n", z, *Arr3D[z]);

38         for (int y = 0; y < Y_DIM; y++) {
39             ptrdiff_t diff;
40             printf(" Array %d e linha %d iniciam em %p ", z, y, Arr3D[z][y]);
41             diff = Arr3D[z][y] - espaco;
42             printf("diff = %3ld ", diff);
43             printf("z = %d y = %d\n", z, y);
44         }
45     }

47     return 0;
48 }
```

Programa 9.4: Alocação de Array Tridimensional..

Se você seguiu este tutorial até este ponto, não deverá ter problemas para decifrar o programa 9.4 com base apenas nos comentários. No entanto, há alguns pontos que devem ser destacados. Vamos começar com a linha que diz:

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

Observe que aqui **espaço** é um ponteiro de caractere, 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 do ponteiro. E ao atribuir valores de ponteiro a variáveis de ponteiro, os tipos de dados do valor e da variável devem corresponder.

10. Ponteiros para Funções

Até este ponto, discutimos ponteiros para objetos de dados. C também permite a declaração de ponteiros para funções. Os ponteiros para funções têm diversos usos e alguns deles serão discutidos aqui.

Considere o seguinte problema real. Você deseja escrever uma função que seja capaz de ordenar virtualmente qualquer coleção de dados que possa ser armazenada em um *array*. Isso pode ser um *array* de strings, inteiros, pontos flutuantes, ou mesmo estruturas. O algoritmo de classificação pode ser o mesmo para todos. Por exemplo, pode ser um algoritmo de classificação de bolha simples, ou algoritmos mais complexos como o *shellsort* ou o *quicksort*. Usaremos uma ordenação pelo método da bolha simples, para fins de demonstração.

Sedgewick [2] descreveu a classificação por bolhas usando código C, configurando uma função que, quando passada por um ponteiro para o *array*, iria ordená-lo. Se chamarmos essa função **bubble**(), um programa de classificação é descrito por **bubble_1.c**, que se segue:

```c
#include <stdio.h>

int arr[] = { 3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
#define N sizeof(arr) / sizeof(arr[0])

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

int main(void) {
    puts("\nAntes de ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }

    bubble(arr, N);

    puts("\n\nApós ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
```

```
19      }
20
21      return 0;
22  }
23
24  void bubble(int a[], int n) {
25      for (int i = n - 1; i >= 0; i--) {
26          for (int j = 1; j <= i; j++) {
27              if (a[j - 1] > a[j]) {
28                  int t = a[j - 1];
29                  a[j - 1] = a[j];
30                  a[j] = t;
31              }
32          }
33      }
34  }
```

Programa 10.1: bubble_1.c

A ordenação pelo método da bolha é um dos tipos mais simples. O algoritmo varre o *array* do segundo ao último elemento, comparando cada elemento com o que o precede. Se aquele que o precede for maior que o elemento atual, os dois serão trocados de forma que o maior fique mais próximo do final do *array*. Na primeira passagem, isso resulta no maior elemento terminando no final do *array*. O *array* agora está limitado a todos os elementos, exceto o último e o processo repetido. Isso coloca o próximo maior elemento em um ponto que precede o maior elemento. O processo é repetido por um número de vezes igual ao número de elementos menos 1. O resultado final é um *array* ordenado.

Aqui, nossa função é projetada para classificar um *array* de inteiros. Portanto, na linha 27 estamos comparando inteiros e nas linhas 28 a 30 estamos usando armazenamento temporário de inteiros para armazenar inteiros. O que queremos fazer agora é ver se podemos converter esse código para que possamos usar qualquer tipo de dados, ou seja, não ficar restrito a inteiros.

Ao mesmo tempo, não queremos ter que analisar nosso algoritmo e o código associado a ele cada vez que o usarmos. Começamos removendo a comparação de dentro da função **bubble()** para tornar relativamente fácil modificar a função de comparação sem ter que reescrever partes relacionadas ao algoritmo real. Isso resulta em **bubble_2.c** (Programa 10.2):

```
1   #include <stdio.h>
2   /* Separando a função de comparação */
3
4   int arr[] = { 3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
5   #define N sizeof(arr) / sizeof(arr[0])
6
7   void bubble(int a[], int n);
8   int compare(int m, int n);
9
10  int main(void) {
11      puts("\nAntes de ordenar:\n");
12      for (int i = 0; i < N; i++) {
13          printf("%d ", arr[i]);
14      }
15
16      bubble(arr, N);
17
18      puts("\n\nApós ordenar:\n");
19      for (int i = 0; i < N; i++) {
```

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

    return 0;
}

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

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

Programa 10.2: bubble_2.c

Se nosso objetivo é tornar nossa função de ordenação independente do tipo de dados, uma maneira de fazer isso é usar ponteiros para o tipo *void* para apontar para os dados em vez de usar o tipo de dados inteiro. Para começar nessa direção, vamos modificar algumas coisas para que os ponteiros possam ser usados. Para começar, usaremos ponteiros para o tipo inteiro (Programa 10.3).

```
#include <stdio.h>

int arr[] = {3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
#define N sizeof(arr) / sizeof(arr[0])

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

int main(void) {
    puts("\nAntes de ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }

    bubble(arr, N);

    puts("\n\nApós ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

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

```
                int t = p[j - 1];
                p[j - 1] = p[j];
                p[j] = t;
            }
        }
    }
}

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

Programa 10.3: bubble_3.c

Observe as mudanças. Agora estamos passando um ponteiro para um inteiro (ou array de inteiros) para **bubble()**. E de dentro de *bubble* estamos passando ponteiros para os elementos do array que queremos comparar com nossa função de comparação. E, é claro, estamos desreferenciando esses ponteiros em nossa função **compare()** para fazer a comparação real. Nossa próxima etapa será converter os ponteiros em **bubble()** em ponteiros para o tipo *void*, de modo que a função se torne mais insensível ao tipo. Isso é mostrado em bubble_4 (Programa 10.4).

```
#include <stdio.h>

int arr[] = { 3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
#define N sizeof(arr) / sizeof(arr[0])

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

int main(void) {
    puts("\nAntes de ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }

    bubble(arr, N);

    printf("\n\nArray ordenado\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

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

Programa 10.4: bubble_4.c

Observe que, ao fazer isso, em **compare()** tivemos que introduzir a conversão dos tipos de ponteiro *void* passados para o tipo real que está sendo classificado. Mas, como veremos mais tarde, está tudo bem. E como o que está sendo passado para **bubble()** ainda é um ponteiro para um *array* de inteiros, tivemos que converter esses ponteiros para ponteiros nulos quando os passamos como parâmetros em nossa chamada para **compare()**.

Agora abordamos o problema do que passamos para **bubble()**. Queremos tornar o primeiro parâmetro dessa função um ponteiro *void* também. Mas, isso significa que dentro de **bubble()** precisamos fazer algo sobre 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 o que quer que substituamos **t**).

Atualmente, em **bubble_4.c**, o conhecimento em **bubble()** quanto ao 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 o tipo inteiro. Se quisermos usar **bubble()** para classificar qualquer tipo de dados, precisamos fazer desse ponteiro um ponteiro para o tipo *void*. Mas, ao fazer isso, perderemos informações sobre o tamanho dos elementos individuais dentro do *array*. Portanto, em **bubble_5.c** adicionaremos um parâmetro separado para lidar com essas informações de tamanho.

Essas mudanças, de **bubble_4.c** para **bubble_5.c** (Programa 10.5), são, talvez, um pouco mais extensas do que as que fizemos no passado. Portanto, compare os dois módulos cuidadosamente para verificar as diferenças.

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

long arr[] = { 3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
#define N sizeof(arr) / sizeof(arr[0])

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

int main(void) {
    puts("\nAntes de ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%ld ", arr[i]);
    }

    bubble(arr, sizeof(long), N);

    printf("\n\nArray ordenado\n");
    for (int i = 0; i < N; i++) {
        printf("%ld ", arr[i]);
    }

    return 0;
}

void bubble(void* p, size_t tamanho, int n) {
```

```
27      unsigned char buf[8];
28      unsigned char* bp = p;
29
30      for (int i = n - 1; i >= 0; i--) {
31          for (int j = 1; j <= i; j++) {
32              if (compare((void*)(bp + tamanho * (j - 1)),
33                          (void*)(bp + j * tamanho))) { /* 1 */
34              /*  t = p[j - 1];    */
35                  memcpy(buf, bp + tamanho * (j - 1), tamanho);
36              /*  p[j - 1] = p[j]; */
37                  memcpy(bp + tamanho * (j - 1), bp + j * tamanho, tamanho);
38              /*  p[j] = t;        */
39                  memcpy(bp + j * tamanho, buf, tamanho);
40              }
41          }
42      }
43  }
44
45  int compare(void* m, void* n) {
46      long* m1, *n1;
47
48      m1 = (long*)m;
49      n1 = (long*)n;
50
51      return (*m1 > *n1);
52  }
```

Programa 10.5: bubble_5.c

Observe que alterei o tipo de dados do *array* de **int** para **long** para ilustrar as alterações necessárias na função **compare**(). Em **bubble**(), eliminei a variável **t** (que teríamos que mudar do tipo *int* para o tipo *long*). Eu adicionei um *buffer* de tamanho 8 caracteres não sinalizados, que é o tamanho necessário para conter um *long* (isso mudará novamente em futuras atualizações neste código). O ponteiro de caracteres não sinalizados ***bp** é usado para apontar para a base do *array* a ser ordenado, ou seja, para o primeiro elemento desse *array*.

Também tivemos que modificar o que passamos para **compare**() e como fazemos a troca de elementos que a comparação indica que precisam ser trocados. O uso de **memcpy**() e notação de ponteiro em vez de notação de array contribui para essa redução na sensibilidade do tipo.

Novamente, fazer uma comparação cuidadosa de **bubble_5.c** com **bubble_4.c** pode resultar em uma melhor compreensão do que está acontecendo e por quê.

Vamos agora para **bubble_6.c** (Programa 10.6), onde usamos a mesma função **bubble**() que usamos em **bubble_5.c** para classificar strings em vez de inteiros longos. É claro que temos que mudar a função de comparação, pois o meio pelo qual as strings são comparadas é diferente daquele pelo qual inteiros longos são comparados. E, em **bubble_6.c**, excluímos as linhas dentro de **bubble**() que foram comentadas em **bubble_5.c**.

```
1   #include <stdio.h>
2   #include <string.h>
3
4   #define MAX_BUF 256
5
6   char arr2[][20] = {
7       "Mickey Mouse",
8       "Pato Donald",
9       "Minnie Mouse",
```

```
    "Pateta",
    "Ted Jensen",
    "João Araujo"
};
#define N sizeof(arr2) / sizeof(arr2[0])

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

int main(void) {
    puts("\nAntes de ordenar:\n");
    for (int i = 0; i < N; i++) {
        printf("%s\n", arr2[i]);
    }

    bubble(arr2, 20, N);

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

    return 0;
}

void bubble(void* p, int tamanho, int n) {
    unsigned char buf[MAX_BUF];
    unsigned char* bp = p;

    for (int i = n - 1; i >= 0; i--) {
        for (int j = 1; j <= i; j++) {
            int k = compare((void*)(bp + tamanho * (j - 1)),
                            (void*)(bp + j * tamanho));

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

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

Programa 10.6: bubble_6.c

Mas, o fato de **bubble()** não ter sido alterado em relação ao usado em **bubble_5.c** indica que essa função é capaz de classificar uma ampla variedade de tipos de dados. O que falta fazer é passar para **bubble()** o nome da função de comparação que queremos usar para que possa ser verdadeiramente universal. Assim como o nome de um *array* é o endereço do primeiro elemento do *array* no segmento de dados, o nome de uma função indica o endereço dessa função no segmento

de código. Portanto, precisamos usar um ponteiro para uma função. Neste caso, a função de comparação.

Os ponteiros para funções devem corresponder às funções apontadas no número e nos tipos dos parâmetros e no tipo do valor de retorno. Em nosso caso, declaramos nosso ponteiro de função como:

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

Observe que se tivéssemos escrito:

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

teríamos um protótipo de função para uma função que retornasse um ponteiro para o tipo **int**. Isso ocorre porque em C o operador parênteses () tem uma precedência mais alta do que o operador ponteiro *. Colocando o parêntese ao redor da string **(* fptr)**, indicamos que estamos declarando um ponteiro de função.

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

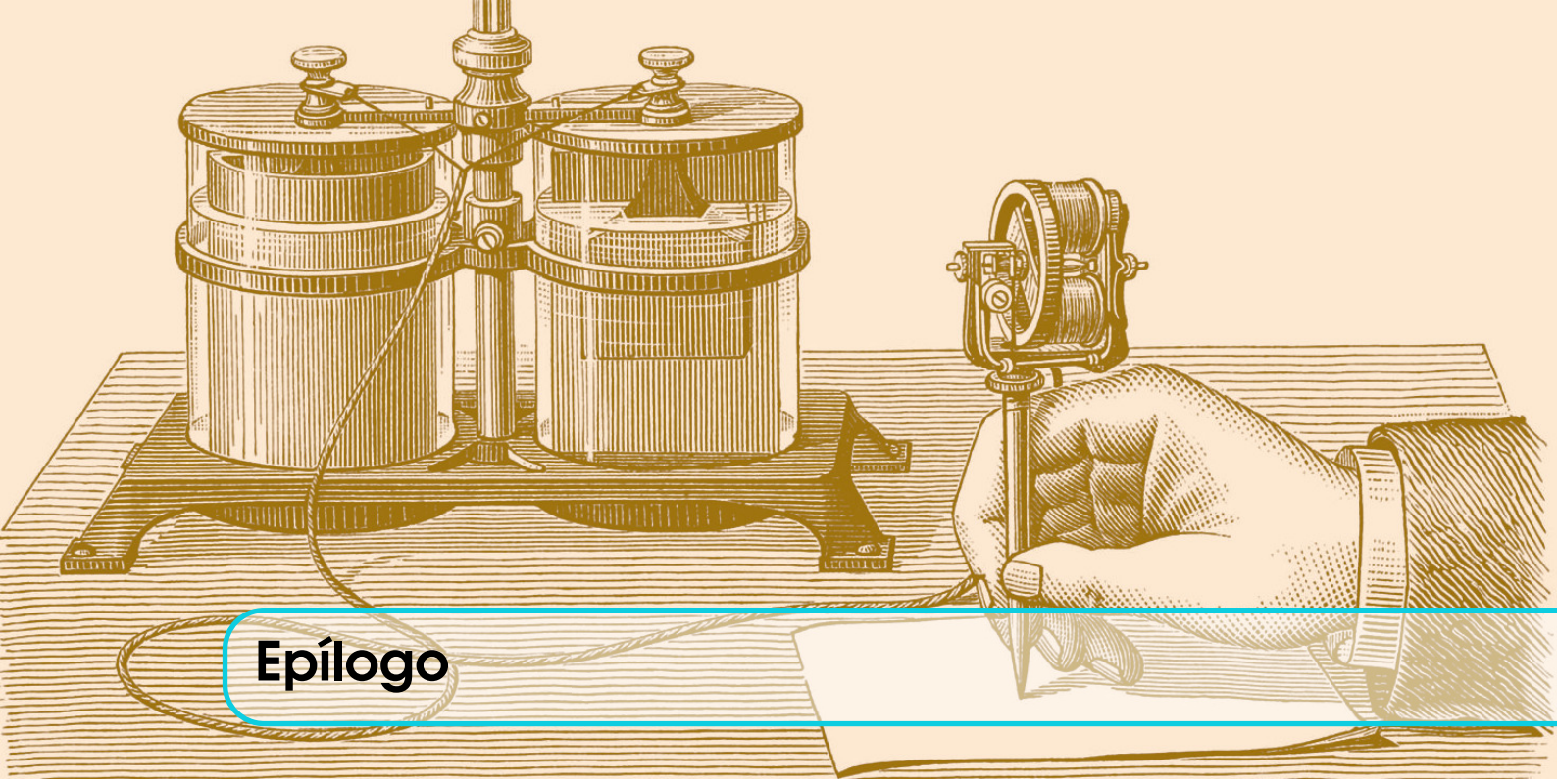
```
void bubble(void *p, int tamanho, int n,
            int(*fptr)(const void *, const void *));
```

Quando chamamos **bubble()**, inserimos o nome da função de comparação que queremos usar. **bubble_7.c** (Programa 10.7) ilustra como essa abordagem permite o uso da mesma função **bubble()** para classificar diferentes tipos de dados.

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

#define MAX_BUF 256

long arr[] = { 3, 6, 1, 2, 3, 8, 4, 1, 7, 2};
#define N1 sizeof(arr) / sizeof(arr[0])
char arr2[][20] = { "Mickey Mouse",
                    "Pato Donald",
                    "Minnie Mouse",
                    "Pateta",
                    "Ted Jensen",
                    "João Araujo"
                  };
#define N2 sizeof(arr2) / sizeof(arr2[0])

void bubble(void* p, int tamanho, 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 de ordenar:\n");
    for (i = 0; i < N1; i++) { /* mostra os long ints */
        printf("%ld ", arr[i]);
    }

    puts("\n");

    for (i = 0; i < N2; i++) { /* mostra as strings */
        printf("%s\n", arr2[i]);
```

```
33      }
34
35      bubble(arr, 8, N1, compare_long); /* ordena os longs */
36      bubble(arr2, 20, N2, compare_string); /* ordena as strings */
37
38      puts("\n\nApós ordenar:\n");
39      for (i = 0; i < N1; i++) { /* mostra os longs ordenados */
40          printf("%ld ", arr[i]);
41      }
42
43      putchar('\n');
44
45      for (i = 0; i < N2; i++) { /* mostra as strings ordenadas */
46          printf("%s\n", arr2[i]);
47      }
48
49      return 0;
50  }
51
52  void bubble(void* p, int tamanho, int n,
53              int(*fptr)(const void*, const void*)) {
54      unsigned char buf[MAX_BUF];
55      unsigned char* bp = p;
56
57      for (int i = n - 1; i >= 0; i--) {
58          for (int j = 1; j <= i; j++) {
59              int k = fptr((void*)(bp + tamanho * (j - 1)),
60                           (void*)(bp + j * tamanho));
61
62              if (k > 0) {
63                  memcpy(buf, bp + tamanho * (j - 1), tamanho);
64                  memcpy(bp + tamanho * (j - 1), bp + j * tamanho, tamanho);
65                  memcpy(bp + j * tamanho, buf, tamanho);
66              }
67          }
68      }
69  }
70
71  int compare_string(const void* m, const void* n) {
72      char* m1 = (char*)m;
73      char* n1 = (char*)n;
74      return (strcmp(m1, n1));
75  }
76
77  int compare_long(const void* m, const void* n) {
78      long* m1, *n1;
79      m1 = (long*)m;
80      n1 = (long*)n;
81      return (*m1 > *n1);
82  }
```

Programa 10.7: bubble_7.c



Epílogo

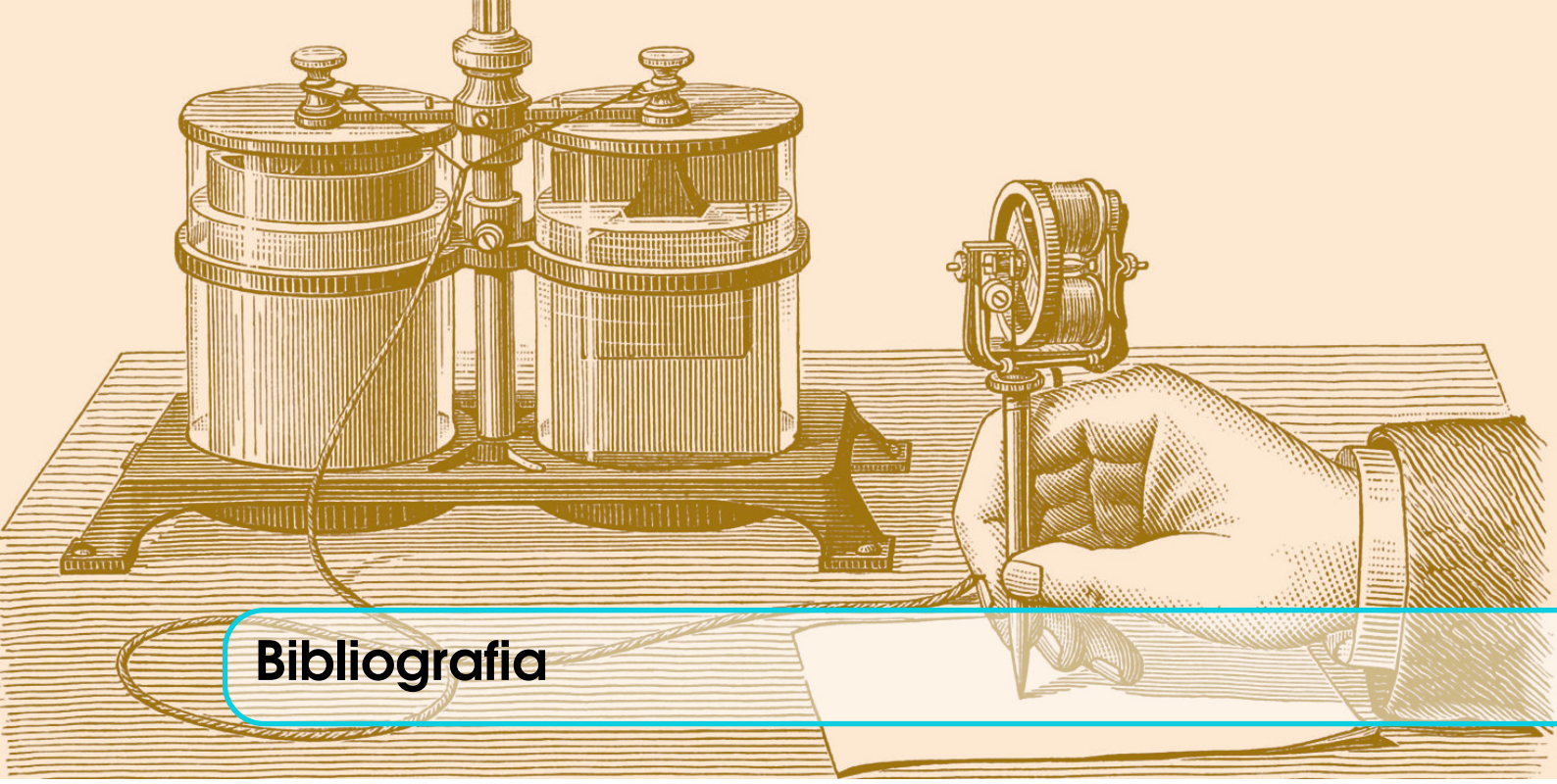
Escrevi este material em português a partir do excelente material fornecido por Ted Jensen. Tentei contactar Ted pelos endereços fornecidos, mas todas as tentativas foram infrutíferas. Pelos meus cálculos, Ted deve estar com mais de 80 anos no momento que faço esta tradução/atualização/adaptação. Espero que esteja bem, apenas afastado da Internet.

Espero que este material em português seja útil para muitos estudantes e professores, e reforçando meu pedido, se este material foi útil para você, me envia uma mensagem, uma crítica, um elogio, uma correção necessária. É sempre bom saber que outros reconhecem nosso trabalho.

Pague-me um café

Não é necessário nenhum pagamento por este trabalho. Um email é suficiente, mas é claro, se você quer contribuir financeiramente para que eu faça mais livros como este, aceito contribuições por **PIX**, com a vantagem que você pode enviar mensagem pelo próprio PIX. Minha chave é meu email:
`araujo@eng.uerj.br`

João Araujo
Outubro de 2021



Bibliografia

[1] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709 (ver página 12).

[2] Robert Sedgewick. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*. 3rd. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201314525 (ver página 49).