

# Ponteiros e Aplicações

*Everything should be made as simple as possible, but not simpler.*

Albert Einstein

## OBJETIVOS

- Entender e saber como utilizar ponteiros
- Compreender a relação e o uso de ponteiros e memória dinâmica
- Aprender e explorar ponteiros como argumento de funções
- Entender e implementar classes lista, pilha e fila

No capítulo anterior, você estudou herança e como ela pode ser empregada no desenvolvimento de programas orientados a objetos quando é necessário reutilizar código, além de tornar o código mais bem estruturado. Observe que as características mais importantes em uma linguagem de programação são a flexibilidade e a eficiência de seu uso. Nesse sentido, ponteiros são um recurso fundamental da linguagem C++ que oferece ao programador a maior flexibilidade para manipulação de dados. Um ponteiro é uma variável que armazena o endereço de outra variável. Na realidade, um ponteiro é um recurso poderoso da linguagem C++ que permite ao programador criar e manipular (isto é, fazer o processamento) de estruturas de dados dinâmicas que podem sofrer modificação, por exemplo, de tamanho e forma ao longo do tempo. Este capítulo explora esse tópico e busca responder a questões como: como utilizar ponteiros? Como fazer manipulação (alocação e desalocação) dinâmica de memória? Como criar lista, pilha e fila? Responder a essas e outras questões é o objetivo deste capítulo e, para tanto, os exemplos usados ilustram situações onde usar ponteiros é apropriado.

## 10.1. INTRODUÇÃO A PONTEIROS

### 10.1.1. Ponteiros

Ponteiros compreendem um dos recursos mais poderosos da linguagem C++. Um ponteiro é uma variável especial que armazena o endereço de outra variável (que contém informação).

### 10.1.2. Entendendo Ponteiros

Em um programa, toda informação manipulada pelo programa, sejam dados ou instruções, reside na memória (do computador) em determinado endereço e ocupa uma quantidade definida de bytes. Assim, quando você executa o programa, essas variáveis residem em endereços específicos. No entanto, quando trabalha com linguagem de alto nível, como C++ ou Java, você não está interessado em saber os endereços específicos de cada uma das variáveis que usa no programa. Tudo isso é tratado de forma transparente pelo compilador e ambiente de execução do programa (ou runtime system).

É importante observar que, conceitualmente, cada variável em um programa constitui simplesmente um nome para um determinado endereço de memória. Dessa forma, em um programa é muito mais fácil e interessante manipular os dados usando esse nome (da variável) do que lidando com endereços numéricos específicos de memória como, por exemplo, 0x27ff44 e 0x27ff40. Para entender melhor, vamos examinar o exemplo a seguir.

**Praticando um Exemplo.** Escreva um programa que declare e inicialize duas variáveis inteiras *x* e *y*. Em seguida, você deve exibir o conteúdo dessas variáveis e seus respectivos endereços. Para exibir, por exemplo, o conteúdo do endereço da variável *x*, você deve utilizar a seguinte sintaxe:

```
cout << "Endereco de x = " << &x;
```

Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 10.1.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar o acesso a endereço de variaveis
4.
5. int main()
6. {
7.
8.     int x = 11; // declaracao e inicializacao de variaveis
```

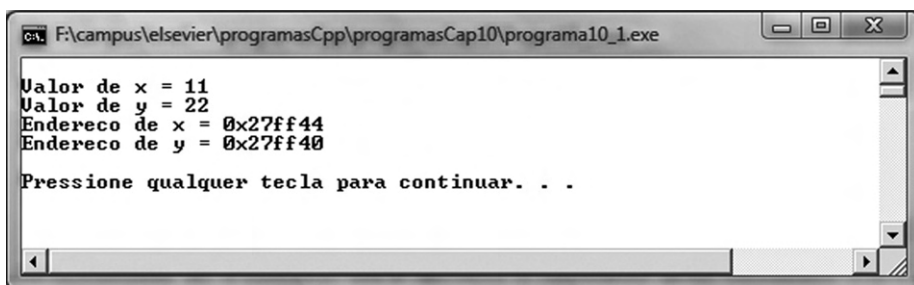
```

9.   int y = 22;
10.  cout << "\nValor de x = " << x;
11.  cout << "\nValor de y = " << y;
12.  cout << "\nEndereco de x = " << &x;
13.  cout << "\nEndereco de y = " << &y;
14.  cout << endl << endl;
15.  system("PAUSE");
16.  return 0;
17. }

```

**Listagem 10.1.**

O programa da Listagem 10.1 declara e inicializa duas variáveis  $x$  e  $y$  e, em seguida exibe seus valores e respectivos endereços. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.1.



**Figura 10.1 – Saída do programa da Listagem 10.1.**

Observe que a Figura 10.1 mostra os conteúdos de memória no instante em que o programa é executado. Além disso, você também vê os endereços reais de memória de cada uma das variáveis  $x$  e  $y$ .

---

■ É importante observar que, quando desenvolve uma solução para um problema, você pode precisar trabalhar com endereços dos dados (o que é considerado um recurso de “baixo nível”, uma vez que requer manipulação de endereços de variáveis e alocação e desalocação dinâmica). Em tal situação, o uso de ponteiros é essencial.

## 10.2. OPERADORES ENDEREÇO E PONTEIRO

### 10.2.1. Operador de Endereço

Em C++, o endereço de uma variável ou objeto pode ser referenciado fazendo uso do operador de endereço `&`. Esse operador foi apresentado e utilizado no Capítulo 5 (Seção 5.4) quando foi abordado a passagem por referência em funções. Lembre-se

de que, quando coloca o operador endereço & no tipo de dado de um parâmetro (formal) do protótipo de uma função e cabeçalho da função, esse parâmetro recebe o endereço do correspondente argumento quando a função é chamada.

---

■ *É importante não confundir o operador de endereço &, que precede o nome de uma variável em uma declaração de variável, com o operador de referência &, que segue o nome do tipo de dado em um protótipo ou definição de uma função.*

---

Usar endereços de memória simplesmente tem uma aplicação um tanto limitada. Não resta dúvida de que é ótimo saber que você pode encontrar um determinado dado em um endereço de memória, como ilustrado no programa da Listagem 10-1. Todavia, existe pouca utilidade em mostrar os endereços de memória de uma variável ou objeto. Então, por que ponteiro é um recurso poderoso?

Esse poder advém do fato de que uma variável do tipo ponteiro armazena valores de endereço. Nos capítulos anteriores, você explorou programas com diversos tipos de variáveis (int, double, char) que armazenam dados desses tipos. Agora, quando se fala de endereços, esses endereços também são armazenados de maneira similar. Portanto, uma variável que armazena um valor de endereço é denominada variável ponteiro ou simplesmente ponteiro.

C++ permite armazenar o endereço de memória em um tipo especial de variável ou objeto, o ponteiro. Assim, quando você define um ponteiro, o tipo de variável ou objeto para o qual ele aponta deve também ser definido. O tipo de variável ou objeto para o qual o ponteiro aponta é referenciado como sendo o tipo base do ponteiro. Em outras palavras, esse tipo base do ponteiro determina como a variável ou objeto será interpretado. Considere a situação a seguir.

```
int x1 = 10; // declara e inicializa a variável x1
int *x1Ptr; // declara x1Ptr como variável ponteiro
x1Ptr = &x1; // atribui o conteúdo de x1 a x1Ptr
cout << *x1Ptr; // exibe conteúdo de x1(10), que é apontado por x1Ptr
```

Observe que a primeira instrução declara uma variável *x1* e, em seguida, uma variável ponteiro (*\*x1Ptr*) é declarada. Na terceira instrução, o conteúdo de *x1* é atribuído a *\*x1Ptr*. Já a quarta instrução faz com que o valor 10 seja exibido na tela porque você solicita que o conteúdo aponte *x1Ptr*. Para entender mais, vamos examinar o próximo exemplo.

**Praticando um Exemplo.** Considere a situação em que você não conhece o nome de uma variável, mas sabe seu endereço. É possível acessar o conteúdo dessa variável?

A resposta é sim. Para ilustrar essa situação, escreva um programa que declare e inicialize duas variáveis inteiras *x* e *y*. Em seguida, você deve criar duas variáveis do tipo ponteiro (*xPtr* e *yPtr*) e atribuir o endereço dessas variáveis (*x* e *y*) a duas variáveis do tipo ponteiro, utilizando a seguinte sintaxe:

$$xPtr = \&x;$$

Para verificar que os valores *\*xPtr* e *\*yPtr* são, respectivamente, iguais aos de *x* e *y*, exiba esses valores na tela. (O mesmo ocorre com os valores *xPtr* e *yPtr* que são, respectivamente, iguais aos de *&x* e *&y*.) Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 10.2.

```

1. #include <iostream>
2. using namespace std;
3.
4. // Programa para ilustrar uso de ponteiros
5.
6. int main()
7. {
8.     int x = 11; // declara e inicializa a variável x
9.     int y = 22; // declara e inicializa a variável y
10.    cout << "\nValor de x = " << x;
11.    cout << "\nValor de y = " << y;
12.
13.    int *xPtr; // declara xPtr como variável ponteiro
14.    int *yPtr; // declara yPtr como variável ponteiro
15.    xPtr = &x; //atribui o conteúdo de x a xPtr
16.    yPtr = &y; //atribui o conteúdo de y a yPtr
17.
18.    cout << "\n\nEndereco de x = " << &x;
19.    cout << "\nEndereco de y = " << &y;
20.    cout << "\n\nValor de xPtr = " << xPtr;
21.    cout << "\nValor de yPtr = " << yPtr;
22.    cout << "\n\nValor de *xPtr = " << *xPtr;
23.    cout << "\nValor de *yPtr = " << *yPtr;
24.    cout << endl << endl;
25.    system("PAUSE");
26.    return 0;
27. }
```

**Listagem 10.2.**

O programa da Listagem 10.2 declara e inicializa duas variáveis  $x$  e  $y$  e, em seguida, atribui os endereços dessas variáveis para  $xPtr$  e  $yPtr$ , que apontam, respectivamente, para  $x$  e  $y$ . Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.2.

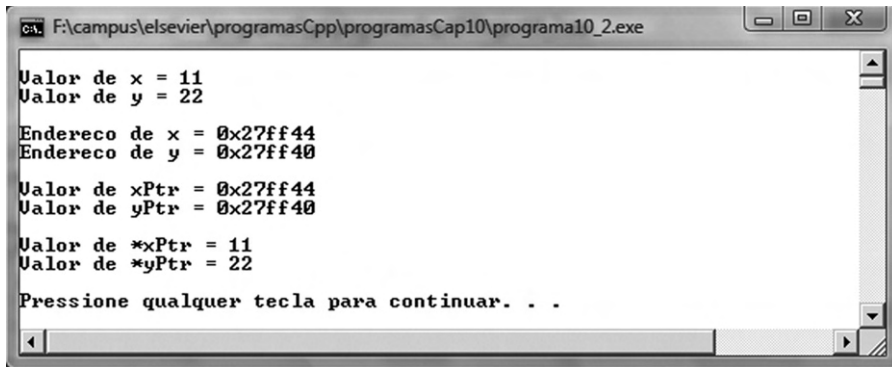


Figura 10.2 – Saída do programa da Listagem 10.2.

### 10.3. OPERAÇÕES COM PONTEIROS

O programa anterior mostrou que, uma vez que um ponteiro tenha o endereço de uma variável, você pode ter acesso ao valor armazenado naquela variável utilizando o operador  $*$  seguido pelo nome do ponteiro. Por exemplo, se  $ptrX$  é o ponteiro para a variável  $x$ , você pode usar  $*ptrX$  para acessar o valor da variável  $x$ . Para entender melhor, vamos examinar o próximo exemplo.

**Praticando um Exemplo.** Modifique o programa anterior de modo que você acesse e altere os valores das variáveis  $x$  e  $y$  usando seus respectivos ponteiros (em vez das próprias variáveis). Assim, inicialmente, seu programa deve declarar e inicializar duas variáveis inteiras  $x$  e  $y$ . Em seguida, você deve criar duas variáveis do tipo ponteiro ( $xPtr$  e  $yPtr$ ) e atribuir o endereço dessas variáveis ( $x$  e  $y$ ) a duas variáveis do tipo ponteiro. Para testar seu programa, inicialize as variáveis  $x$  e  $y$  com os valores 1 e 3, respectivamente. Multiplique  $x$  por 10, incremente  $y$  de 20 e some os valores de  $x$  e  $y$ , tudo isso usando os ponteiros. Por fim, seu programa deve exibir os resultados obtidos. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 10.3.

```
1. #include <iostream>
2. using namespace std;
3.
4. // Programa para ilustrar operacao com ponteiros
5.
6. int main()
```

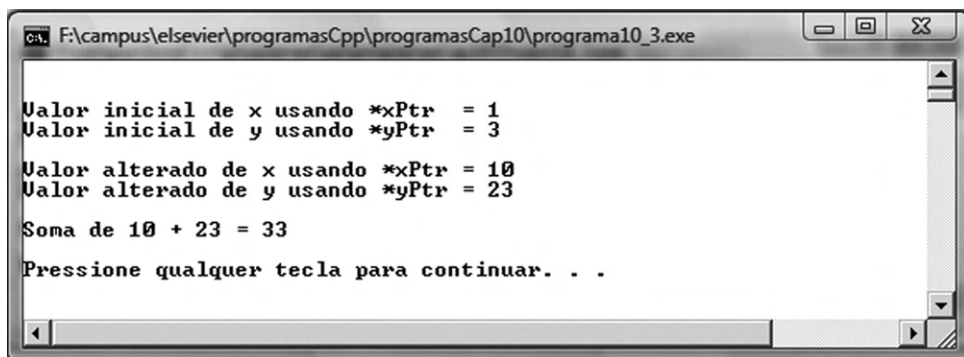
```

7.  {
8.    int x = 1; // declara e inicializa a variável x
9.    int y = 3; // declara e inicializa a variável y
10.   int *xPtr; // declara x1Ptr como variável ponteiro
11.   int *yPtr; // declara x1Ptr como variável ponteiro
12.
13.   xPtr = &x; //atribui o conteúdo de x a xPtr
14.   yPtr = &y; //atribui o conteúdo de y a yPtr
15.   cout << "\n\nValor inicial de x usando *xPtr = " << *xPtr;
16.   cout << "\nValor inicial de y usando *yPtr = " << *yPtr;
17.
18.   *xPtr *= 10;
19.   *yPtr += 20;
20.   cout << "\n\nValor alterado de x usando *xPtr = " << *xPtr;
21.   cout << "\nValor alterado de y usando *yPtr = " << *yPtr;
22.
23.   cout << "\n\nSoma de " << *xPtr << " + " << *yPtr << " = "
      << *xPtr + *yPtr;
24.   cout << endl << endl;
25.   system("PAUSE");
26.   return 0;
27. }

```

**Listagem 10.3.**

O programa da Listagem 10.3 declara e inicializa duas variáveis  $x$  e  $y$ , em seguida, uso os ponteiros  $*xPtr$  e  $*yPtr$  para realizar operações e exibir resultados obtidos. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.3.



```

F:\campus\elsevier\programasCpp\programasCap10\programa10_3.exe
Valor inicial de x usando *xPtr = 1
Valor inicial de y usando *yPtr = 3
Valor alterado de x usando *xPtr = 10
Valor alterado de y usando *yPtr = 23
Soma de 10 + 23 = 33
Pressione qualquer tecla para continuar. . .

```

**Figura 10.3** – Saída do programa da Listagem 10.3.

As linguagens C e C++ oferecem um suporte especial para nomes de arrays. O compilador interpreta o nome de um array como o endereço de seu primeiro elemento. Assim, se *a* é um array, as expressões *&a[0]* e *a* são equivalentes. Note que, uma vez que tenha o endereço de um dado, você pode obter esse dado. Esse conhecimento de endereço de memória de uma variável ou array permite manipular o conteúdo desses itens usando ponteiros. Para entender mais, vamos examinar o próximo exemplo.

**Praticando um Exemplo.** Escreva um programa que crie um array de double e que permita entrar com até 100 valores. Entretanto, quando seu programa ler os dados digitados pelo usuário, ele deve usar a instrução

```
cin >> *(a + j);           // usando *(a+j) para armazenar em a[j]
```

E, quando acumular os valores lidos, seu programa deve usar:

```
soma += *(aPtr + j);       // usando *aPtr para acessar a[j]
```

Por fim, seu programa deve exibir a soma e a média dos valores digitados. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 10.4.

```
1. #include <iostream>
2. const int MAX = 100;
3. using namespace std;
4.
5. // Programa para ilustrar uso de ponteiros para arrays
6. int main()
7. {
8.     double a[MAX]; // declara array de double
9.     double *aPtr = a; // declara *aPtr
10.    double media;
11.    double soma = 0.0;
12.    int n;
13.
14.    do {
15.        cout << "\nInforme a quantidade de elementos do array de
            '2 a 100': ";
16.        cin >> n;
17.        cout << endl << endl;
18.    } while (n < 2 || n > MAX);
19.
20.    for (int j = 0; j < n; j++)
21.    {
```



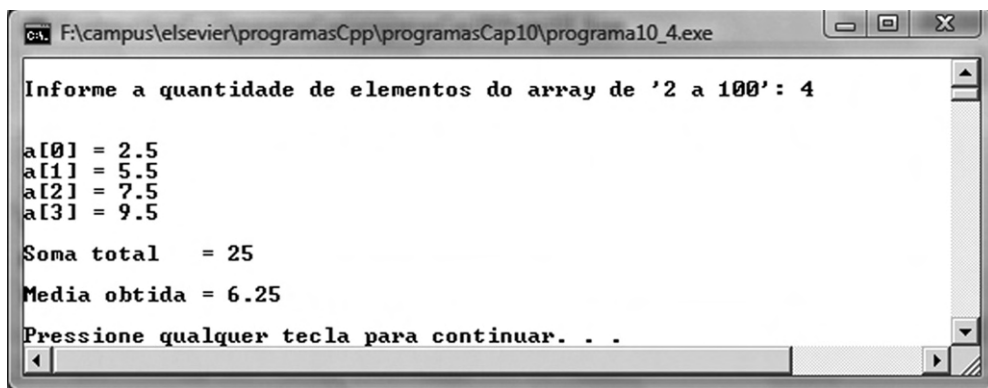
```

22.     cout << "a[" << j << "]" = ";
23.     cin >> *(a + j); // usando *(a+j) para armazenar em a[j]
24. }
25.
26. for (int j = 0; j < n; j++)
27.     soma += *(aPtr + j); // usando *aPtr para acessar a[j]
28.
29. media = soma / n;
30. cout << "\nSoma total = " << soma;
31. cout << "\n\nMedia obtida = " << media << endl << endl;
32. system("PAUSE");
33. return 0;
34. }

```

**Listagem 10.4.**

O programa da Listagem 10.4 cria um array que pode armazenar até 100 valores do tipo double. O usuário é solicitado a digitar a quantidade e os valores, e depois o programa exibe a soma total e a média. Entretanto, esse programa faz uso de ponteiros para ter acesso aos elementos do array. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.4.



**Figura 10.4** – Saída do programa da Listagem 10.4.

## 10.4. OBJETOS DINÂMICOS COM PONTEIROS

Os programas que você estudou até o momento têm tido o espaço de armazenamento das variáveis, arrays ou objetos definidos em tempo de compilação, isto é, antes da execução dos programas. Dessa forma, quando você executa esses programas, o espaço de armazenamento (ou memória) desses itens já está pré-definido. Entretanto, há situações em que é preciso alocar memória dinamicamente.

### 10.4.1. Alocação Dinâmica de Memória

Alocação dinâmica de memória é um mecanismo muito útil na solução de diversos tipos de problemas que exigem arrays de tamanho grande. A alocação dinâmica de memória permite que o espaço de armazenamento de itens (como, por exemplo, variáveis e arrays) do programa seja dinamicamente tratado, ou seja, em tempo de execução (do programa).

A linguagem C++ oferece dois operadores, *new* e *delete*, que servem para controlar a alocação e a desalocação dinâmica de memória, respectivamente. Para fazer uso desses operadores e alocar e desalocar dinamicamente um array, você deve seguir a seguinte sintaxe:

```
ponteiroArray = new tipoDado[tamanhoArray];  
delete [ ] ponteiroArray;
```

---

■ *Note que uma variável é um nome que rotula um endereço de memória. Nesse sentido, usar uma variável em um programa permite ter acesso ao conteúdo do endereço de memória associado a essa variável, especificando seu nome. Contudo, vale ressaltar que tal variável nada mais é do que um nome que aponta para um determinado endereço de memória, ou seja, um ponteiro.*

Nessas instruções, o operador *new* retorna o endereço do array que foi dinamicamente alocado, enquanto o operador *delete[ ]* remove o array que foi dinamicamente alocado e que é acessado por um ponteiro. Para entender melhor vamos examinar o exemplo a seguir.

```
1. #include <iostream>  
2. const int MAX = 10;  
3. using namespace std;  
4. // Programa para ilustrar alocacao dinamica de memoria  
5.  
6. int main()  
7. {  
8.     int *intPtr;  
9.     intPtr = new int[MAX]; // cria um array dinamico usando new  
10.    for (int j = 0; j < MAX; j++)  
11.        intPtr[j] = j * j; // armazena o produto j*j no array di-  
12.        namico intPtr[]  
13.    for (int j = 0; j < MAX; j++)  
14.        cout << "\nValor de intPtr[" << j << "] = "
```

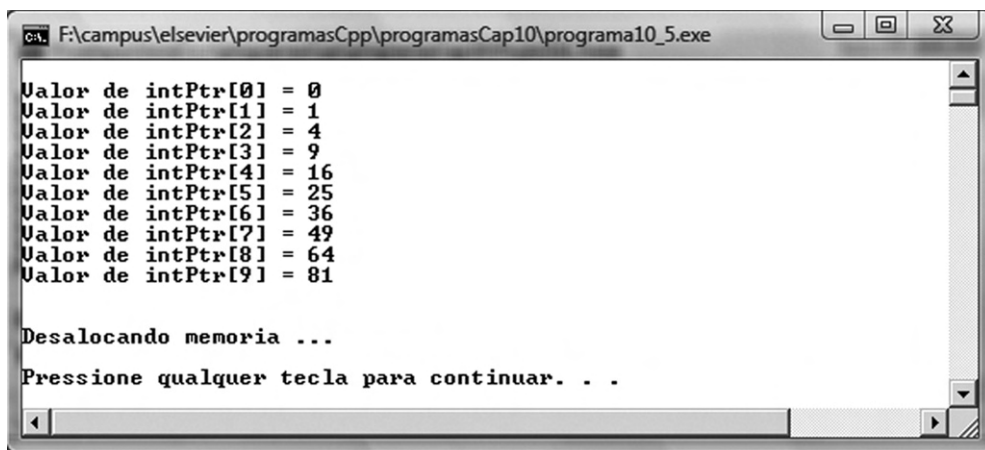
```

15.     << *(intPtr + j); // acessa o conteudo do array dinamico
        usando *(intPtr+j)
16.     cout << endl << endl;
17.
18.     delete [ ] intPtr; // desaloca o array dinamico acessado
        pelo ponteiro intPtr
19.     cout << "\nDesalocando memoria...\n\n";
20.     system("PAUSE");
21.     return 0;
22. }

```

**Listagem 10.5.**

O programa da Listagem 10.5 utiliza o operador `new`, na linha 9, para criar um array dinâmico. Na linha 15, o conteúdo do array dinâmico é acessado usando `*(intPtr+j)`. Depois, o conteúdo do array dinâmico que é acessado por `intPtr` é desalocado. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.5.



```

c:\ F:\campus\elsevier\programasCpp\programasCap10\programa10_5.exe
Valor de intPtr[0] = 0
Valor de intPtr[1] = 1
Valor de intPtr[2] = 4
Valor de intPtr[3] = 9
Valor de intPtr[4] = 16
Valor de intPtr[5] = 25
Valor de intPtr[6] = 36
Valor de intPtr[7] = 49
Valor de intPtr[8] = 64
Valor de intPtr[9] = 81

Desalocando memoria ...

Pressione qualquer tecla para continuar. . .

```

**Figura 10.5** – Saída do programa da Listagem 10.5.

**Praticando um Exemplo.** Modifique o programa da Listagem 10.4 de modo que um array dinâmico seja criado. Para tanto, você deve usar o operador `new`. Ao final do programa, depois que todo o processamento tenha sido terminado, o programa deve usar o operador `delete[ ]` para desalocar memória dinamicamente. Lembre-se de que seu programa deve criar um array de `double` e permitir que o usuário digite até 100 valores. Entretanto, seu programa lerá os dados digitados pelo usuário usando:

```
cin >> *(a + j);           // usando *(a+j) para armazenar em a[j]
```

E, quando acumular os valores lidos, seu programa deve usar:

```
soma += *(aPtr + j);    // usando *aPtr para acessar a[j]
```

Por fim, seu programa deve exibir a soma e a média dos valores digitados. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 10.6.

Perceba que o uso de ponteiros oferece maior flexibilidade e eficiência na manipulação de dados de um array. Vamos examinar outro exemplo que ilustra isso.

```
1. #include <iostream>
2. const int MAX = 100;
3. using namespace std;
4. // Programa para ilustrar uso de ponteiros para arrays
5.
6. int main()
7. {
8.     double *a; // declara *a como ponteiro que sera usado
9.               // para alocar e acessar array dinamico a[]
10.    double media, N;
11.    double soma = 0.0;
12.    int *n;
13.
14.    n = new int; // usa operador para alocar memoria dinamica
15.               // para variavel int n
16.    if (n == NULL) // testa se alocao dinamica falhou
17.        return 1; // sinaliza erro
18.
19.    do {
20.        cout << "\nInforme a quantidade de elementos do array de
21.            '2 a 100': ";
22.        cin >> *n;
23.        cout << endl << endl;
24.    } while (*n < 2 || *n > MAX);
25.
26.    a = new double[*n]; // cria um array dinamico usando ope-
27.        rador new
28.    if (!a) // testa se alocao de array dinamico esta ok
29.    {
30.        delete n; // caso contrario, desaloca memoria
31.        return 1;
32.    }
33.    for (int j = 0; j < *n; j++)
```

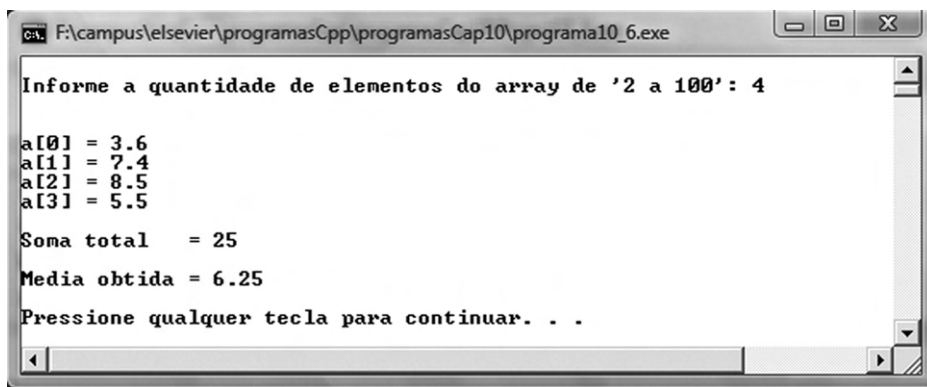
```

33.     cout << "a[" << j << "]" = ";
34.     cin >> a[j]; // ler j-esima entrada e a armazena no
35.                // elemento j do array dinamico
36. }
37.
38. N = *n;
39. for (int j = 0; j < *n; j++)
40.     soma += *(a + j); // acesso aos elementos do array dinamico
41. // usando *(a+j)
42. media = soma / N;
43. cout << "\nSoma total  = " << soma;
44. cout << "\n\nMedia obtida = " << media << endl << endl;
45. delete n; // desaloca memoria da variavel dinamica acessada
           pelo ponteiro n
46. delete [] a; // desaloca o array dinamico que é acessado
           pelo ponteiro a
47. system("PAUSE");
48. return 0;
49. }

```

**Listagem 10.6.**

O programa da Listagem 10.6 cria um array dinâmico que pode armazenar até 100 valores do tipo double. O usuário é solicitado a digitar a quantia e os valores, e depois o programa exibe a soma total e a média. Note, contudo, que esse programa faz uso dos operadores *new* e *delete[]* para fazer alocação e desalocação dinâmica, respectivamente. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.6.



**Figura 10.6** – Saída do programa da Listagem 10.6.

Observe que, na linha 8 da Listagem 10.6, é feita declaração de *\*a* como ponteiro que será usado para alocar e acessar o array dinâmico *a[]*. Já na linha 25, o operador *new* é usado para criar o array dinâmico *a[]*. Na linha 40, *\*(a+j)* é usado para acessar o conteúdo do array dinâmico, enquanto a instrução *delete[] a* é usada para desalocar o array dinâmico que é acessado pelo ponteiro *a*.

## 10.5. PONTEIROS COMO ARGUMENTOS

Nos exemplos vistos até o momento, você já teve oportunidade de explorar o uso de arrays e objetos passados como argumentos de funções, bem como os elementos do array sendo acessados por funções. No entanto, você ainda não teve oportunidade de utilizar ponteiros para manipular arrays. Cabe destacar que é comum fazer uso de ponteiros (em vez de arrays) quando se necessita que arrays sejam passados (como argumento) para funções.

Para entender como isso pode ocorrer, vamos examinar o próximo exemplo, que implementa uma função de ordenação para ordenar um conjunto de números armazenados em um array. Para tanto, considere o código da Listagem 10.7.

```
1. #include <iostream>
2. const int MAX = 10;
3. using namespace std;
4. // Programa para ilustrar uso de ponteiros como argumentos
5.
6. void bubbleSort(int *ptr, int n)
7. {
8.     void ordena(int *, int *); // prototipo da funcao
9.     for (int i = 0; i < n-1; i++)
10.        for (int j = i + 1 ; j < n; j++)
11.            ordena(ptr + i, ptr + j); // ordena conteudo do ponteiro
12. }
13.
14. void ordena(int *n1, int *n2)
15. {
16.     if (*n1 > *n2) // testea se n1 > n2
17.     {
18.         int tmp = *n1; // efetua troca de n1 por n2
19.         *n1 = *n2;
20.         *n2 = tmp;
21.     }
22. }
23.
24. void bubbleSort(int *, int);
25.
```

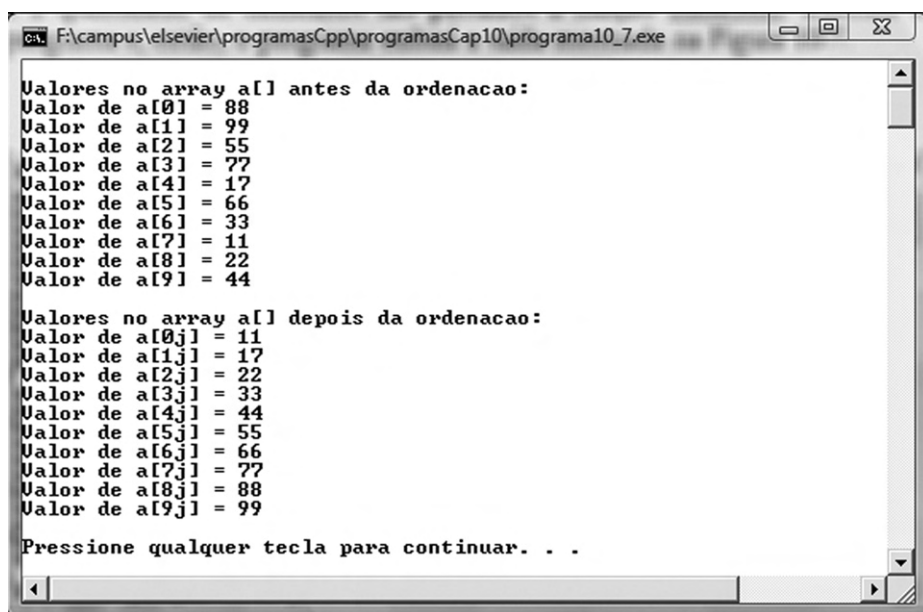
```

26. int main()
27. {
28.     int a[MAX] = {88, 99, 55, 77, 17, 66, 33, 11, 22, 44};
29.     cout << "\nValores no array a[] antes da ordenacao: ";
30.
31.     for (int i = 0; i < MAX; i++)
32.         cout << "\nValor de a[" << i << "] = " << a[i];
33.
34.     bubbleSort(a, MAX);
35.     cout << "\n\nValores no array a[] depois da ordenacao: ";
36.     for (int i = 0; i < MAX; i++)
37.         cout << "\nValor de a[" << i << "] = " << a[i];
38.     cout << endl << endl;
39.     system("PAUSE");
40.     return 0;
41. }

```

**Listagem 10.7.**

Observe que a função *ordena()*, nas linhas 14 a 22, trabalha com qualquer outra função. A diferença na implementação da Listagem 10-7 é que os argumentos da função *ordena()* são ponteiros. Além disso, o array *a[]* é inicializado na linha 28 com um conjunto de valores não ordenados. O endereço do array *a[]* e a quantidade de elementos são passados à função *bubbleSort*. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.7.



```

F:\campus\elsevier\programasCpp\programasCap10\programa10_7.exe
Valores no array a[] antes da ordenacao:
Valor de a[0] = 88
Valor de a[1] = 99
Valor de a[2] = 55
Valor de a[3] = 77
Valor de a[4] = 17
Valor de a[5] = 66
Valor de a[6] = 33
Valor de a[7] = 11
Valor de a[8] = 22
Valor de a[9] = 44

Valores no array a[] depois da ordenacao:
Valor de a[0] = 11
Valor de a[1] = 17
Valor de a[2] = 22
Valor de a[3] = 33
Valor de a[4] = 44
Valor de a[5] = 55
Valor de a[6] = 66
Valor de a[7] = 77
Valor de a[8] = 88
Valor de a[9] = 99

Pressione qualquer tecla para continuar. . .

```

**Figura 10.7** – Saída do programa da Listagem 10.7.

## 10.6. CLASSES LISTA, PILHA E FILA

### 10.6.1. Lista

Agora, vamos examinar três aplicações interessantes. A primeira delas faz uso da classe *list*, que provê implementação de uma lista ligada. Uma lista ligada é uma estrutura de dado que organiza um conjunto de elementos ligados através de ponteiros. Cada elemento da lista consiste em um elemento de dado e um ponteiro (que aponta para o próximo elemento da lista).

Para acessar dados de uma lista ligada, você deve utilizar o ponteiro para o primeiro elemento da lista (*head*). Já o último elemento da lista (*tail* ou cauda) contém o valor NULL, que serve para indicar que o final da lista foi alcançado. Para entender mais, vamos examinar o exemplo seguinte, que implementa uma lista fazendo uso da classe *list*.

O exemplo a seguir solicita que o usuário digite valores inteiros que vão sendo inseridos em uma lista *lista* até o momento em que o usuário decide encerrar e digitar 's' para sair. Nesse instante, o programa ordena os valores armazenados na lista, colocando-os em ordem ascendente, fazendo uso da função *sort()* da classe *list*, conforme linha 23.

```
1. #include <iostream>
2. #include <list>
3. using namespace std;
4. // Programa para ilustrar uma lista
5.
6. int main()
7. {
8.     list<int> lista;           // cria um objeto lista ligada a lista
9.     list<int>::iterator i;    // ponteiro especial da classe list
10.                                // para acessar elementos numa lista
11.     int entrada;
12.     i = lista.begin(); // begin() retorna um (ponteiro) iterator
13.                                // que aponta para o primeiro elemento
                                // da lista
14.     cout << "\nDigite valores inteiros ou 's' para sair: \n";
15.
16.     while (cin >> entrada)
17.     {
18.         lista.insert(i, entrada); // insere valor de entrada no
19.                                // endereço especificado pelo
                                // iterator 'i'
20.         ++i;
21.     }
22.
```



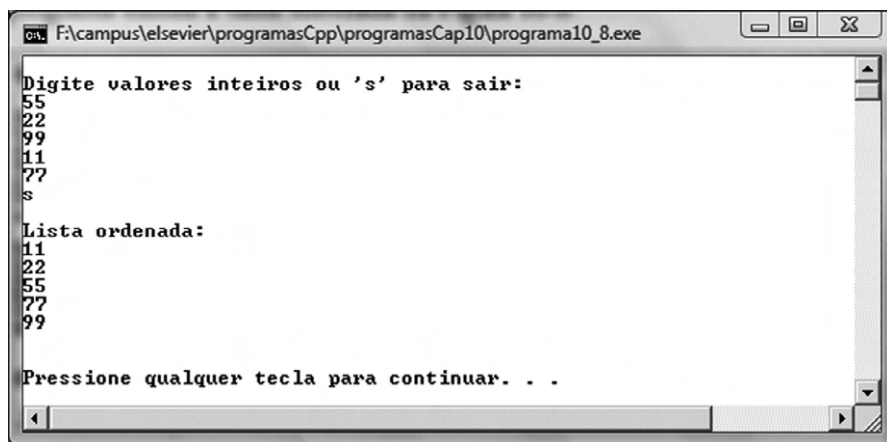
```

23. lista.sort(); // ordena elementos da lista em ordem ascen-
    dente
24. cout << "\nLista ordenada: \n";
25. for(i = lista.begin(); i != lista.end(); ++i) // end() testa
    se final da lista
26.                                     // foi alcançado
27. {
28.     cout << *i << endl;
29. }
30.     cout << endl << endl;
31.     system("PAUSE");
32.     return 0;
33. }

```

**Listagem 10.8.**

Observe que a função na linha 8 é uma lista criada da classe *list*, a qual será usada para armazenar os valores digitados pelo usuário. Os números digitados pelo usuário são tratados nas linhas 16-21, usando a função *insert()*, linha 18, da classe *list*. Os elementos da lista são colocados em ordem ascendente usando a função *sort()* na linha 23 da classe *list*. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.8.



**Figura 10.8 – Saída do programa da Listagem 10.8.**

## 10.6.2. Pilha

Outra aplicação interessante utiliza a estrutura de dados pilha. Uma pilha é uma estrutura que possui uma política de acesso LIFO (*Last-In, First-Out*), a qual define a maneira pela qual os dados são inseridos e removidos da estrutura de dados pilha.

Se precisar de uma estrutura pilha, você pode adicionar ao seu programa a classe *stack*, que provê implementação de uma pilha. Uma pilha permite inserir e remover elementos da pilha fazendo uso das funções *push()* e *pop()*. Entretanto, essas funções podem atuar apenas sobre os elementos que estão no topo da pilha.

O exemplo da Listagem 10.9 solicita que o usuário digite valores inteiros que vão sendo inseridos em uma pilha até o momento em que o usuário decide encerrar e digita 's' para sair. Nesse instante, o programa exibe o conteúdo da pilha, removendo os dados em ordem inversa na qual foram inseridos.

```
1. #include <iostream>
2. #include <stack>
3. using namespace std;
4. // Programa para ilustrar uma pilha
5.
6. int main()
7. {
8.     stack<int> pilha; // cria um objeto pilha
9.     int entrada;
10.
11.     cout << "\nDigite valores inteiros ou 's' para sair: \n";
12.     while (cin >> entrada)
13.     {
14.         pilha.push(entrada); // coloca dado no topo da pilha
15.     }
16.
17.     cout << "\nConteudo da pilha: \n";
18.     while (!pilha.empty()) // testa se pilha nao esta vazia.
19.     {                                     Retorna true
20.                                     // se pilha esta vazia e false caso
21.                                     contrario
22.         cout << pilha.top() << endl; // top() retorno o elemento
23.         do topo
24.                                     // da pilha, mas nao o remove
25.         pilha.pop();                // pop() remove o elemento do topo
26.                                     // da pilha mas nao o remove
27.     }
28.     cout << endl << endl;
29.     system("PAUSE");
30.     return 0;
31. }
```

**Listagem 10.9.**

Observe que a função na linha 8 é uma pilha criada da classe *stack*, a qual será usada para armazenar os valores digitados pelo usuário. Os números digitados pelo usuário são tratados nas linhas 12-15, usando a função *push()*, linha 14, da classe *stack*. Os elementos são removidos da pilha no laço das linhas 18-24 usando a função *pop()* da classe *stack*, conforme a linha 22. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.9.

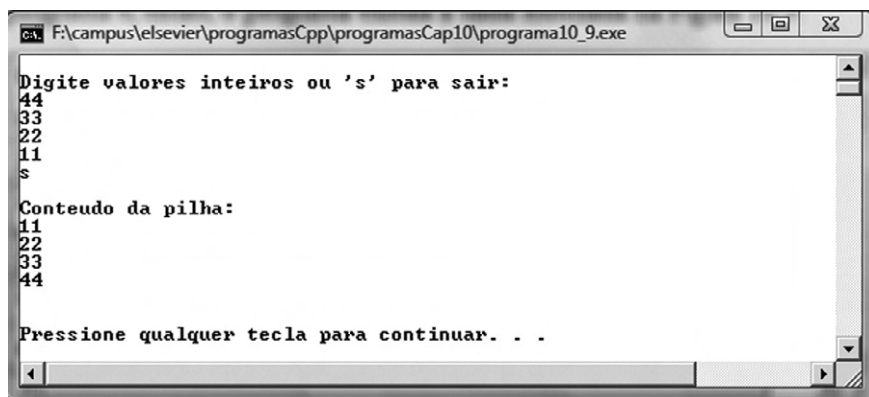


Figura 10.9 – Saída do programa da Listagem 10.9.

### 10.6.3. Fila

Outra aplicação importante é a estrutura de dados fila. Uma fila é uma estrutura que possui uma política de acesso FIFO (*First-In, First-Out*), a qual define a maneira pela qual os dados são inseridos e removidos da estrutura de dados fila.

Caso necessite de uma estrutura fila, você pode adicionar ao seu programa a classe *queue*, que provê implementação de uma fila. Uma fila permite inserir e remover elementos da pilha fazendo uso das funções *push()* e *front()*. Cabe destacar que, toda vez que você utiliza a função *push()*, o elemento é sempre inserido no final da fila, e, quando você remove um elemento da fila usando a função *front()*, esse elemento é o que está na frente da fila.

O exemplo da Listagem 10.10 ilustra o funcionamento de uma fila. O programa solicita que o usuário digite valores inteiros que vão sendo inseridos em uma fila até o momento em que decide encerrar e digita 's' para sair. Nesse instante, o programa exibe o conteúdo da fila, removendo os dados na ordem em que eles foram inseridos.

```
1. #include <iostream>
2. #include <queue>
3. using namespace std;
4. // Programa para ilustrar uma fila
5.
```

```
6. int main()
7. {
8.     queue<int> fila; // cria um objeto fila
9.     int entrada;
10.    cout << "\nDigite valores inteiros ou 's' para sair: \n";
11.    while (cin >> entrada)
12.    {
13.        fila.push(entrada); // coloca dado na fila
14.    }
15.
16.    cout << "\nConteudo da fila: \n";
17.    while (!fila.empty()) // testa se a fila esta vazia. Retorna true
18.    {                      // se fila esta vazia e false caso contrario
19.        cout << fila.front() << endl; // front() retorna o elemento
            da frente
20.                                // da fila, mas nao o remove
21.        fila.pop();           // pop() remove o elemento do topo da pilha
22.                                // mas nao o retorna
23.    }
24.    cout << endl << endl;
25.    system("PAUSE");
26.    return 0;
27. }
```

#### Listagem 10.10.

Observe que a função na linha 8 é um objeto fila criado da classe *queue*, o qual será usado para armazenar os valores digitados pelo usuário. Os números digitados pelo usuário são tratados na linha 11 usando a função *push()*, linha 13, da classe *queue*. Os elementos são removidos da fila no laço da linhas 17-23 usando a função *front()* da classe *queue*, conforme a linha 19. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 10.10.

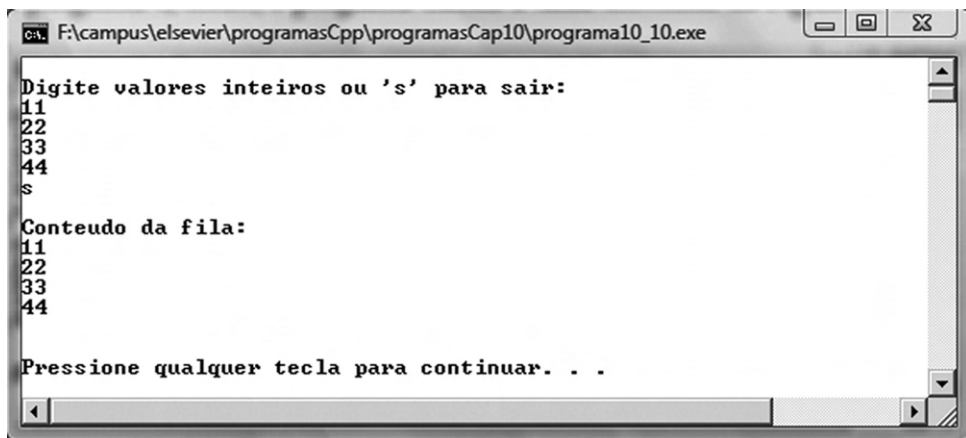


Figura 10.10 – Saída do programa da Listagem 10.10.

## RESUMO

Neste capítulo, você teve a oportunidade de estudar ponteiros e explorar os conceitos e características oferecidas por esse recurso da linguagem C++. Diversas operações fazendo uso de ponteiros foram examinadas em várias situações. Além disso, você teve a oportunidade de saber como fazer o gerenciamento dinâmico de memória, utilizando os operadores *new* e *delete[]*, bem como aprendeu a usar ponteiros como argumentos de funções que proporcionam manipulação eficiente de dados. Por fim, exemplos de como utilizar as classes *list* (lista), *stack* (pilha) e *queue* (fila) foram apresentados. Diversos exemplos foram usados para a apresentação de todo o conteúdo. O próximo capítulo mostra a importância da educação continuada.

## QUESTÕES

1. O que são ponteiros? Onde eles podem ser utilizados? Use exemplos para ilustrar sua resposta.
2. O que é armazenado em um ponteiro? Use um exemplo para ilustrar sua resposta.
3. O que é operador endereço e operador ponteiro? Como podemos usá-los? Use exemplos para ilustrar sua resposta.
4. Como realizar operações com ponteiros? Use um exemplo para ilustrar sua resposta.
5. É possível utilizar ponteiros como argumentos de funções? Em que situações eles devem ser utilizados? Use exemplos para ilustrar sua resposta.

## EXERCÍCIOS

1. Faça uma pesquisa visando responder à seguinte questão: como é possível fazer o gerenciamento da alocação dinâmica de memória? Apresente um exemplo para ilustrar sua resposta.
2. Escreva um programa que crie um array no qual você deve inserir um conjunto de números inteiros lidos do usuário. Depois, seu programa deve chamar uma função `f(int *arrayPtr, int n)` passando o ponteiro do array que contém os dados lidos e o tamanho do array. Essa função é encarregada de ordenar e exibir o conteúdo do array ordenado.
3. Escreva um programa que implemente a classe *Lista*, que deve conter um conjunto de notas de alunos. Em seguida, seu programa deve verificar se as notas obtidas pelos alunos são maiores ou iguais a 7,0. Apenas as notas maiores ou iguais a 7,0 devem ser exibidas em ordem ascendente na saída.