Capítulo 8

Sobrecarga de Operadores

The most incomprehensible thing about the world is that it is comprehensible.

Albert Einstein

OBJETIVOS

- Entender e saber onde empregar sobrecarga de operadores.
- Aprender a definir operadores sobrecarregados unário e binário.
- Conhecer e utilizar funções friend.
- Entender e explorar a conversão de dados.

No capítulo anterior, você aprendeu o que são arrays e como utilizá-los. Um array compreende uma coleção contínua de dados do mesmo tipo, diferentemente de uma estrutura que agrupa dados de tipos distintos. Neste capítulo, você estudará sobrecarga de operadores, que constitui um dos aspectos mais importantes que se tem na programação orientada a objetos. Ela permite transformar uma listagem complexa de programa em outra mais intuitiva. Você pode utilizar classes para criar novos tipos de variáveis e, fazendo uso da sobrecarga de operadores, pode criar novas definições para os operadores. Este capítulo explora esse tópico, e um conjunto de exemplos ilustrativos é usado para a apresentação do conteúdo. Nesse sentido, questões a serem exploradas são: como sobrecarregar e utilizar um operador? Como fazer conversão de dados? O que são e como utilizar funções friend? Responder a essas e outras questões é o objetivo deste capítulo e, para tanto, exemplos são usados para ilustrar situações em que é adequado o uso de sobrecarga de operadores.

8.1. INTRODUÇÃO

Anteriormente, no Capítulo 5, você teve a oportunidade de estudar funções com o mesmo nome mas funcionalidades distintas, ou seja, você usou o mesmo nome

para várias funções, onde a diferença entre elas era a quantidade de parâmetros ou os tipos dos argumentos.

A linguagem C++ provê suporte a esse tipo de sobrecarga, que é denominado sobrecarga funcional ou polimorfismo funcional. Todavia, similarmente à sobrecarga de funções, a linguagens C++ oferece suporte à sobrecarga de operadores.

8.1.1. Sobrecarga de Operadores

Sobrecarga de operadores é um dos aspectos mais importantes que se tem na programação orientada a objetos. Ela permite transformar uma listagem de programa complexa em outra mais intuitiva. A sobrecarga de operadores possibilita estender o conceito de sobrecarga para operadores, o que permite atribuir múltiplos significados para os operadores da linguagem C++.

Note que C++ permite empregar também o conceito de sobrecarga de operadores em tipos definidos pelo usuário, onde se poderia, por exemplo, somar dois objetos. Para tanto, considere que n1 e n2 são dois objetos números complexos.

Você sabe que o operador soma (+), predefinido na linguagem C++, não pode ser utilizado para somar números complexos. Em tal situação, você teria de definir uma classe *NumerosComplexos*, bem como implementar a função soma de números complexos (que deve somar as partes real e imaginária) separadamente. Caso precisasse somar dois números complexos n1 e n2 e atribuir o resultado a n3, você escreveria:

```
n3.somaComplexos(n1, n2);
```

Entretanto, se fizer uso da sobrecarga de operadores, essa instrução pode ser modificada para simplesmente:

```
n3 = n1 + n1;
```

O termo sobrecarga de operadores vem do fato de se dar aos operadores normais de C++ como +, *, + e = significados adicionais quando eles são aplicados a tipos de dados definidos pelo usuário como, por exemplo, classes definidas pelo usuário. Usando classes, você pode criar novos tipos de variáveis, e, usando operadores sobrecarregados, pode criar novas definições para esses operadores.

8.2. SOBRECARGA DE OPERADORES UNÁRIOS

Os operadores unários atuam sobre apenas um operando (ou uma variável). Exemplos de operadores unários são os operadores ++ (incremento) e –– (decremento), estudados no Capítulo 2.

Agora, suponha que você tenha uma classe chamada *Contador* (que permite criar objetos de contagem). Assim, para os objetos dessa classe serem incrementados, é preciso fazer uma chamada a uma função-membro, como mostrado a seguir:

```
c1.incrementaContador();
```

Todavia, você poderia ter uma listagem mais compreensível se usasse um operador de incremento da seguinte forma:

```
++c1;
```

Para que possa tornar isto possível, você precisa sobrecarregar o operador. Então, vamos examinar um exemplo para entender como funciona.

Praticando um Exemplo. Escreva um programa que crie uma classe *Contador*. Nessa classe, você deve especificar o construtor *Contador()*, a função *getContador()* e sobrecarregar o operador de incremento (++) utilizando a seguinte sintaxe:

void operator ++() { ++contador; }

Note que a palavra-chave *operator* é usada para sobrecarregar o operador ++ na declaração *void operator ++ ().* Essa sintaxe diz ao compilador para chamar essa função toda vez que o operador ++ for encontrado, contanto que o operando (ou a variável sobre a qual ++ atua) seja do tipo *Contador*. Seu programa deve criar dois objetos *c1* e *c2*. *c1* é incrementado uma vez e *c2* três vezes. Em seguida, os valores de *c1* e *c2* são mostrados, chamando a função *getContador()* a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.1.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar sobrecarga de operadores
4.  class Contador // especificacao de classe
5.  {
6.    private:
7.     unsigned int contador;
8.    public:
9.     Contador() { contador = 0; } // construtor
10.    int getContador() { return contador; } // retorna contador
11.    void operator ++() { ++contador; } // pre-incremento
12. };
13. int main()
14. {
15.   Contador c1, c2; // declaracao de 2 objetos contador
16.   cout << "\nContador 1 antes do incremento = " <<
      c1.getContador();
17.   cout << "\nContador 2 antes do incremento = " <<
      c2.getContador();
18.
```

```
19.   ++c1; // incrementa c1 uma vez
20.   ++c2;     // incrementa c2 tres vezes
21.   ++c2;
22.   ++c2;
23.   cout << "\n\nContador 1 depois do incremento = " <<
      c1.getContador();
24.   cout << "\nContador 2 depois do incremento = " <<
      c2.getContador();
25.   cout << endl << endl;
26.   system("PAUSE");
27.   return 0;
28. }
```

Listagem 8.1

O programa da Listagem 8.1 define a classe *Contador* e cria dois objetos *Contador* chamados de *c1* e *c2*. O objeto *c1* é incrementado uma vez e *c2* três vezes. Em seguida, os valores de *c1* e *c2* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.1.

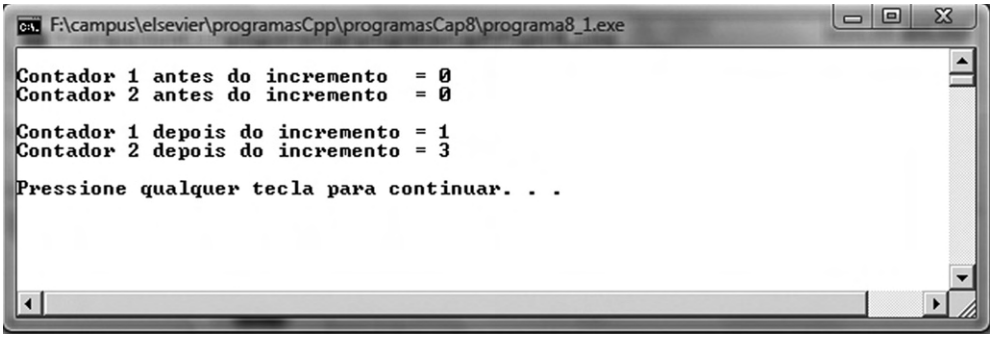

Figura 8.1 – Saída do programa da Listagem 8.1.

Observe que, na linha 11 da Listagem 8.1, a palavra-chave *operator* é usada para sobrecarregar o operador ++ na declaração *void operator ++ ()*. Essa sintaxe diz ao compilador para chamar essa função toda vez que o operador ++ for encontrado, contanto que o operando (ou a variável sobre a qual ++ atua) seja do tipo *Counter*.

Anteriormente, você aprendeu que a única forma de o compilador distinguir entre funções sobrecarregadas é analisando os tipos de dados e a quantidade de argumentos da função (consulte o Capítulo 5).

Da mesma forma, o operador sobrecarregado é diferenciado pelo tipo de dado de seus operandos. Se um operando é de um tipo predefinido como, por exemplo, um inteiro, então o compilador utiliza o operador normal. Caso contrário, se o operando

(ou variável) é de um tipo definido pelo usuário, como a classe *Contador*, o compilador usa o operador sobrecarregado definido pelo usuário (programador).

■ *É importante observar que em operator++() não há argumentos. Isso ocorre porque as funções-membros sempre podem ter acesso ao objeto para o qual elas tenham sido chamadas (então não se usam argumentos).*

Vale ressaltar que a função *operator++()* não trata todas as possíveis situações que você poderia desejar. Por exemplo, se você tentar fazer algo como:

```
c1 = ++c2;
```

o compilador não saberá como tratar essa instrução. Em uma situação como a anterior, você poderia inserir um tipo de dado de retorno em vez de usar *void*. Vejamos isso examinando o próximo exemplo.

Praticando um Exemplo. Modifique o programa anterior, que cria uma classe *Contador*. Nessa classe, você deve especificar o construtor *Contador()*, a função *getContador()* e sobrecarregar o operador de incremento (++) utilizando a seguinte sintaxe:

Contador operator ++() { ... }

A sintaxe *Contador operator ++ ()* diz ao compilador para chamar essa função toda vez que o operador ++ for encontrado, contanto que o operando (ou variável sobre a qual ++ atua) seja do tipo *Contador*. Seu programa deve criar três objetos *c1*, *c2* e *c3*. *c1* é incrementado uma vez e *c2* três vezes, e depois tem seu valor atribuído a *c3*, fazendo *c3 = ++c2;*. Em seguida, os valores de *c1* e *c2* são mostrados, chamando a função *getContador()* a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.2.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar sobrecarga de operadores
4.
5.  class Contador // especificacao de classe
6.  {
7.    private:
8.     unsigned int contador;
9.    public:
10.    Contador() { contador = 0; } // construtor
11.    int getContador() { return contador; } // retorna contador
12.    Contador operator ++()       // pre-incremento
13.    {
14.      ++contador;   // incrementa contador
15.      Contador tmp; // cria um Contador temporario
16.      tmp.contador = contador; // atribui a tmp o valor de contador
```

```
17.     return tmp; // retorna o valor
18.    }
19. };
20. int main()
21. {
22.  Contador c1, c2, c3; // declaracao de 2 objetos contador
23.  cout << "\nContador 1 antes do incremento = " <<
     c1.getContador();
24.  cout << "\nContador 2 antes do incremento = " <<
     c2.getContador();
25.  cout << "\nContador 3 antes do incremento = " <<
     c3.getContador();
26.
27.  ++c1; // c1=1 incrementa c1 uma vez
28.  ++c2;     // c2=1 incrementa c2 tres vezes
29.  ++c2;     // c2=2
30.  c3 = ++c2; // c3=3 c2=3
31.
32.  cout << "\n\nContador 1 depois do incremento = " <<
     c1.getContador();
33.  cout << "\nContador 2 depois do incremento = " <<
     c2.getContador();
34.  cout << "\nContador 3 depois do incremento = " <<
     c3.getContador();
35.  cout << endl << endl;
36.  system("PAUSE");
37.  return 0;
38. }
```

Listagem 8.2

O programa da Listagem 8.2 define a classe *Contador* e cria três objetos *Contador* chamados de *c1*, *c2* e *c3*. O objeto *c1* é incrementado uma vez e *c2* três vezes, e seu resultado é atribuído a *c3*. Em seguida, os valores de *c1*, *c2* e *c3* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.2.

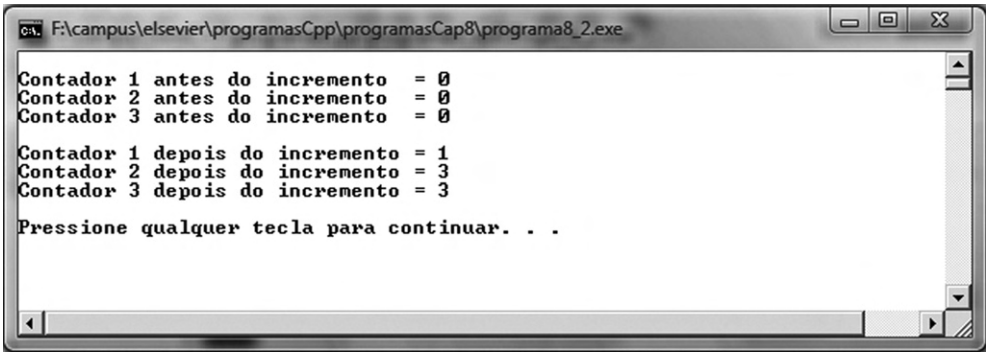

Figura 8.2 – Saída do programa da Listagem 8.2.

No exemplo, a função *operator++()* cria um novo objeto *tmp* (na linha 15) do tipo *Contador*, *contador* é incrementado e seu valor atribuído ao novo objeto. Finalmente, o objeto *tmp* é retornado (linha 17).

Embora o exemplo anterior tenha implementado o que queríamos, foi necessário criar um objeto temporário (*tmp*) com o objetivo de retornar o valor do operador ++. Mas será que é possível simplificar isso?

A resposta é sim. No programa da Listagem 8.2, você utilizou as instruções a seguir para implementar a sobrecarga de operador de pré-incremento onde incrementa uma variável e atribui seu conteúdo a outra variável, como em *c3 = ++c2;* (linha 30 da Listagem 8.2).

```
14. ++contador;
15. Contador tmp;
16. tmp.contador = contador;
17. return tmp;
```

Você pode simplificar esse código substituindo as instruções das linhas 15-17 (da Listagem 8.2) pela instrução *return Contador(contador);*. Fazendo essa modificação no código do programa anterior, você obtém o programa da Listagem 8.3.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  // Programa para ilustrar sobrecarga de operadores
5.  class Contador // especificacao de classe
6.  {
7.    private:
8.     unsigned int contador;
9.    public:
10.    Contador() { contador = 0; } // construtor qdo sem argumento
11.    Contador(int c) { contador = c; } // construtor qdo com 1
       argumento
12.    int getContador() { return contador; } // retorna contador
13.    Contador operator ++()       // pre-incremento
14.    {
15.      ++contador;   // incrementa contador
16.      return Contador(contador); // retorna o valor
17.    }
18. };
19. int main()
20. {
21.   Contador c1, c2, c3; // declaracao de 2 objetos contador
22.   cout << "\nContador 1 antes do incremento = " << c1.getContador();
```

```
23.  cout << "\nContador 2 antes do incremento = " <<
     c2.getContador();
24.  cout << "\nContador 3 antes do incremento = " <<
     c3.getContador();
25.
26.  ++c1; // c1=1 incrementa c1 duas vezes
27.  ++c1;
28.  ++c2;     // c2=1incrementa c2 tres vezes
29.  ++c2;     // c2=2
30.  c3 = ++c2; // c3=3 c2=3
31.  cout << "\n\nContador 1 depois do incremento = " <<
     c1.getContador();
32.  cout << "\nContador 2 depois do incremento = " <<
     c2.getContador();
33.  cout << "\nContador 3 depois do incremento = " <<
     c3.getContador();
34.  cout << endl << endl;
35.  system("PAUSE");
36.  return 0;
37. }
```

Listagem 8.3

O programa da Listagem 8.3 define a classe *Contador* e cria três objetos *Contador* chamados de *c1*, *c2* e *c3*. O objeto *c1* é incrementado duas vezes e *c2* três vezes, e seu resultado atribuído a *c3*. Em seguida, os valores de *c1*, *c2* e *c3* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.3.

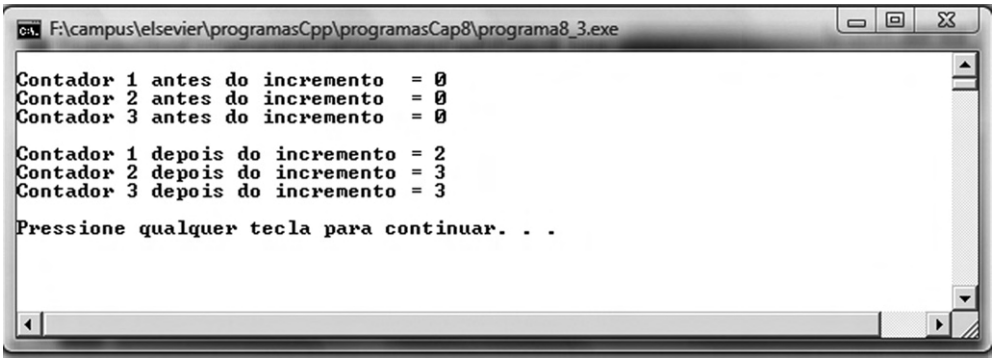

Figura 8.3 – Saída do programa da Listagem 8.3.

■ *É importante observar que precisamos de um construtor com um argumento (como mostrado na linha 11 da Listagem 8.3) para que, uma vez que um objeto (sem nome) seja inicializado para o valor de contador, ele possa ser retornado.*

Praticando um Exemplo. Modifique o programa anterior, que cria uma classe *Contador*. Essa classe sobrecarrega o operador de incremento (++) de modo a oferecer suporte ao pós-incremento, situação na qual o incremento ocorre após a variável ter sido usada, representada como *c1++*. Para tanto, você deve utilizar a sintaxe a seguir:

Contador operator ++(int) { return Counter(count++); }

Observe, nessa sintaxe, que o papel de *int* é o de informar ao compilador que uma versão de *pós-incremento* do operador ++ deve ser criada. Seu programa deve criar três objetos *c1*, *c2* e *c3*. *c1* é incrementado duas vezes e *c2* três vezes, e depois tem seu valor atribuído a *c3*, fazendo *c3 = ++c2;*. Em seguida, os valores de *c1, c2* e *c3* são mostrados, chamando a função *getContador()* a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.4.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar sobrecarga de operadores
4.
5.  class Contador // especificacao de classe
6.  {
7.    private:
8.     unsigned int contador;
9.    public:
10.    Contador() { contador = 0; } // construtor qdo sem argu-
       mento
11.    Contador(int c) { contador = c; } // construtor qdo com 1
       argumento
12.    int getContador() { return contador; } // retorna contador
13.    Contador operator ++()       // pre-incremento
14.    {
15.      return Contador(++contador); // incrementa e retorna o valor
16.    }
17.    Contador operator ++(int) // pos-incremento
18.    {
19.      return Contador(contador++); // utiliza valor atual de
         contador e
20.        // depois o incrementa, retornando valor atual
21.    }
22. };
23. int main()
24. {
25.   Contador c1, c2, c3; // declaracao de 2 objetos contador
26.   cout << "\nContador 1 antes do incremento = " << c1.getContador();
27.   cout << "\nContador 2 antes do incremento = " << c2.getContador();
```

```
28.   cout << "\nContador 3 antes do incremento = " << c3.getContador();
29.
30.   ++c1; // c1=1 incrementa c1 duas vezes
31.   ++c1;
32.   ++c2;     // c2=1incrementa c2 tres vezes
33.   ++c2;     // c2=2
34.   c3 = c2++; // c3=3 c2=3
35.
36.   cout << "\n\nContador 1 depois do incremento = " <<
      c1.getContador();
37.   cout << "\nContador 2 depois do incremento = " <<
      c2.getContador();
38.   cout << "\nContador 3 depois do pos-incremento = " <<
      c3.getContador();
39.   cout << endl << endl;
40.   system("PAUSE");
41.   return 0;
42. }
```

Listagem 8.4

O programa da Listagem 8.4 define a classe *Contador* e ilustra como sobrecarregar o operador ++ para atuar como pós-incremento de um objeto. Para tanto, o programa cria três objetos *Contador* chamados de *c1*, *c2* e *c3*. O objeto *c1* é incrementado duas vezes e *c2* três vezes, sendo a última um pós-incremento, e seu resultado é atribuído a *c3*. Em seguida, os valores de *c1*, *c2* e *c3* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.4.

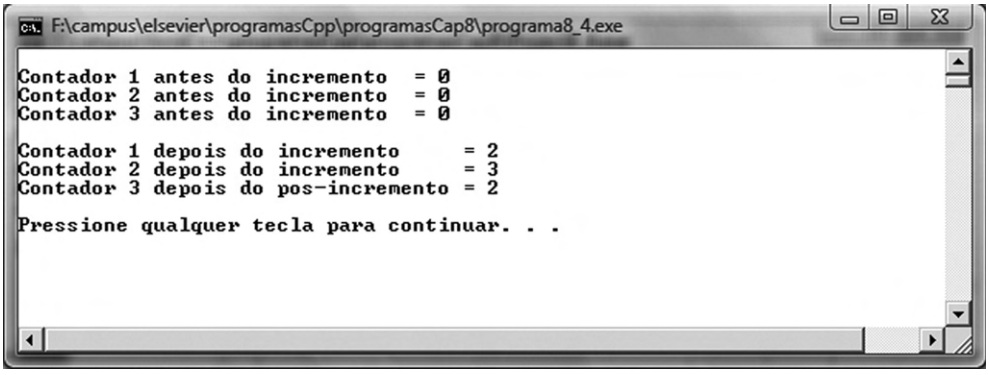

Figura 8.4 – Saída do programa da Listagem 8.4.

Note, na linha 17 da Listagem 8.4, a notação usada para pós-incremento, onde se tem *Contador operator ++ (int)*. A diferença em relação ao pré-incremento é o *int* entre parênteses.

■ *É importante observar que esse int (usado na linha 17 da Listagem 8.4) não é um argumento. Sua função é informar ao compilador que uma versão de pós-incremento do operador ++ deve ser criada. Da mesma forma que os operadores de incremento foram apresentados, um processo similar permite implementar os operadores de decremento.*

8.3. SOBRECARGA DE OPERADORES BINÁRIOS

Os operadores binários podem ser *sobrecarregados* similarmente como ocorre com os operadores unários. Para tanto, vamos examinar exemplos ilustrando como sobrecarregar operadores aritméticos e de comparação.

No Capítulo 6, você aprendeu como dois objetos Medidas podiam ser adicionados. Lá tínhamos a função *funcaoSoma(v1, v2),* conforme a seguir.

```
m3 = funcaoSoma(m1, m2); // 3 operandos
```

Note que, se você sobrecarregar o operador *soma* (+), poderá transformar a expressão anterior em: *m3 = m1 + 22*. Vamos examinar o próximo exemplo, que ilustra como obter a sobrecarga do operador + (soma).

Praticando um Exemplo. Modifique o programa da Listagem 8.5 de modo a implementar a sobrecarga do operador soma (+) e permitir que se possa somar dois objetos da classe Medida, como ilustrado a seguir:

m3 = m1 + m2;

Observe que você deve implementar a sobrecarga do operador soma (+) fazendo *Medida Medida::operator + (Medida m1)*.

Seu programa deve criar três objetos *m1*, *m2* e *m3*. *M2* deve ser inicializado com *Medida m2(100, 25.0)*. Você deve solicitar que o usuário entre com os dados dos objetos m1 e m3. Depois, o programa deve efetuar a operação *m3 = m1 + m2;* e exibir os valores dos três objetos, chamando a função *mostraMedida()* a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.5.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar sobrecarga de operador binario
4.  // que permite a soma de dois objetos
5.
6.  class Medida // Especificacao da classe Medida
7.  {
8.    private:
9.     int metro;
10.    double centimetro;
```

```
11.  public:
12.   Medida ()
13.   {
14.   metro = 0;
15.   centimetro = 0.0;
16.   }
17.  Medida(int m, double c) // define medida
18.  {
19.   metro = m;
20.   centimetro = c;
21.  }
22.  void getMedida() // obtem medida do usuaripo
23.  {
24.   cout << "\nDigite a parte de metros da medida: ";
25.   cin >> metro;
26.   cout << "Digite a parte de centimetros da medida: ";
27.   cin >> centimetro;
28.  }
29.  void mostraMedida() // exibe medida
30.  {
31.   cout << (metro + centimetro/100) << " metros \n";
32.  }
33.  Medida operator + ( Medida );
34. };
35.
36. Medida Medida::operator + (Medida m1) // retorna o valor da
    soma
37. {
38.  int m = metro + m1.metro;
39.  double c = centimetro + m1.centimetro;
40.  if(centimetro >= 100.0) // testa se centimetro >= 100
41.  {
42.   c -= 100.0; // decrementa centimetro
43.   m++;
44.  }
45.  return Medida(m, c);
46. }
47.
48. int main()
49. {
50.  Medida m1, m3; // cria dois objetos Medida
51.  Medida m2(100, 25.0);
52.  m1.getMedida(); // obtem valores para objeto m1
53.  cout << "\nValores iniciais das medidas"; // exibe as me-
     didas
```

```
54.   cout << "\nMedida 1 = "; m1.mostraMedida();
55.   cout << "\nMedida 2 = "; m2.mostraMedida();
56.   cout << "\nMedida 3 = "; m3.mostraMedida();
57.
58.   m3 = m2 + m1; // m3 = m2 + m1
59.   cout << "\nValores atualizados das medidas \n";
60.   cout << "\nMedida 1 = "; m1.mostraMedida(); // exibe valor
      total em metros
61.   cout << "\nMedida 2 = "; m2.mostraMedida();
62.   cout << "\nMedida 3 = "; m3.mostraMedida();
63.   cout << endl;
64.   system("PAUSE");
65.   return 0;
66. }
```

Listagem 8.5

O programa da Listagem 8.5 define a classe *Medida* e ilustra como sobrecarregar o operador soma (+) para somar dois objetos Medida. Para tanto, o programa cria três objetos *Medida* chamados de *m1*, *m2* e *m3*. O objeto *m3* recebe a soma dos valores dos objetos *m1*e *m2*. Em seguida, os valores de *m1*, *m2* e *m3* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.5.

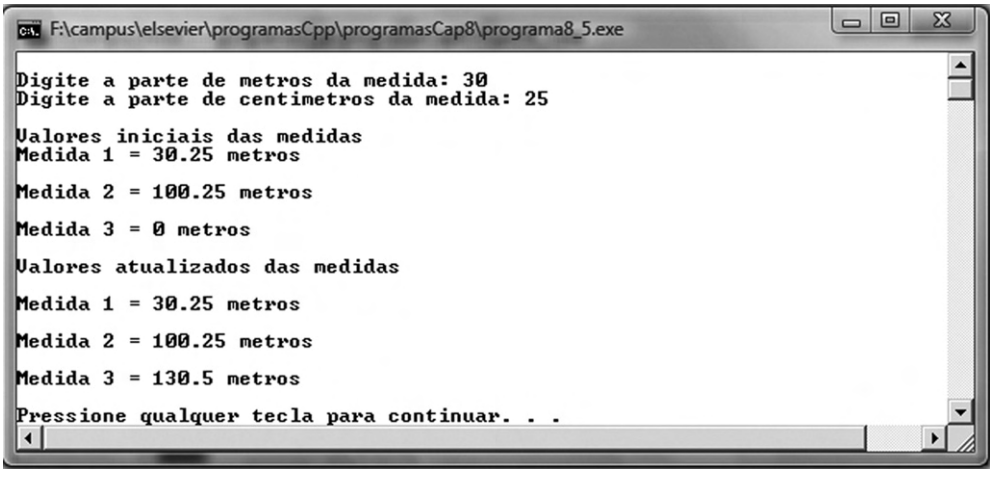

Figura 8.5 – Saída do programa da Listagem 8.5.

■ *Note que podemos dizer que um operador sobrecarregado sempre pede um argumento a menos do que o seu número de operandos, desde que um operando seja um objeto da classe da qual o operador é uma função-membro. Essa é a razão por que os operadores unários não exigem argumentos.*

Observe, na especificação da classe *Medida*, que na linha 33 da Listagem 8.5 a declaração para a função *operator+()* é:

```
Medida operator + (Medida);
```

Essa função tem um tipo de retorno *Medida* e pede um argumento do tipo *Medida*. Perceba que é importante saber o tipo de retorno e argumento utilizado pelo operador. Observe que, quando o compilador encontra na linha 58 a expressão:

```
m3 = m2 + m1;
```

ele verifica que deve usar a função-membro *operator+()* de *Medida*. Note que o argumento do lado esquerdo do operador (*m2*, no exemplo anterior) é o objeto da classe da qual o operador é uma função-membro. O objeto do lado direito do operador (*m1*) deve ser fornecido como argumento para o operador.

Agora, vamos examinar um exemplo de concatenação de strings. Normalmente, não poderíamos fazer:

```
string3 = string1 + string1;
```

onde *string1*, *string2* e *string3* são variáveis do tipo string (ou seja, arrays do tipo *char*). Nesse caso, você teria algo do tipo:

```
"Antonio" + "Brasil" como sendo "AntonioBrasil".
```

Entretanto, se você definir a classe *String* (como visto no Capítulo 7), poderá sobrecarregar o operador + para que ele realize essa concatenação. Para entender melhor, vamos examinar o próximo exemplo.

Praticando um Exemplo. Escreva um programa de modo a implementar a sobrecarga do operador soma (+) e permitir que possa concatenar dois objetos da classe String, como ilustrado a seguir:

s3 = s1 + s2;

Observe que você deve implementar a sobrecarga do operador soma (+) fazendo *String operator + (String argString),* que mostra o operador +, o qual pega um argumento do tipo *String* e retorna um objeto do mesmo tipo. Quanto ao processo de concatenação em *operator +(),* isso envolve a criação de um objeto temporário do tipo *String*, copiar a string (objeto de *String*) na string temporária (*tmp*), fazer a concatenação usando a função *strcat()* e retornar a string temporária (*tmp*) resultante. Seu programa deve criar três objetos *s1*, *s2* e *s3*. *s1* e *s2* devem ser inicializados com strings. Depois, o programa deve efetuar a operação *s3 = s1 + s2;* concatenando *s1* e *s2* e exibindo *s3* com a chamada à função *mostraString()* a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.6.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar concatenacao de strings
4.
5.  const int MAX = 100; // tamanho dos objetos String
6.
7.  class String // Especificacao da classe String
8.  {
9.    private:
10.    char string[MAX];
11.   public:
12.    String() // construtor sem argumento
13.    {
14.      strcpy(string, "");
15.    }
16.    String( char s[] ) // construtor com 1 argumento
17.    {
18.      strcpy(string, s);
19.    }
20.    void mostraString() // exibe string
21.    {
22.      cout << string;
23.    }
24.    String operator + (String argString) // concatena strings
25.    {
26.      String tmp;
27.      if( strlen(string) + strlen(argString.string) < MAX )
28.      {
29.        strcpy(tmp.string, string); // copia string para tmp
30.        strcat(tmp.string, argString.string); // concatena ar-
           gumento com string
31.      }
32.      else
33.      {
34.        cout << "\nOcorreu overflow de string";
35.        exit(1);
36.      }
37.    return tmp;     // retorna string tmp
38.    }
39. };
40. int main()
41. {
42.   String s1 = "Antonio Mendes "; // cria tres objetos string
43.   String s2 = "da Silva Filho";
44.   String s3;
```

```
45.   cout << "\nString s1 = "; s1.mostraString(); // exibe strings
46.   cout << "\nString s2 = "; s2.mostraString();
47.   cout << "\n\nString s3 = ";
48.
49.   s3 = s1 + s2; // concatena string s2 com s1 e atribui a s3
50.   s3.mostraString(); // exibe string s3
51.   cout << endl << endl;
52.   system("PAUSE");
53.   return 0;
54. }
```

Listagem 8.6

O programa da Listagem 8.6 define a classe *String* e ilustra como sobrecarregar o operador soma (+) para concatenar dois objetos String. Para tanto, o programa cria três objetos *String* chamados de *s1*, *s2* e *s3*. O objeto *s3* recebe a concatenação das strings *s1*e *s2*. Em seguida, os valores de *s1*, *s2* e *s3* são mostrados. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.6.

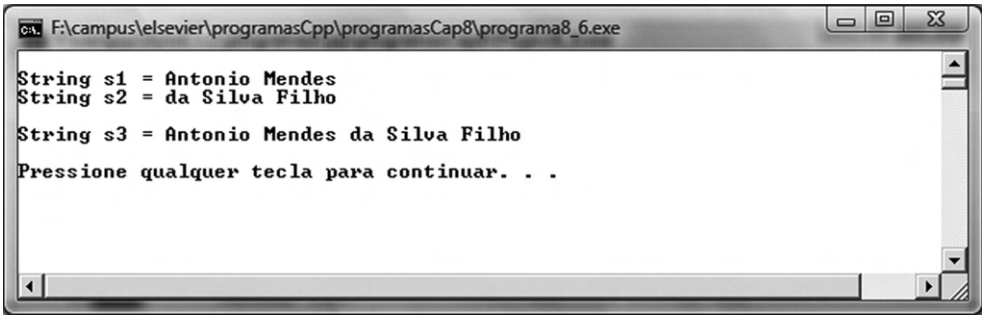

Figura 8.6 – Saída do programa da Listagem 8.6.

■ *Note que você deve ter cautela para não exceder o tamanho das strings usadas na classe String. A fim de evitar isso na função operator +(), você deve checar se o tamanho combinado de tamanhos das strings concatenadas não excede o tamanho máximo de string (como verificado na linha 27 da Listagem 8.6).*

Observe que a sobrecarga de operador feita no programa da Listagem 8.6 que sobrecarrega o operador + para concatenação também pode ser aplicada em outras situações. Você poderia usar uma abordagem similar em um operador < ou == para fazer comparação. Para entender melhor, vamos examinar um exemplo.

Praticando um Exemplo. Escreva um programa de modo a implementar a sobrecarga do operador soma (==) e permitir que comparar dois objetos (*s3* e *s1*) da classe String, como ilustrado a seguir:

if (s3 == s1) { ...};

Observe que você deve implementar a sobrecarga do operador soma (==) fazendo *meuBoolean operator == (String argString),* onde *meuBoolean* é definido como um tipo de dado enumerado:

enum meuBoolean { falso, verdadeiro };

Nesse programa, duas strings devem ser comparadas, e o valor *verdadeiro* é retornado se elas são as mesmas. Caso contrário, *falso* é retornado. O processo de comparação em *operator ==()* requer o uso da função *strcmp()* que retornará 0 se ambas forem iguais. Seu programa deve criar três objetos *s1*, *s2* e *s3*. *s1* e *s2* devem ser inicializados com as strings "sim" e "nao", respectivamente. Depois, o programa deve solicitar que o usuário digite uma palavra ("sim" e "nao"), efetuar a comparação entre *s3* com *s1* e *s2*, exibindo uma resposta se a palavra digitada for igual ("sim" e "nao") ou mensagem de alerta, caso contrário. Além disso, a função *mostraString()* deve ser chamada a partir de main(). Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.7.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar concatenacao de strings
4.
5.  const int MAX = 100; // tamanho dos objetos String
6.  enum meuBoolean { falso, verdadeiro };
7.
8.  class String // Especificacao da classe String definida pelo
    usuario
9.  {
10.   private:
11.    char string[MAX];
12.   public:
13.    String() // construtor sem argumento
14.    {
15.      strcpy(string, "");
16.   }
17.   String( char s[] ) // construtor com 1 argumento
18.    {
19.      strcpy(string, s);
20.    }
21.    void mostraString() // exibe string
22.    {
```

```
23.     cout << string;
24.    }
25.    void getString() // ler string
26.    {
27.     cin.get(string, MAX);
28.    }
29.    meuBoolean operator == (String argString) //testa se strings
       sao iguais
30.    {
31.     return ( strcmp(string, argString.string)==0 )? verda-
        deiro: falso;
32.    }
33. };
34. int main()
35. {
36.  String s1 = "sim"; // cria tres objetos string
37.  String s2 = "nao";
38.  String s3;
39.
40.  cout << "\nResponda 'sim' ou 'nao': ";
41.  s3.getString(); // ler string do usuario
42.
43.  if(s3==s1) // checa se 'sim'
44.   cout << "\nVoce digitou sim\n\n";
45.  else if(s3==s2) // checa se 'nao'
46.   cout << "\nVoce digitou nao\n\n";
47.  else
48.   cout << "\nResposta diferente das esperadas ('sim' ou
       'nao')!\n\n";
49.  system("PAUSE");
50.  return 0;
51. }
```

Listagem 8.7

O programa da Listagem 8.7 define a classe *String* e ilustra como sobrecarregar o operador soma (==) para comparar dois objetos String. Para tanto, o programa cria três objetos *String* chamados de *s1*, *s2* e *s3*. O objeto *s3* é comparado com as strings *s1*e *s2*. Em seguida, o programa mostra o resultado da comparação. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.7.

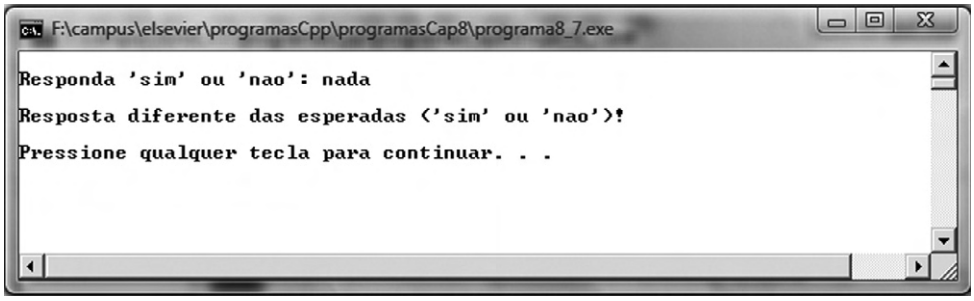

Figura 8.7 – Saída do programa da Listagem 8.7.

■ *Note que a função strcmp() realiza a comparação entre duas strings, como ocorre na linha 31 da Listagem 8.7:*

(strcmp(string, argString.string)==0) ? verdadeiro : falso;

e o resultado é comparado com 0 . Se ambas forem iguais, a função strcmp retorna 0. Caso contrário, ela retorna um número positivo se a primeira string for maior do que a segunda ou um negativo se a primeira string for menor.

8.4. SOBRECARGA DE OPERADORES ASSOCIADOS A FUNÇÕES

8.4.1. Sobrecarga do Operador []

Agora, que você já estudou sobrecarga de operadores binários, é o momento de examinar outro tipo de sobrecarga de operador. Vejamos como implementar o operador *indice [ ]*. Esse operador é normalmente usado quando você precisa ter acesso aos elementos de um array.

Ele também pode ser sobrecarregado. Por exemplo, poderíamos desejar fazer um acesso seguro ao array de modo que ele automaticamente verifica o intervalo (ou tamanho) do array. Para entender mais, vamos examinar um exemplo.

Praticando um Exemplo. Escreva um programa que crie um array (seguro), uma vez que ele verifica os índices antes de qualquer acesso ao array. Para tanto, você deve implementar a sobrecarga do operador [] (índice) de modo a ter uma única função para ler e inserir dados. Nesse caso, o retorno da função é feito por referência. Isso permite colocar a função do lado esquerdo do sinal de igual (=), e o valor do lado direito deve receber a variável retornada pela função. Observe que você deve implementar a sobrecarga do operador soma ([]), fazendo:

int & *operator [](int n)*

na classe *ArraySeguro*. Você deve criar um objeto *array1* e realizar a inserção de valores do lado esquerdo de =, fazendo *array1[i] = i*i;* para todos os elementos que têm em *ar-ray1*. Além disso, você deve fazer a inserção dos elementos do lado direito fazendo *int x = array1[i];* cujos valores serão exibidos em seguida. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.8.

```
1.  #include <iostream>
2.  #include <iomanip>
3.  using namespace std;
4.  // Programa para ilustrar sobrecarga de operadores
5.  const int MAX = 10; // tamanho dos objetos Array
6.
7.  class ArraySeguro // Especificacao da classe Array
8.  {
9.    private:
10.    int array[MAX];
11.   public:
12.    int& operator [](int n) // retorno por referencia
13.    {
14.      if( n < 0 || n >= MAX )
15.      {
16.        cout << "\nIndice excede o valor maximo!";
17.        exit(1);
18.      }
19.    return array[n];
20.    }
21. };
22. int main()
23. {
24.   ArraySeguro array1;
25.
26.   for(int i = 0; i < MAX; i++)
27.    array1[i] = i*i; // insercao de elementos do lado esquerdo
       de '='
28.
29.   cout << "Elemento Valor quadrado\n";
30.   for(int j = 0; j < MAX; j++)
31.   {
32.    int x = array1[j]; // insercao de elementos do lado direito
       de '='
33.    cout << j << setw(11) << x << endl; // exibe elementos
34.   }
35.   cout << endl << endl;
36.   system("PAUSE");
37.   return 0;
38. }
```

Listagem 8.8

O programa da Listagem 8.8 define a classe *ArraySeguro* e ilustra como sobrecarregar o operador índice ([ ]) para permitir ler e inserir dados em um objeto

ArraySeguro. Para tanto, o programa cria um objeto *ArraySeguro* chamado de *array1*. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.8.

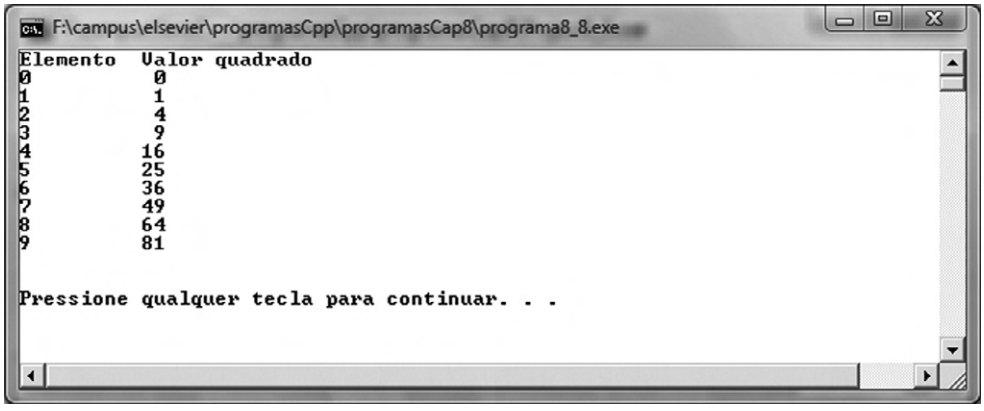

Figura 8.8 – Saída do programa da Listagem 8.8.

Observe que, no programa da Listagem 8.8, a instrução da linha 27

```
array1[i] = i*i;
```

faz com que o valor *i*i* seja colocado em *array1[i]* (ou seja, o valor de retorno da função), inserindo o valor no lado esquerdo do '='.

8.5. CONVERSÃO DE DADOS

Até o momento, você tem visto vários exemplos de sobrecarga de operadores. Entretanto, atenção especial deve ser dada ao operador '='. Esse operador possui características importantes. Sabemos que o operador '=' (que serve para fazer a atribuição de um valor para variável) é usado em situações como:

```
y = x2;
```

onde *y* e *x2* são variáveis do tipo inteiro. Também vimos que o sinal '=' pode ser usado com valores de tipos de dados definidos pelo usuário como, por exemplo, objetos destacados no exemplo a seguir:

```
string3 = string1 + string2;
```

onde os três objetos pertencem à classe *String*.

No entanto, quando o valor de um objeto é atribuído a outro, você em geral tem que os objetos são do mesmo tipo e os valores dos dados-membros da classe são simplesmente copiados no novo objeto. Nesse caso, o compilador não precisa de qualquer instrução especial para realizar a operação.

Todavia, o que acontece quando as variáveis são de tipos diferentes?

Em tal situação, você pode ter a conversão entre tipos de dados básicos (predefinidos) ou tipos definidos pelo usuário. Na conversão entre os tipos de dados básicos, você pode ter a conversão implícita (na qual o compilador usa rotinas embutidas) ou explícita (quando se faz o uso do operador *cast*).

Contudo, quando queremos fazer a conversão entre tipos dados básico e definidos pelo usuário, precisamos escrever as rotinas para informar ao compilador como ele deve fazer essa conversão. Para entender melhor como isso acontece, vamos examinar um exemplo.

Praticando um Exemplo. Modifique o programa da Listagem 8.5 de modo que ele faça a conversão de um tipo de dado básico (*double*), cujo valor está em *metros*, para um tipo definido pelo usuário (*Medida*), cujo valor está em *pés*. Note que o procedimento para a criação da classe Medida é similar ao que foi feito na Listagem 8.5 e você precisa apenas adicionar a definição do operador para implementar a conversão, fazendo:

operator double ()

na classe *Medida*. Você deve criar três objetos *m1*, *m2* e *m3*, em que *m2* é inicializado com *Medida m2(100.25)* e *m1* é inicializado com valor digitado pelo usuário. Os valores *m1*e *m2* são somados, e o resultado atribuído a *m3*. Finalmente, o valor de *m3* é convertido para pés. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.9.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar conversao de dados
4.
5.  const float METRO_PARA_PES = 3.2808; //taxa conversao metros
    p/ pés
6.
7.  class Medida // Especificacao da classe Medida
8.  {
9.    private:
10.    int metro;
11.    double centimetro;
12.   public:
13.    Medida ()
14.    {
15.      metro = 0;
16.      centimetro = 0.0;
17.    }
18.    Medida(int m, double c) // define medida
19.    {
```

```
20.     metro = m;
21.     centimetro = c;
22.    }
23.    Medida( double m ) // construtor com argumento
24.    {
25.     double doubleMetro = m;
26.     metro = int(doubleMetro); // parte inteira
27.     centimetro = 100*(doubleMetro – metro); // parte fracio-
        naria
28.    }
29.    void getMedida() // obtem medida do usuario
30.    {
31.     cout << "\nDigite a parte de metros da medida: ";
32.     cin >> metro;
33.     cout << "Digite a parte de centimetros da medida: ";
34.     cin >> centimetro;
35.    }
36.    void mostraMedida() // exibe medida
37.    {
38.     cout << (metro + centimetro/100) << " metros \n";
39.    }
40.    operator double () // funcao de conversao
41.    {
42.     double parteMetro = centimetro/100; // converte para metro
43.     parteMetro += double(metro); // adiciona parte de metros
44.     return parteMetro*METRO_PARA_PES; // converte para pes
45.    }
46.
47.   Medida Medida::operator + (Medida m1) // retorna o valor da
      soma
48.   {
49.    int m = metro + m1.metro;
50.    double c = centimetro + m1.centimetro;
51.    if(centimetro >= 100.0) // testa se centimetro >= 100
52.    {
53.     c -= 100.0; // decrementa centimetro
54.     m++;
55.    }
56.    return Medida(m, c);
57.   }
58. };
59. int main()
60. {
61.   Medida m1, m3; // cria dois objetos Medida
62.   Medida m2(100.25);
63.   m1.getMedida(); // obtem valores para objeto m1
64.   cout << "\nValores iniciais das medidas"; // exibe as medidas
65.   cout << "\nMedida 1 = "; m1.mostraMedida();
```

```
66.   cout << "\nMedida 2 = "; m2.mostraMedida();
67.   cout << "\nMedida 3 = "; m3.mostraMedida();
68.
69.   m3 = m2 + m1; // m3 = m2 + m1
70.   cout << "\nValores atualizados das medidas \n";
71.   cout << "\nMedida 1 = "; m1.mostraMedida(); //exibe valor
      total em metros
72.   cout << "\nMedida 2 = "; m2.mostraMedida();
73.   cout << "\nMedida 3 = "; m3.mostraMedida();
74.   cout << endl;
75.
76.   double pes = double(m1); // chamada a funcao de conversao
77.   cout << "\nValor medida m1 convertido para pes\n";
78.   cout << "\nMedida m1 = " << pes << " pes \n\n";
79.   system("PAUSE");
80.   Return 0;
81. }
```

Listagem 8.9

O programa da Listagem 8.9 define a classe *Medida* e ilustra a conversão de um tipo de dado básico para um tipo definido pelo usuário. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.9.

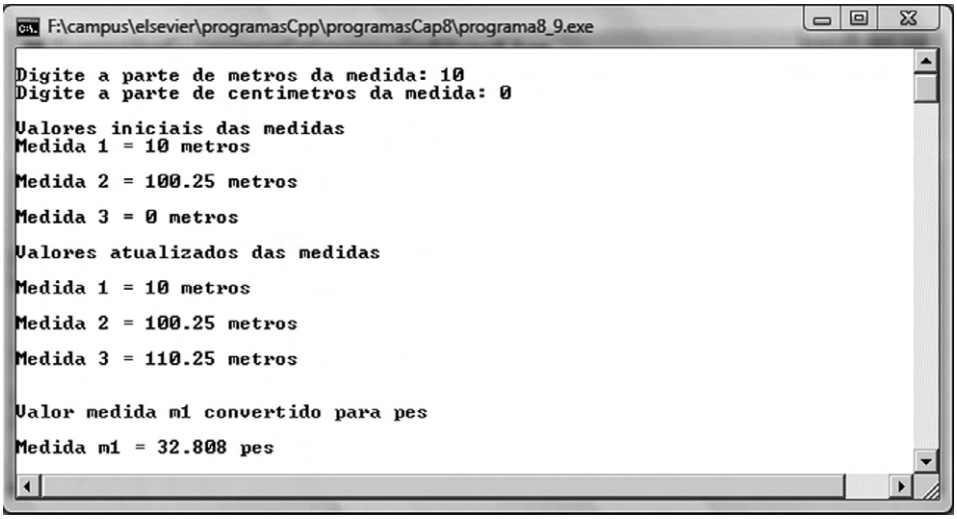

Figura 8.9 – Saída do programa da Listagem 8.9.

Perceba que, para fazer a conversão de *metro* para *Medida* (de metros para metros e centímetros) na linha 62, foi utilizado o construtor com um argumento (definido nas linhas 23-28). Ele é chamado quando um objeto do tipo *Medida* é criado com um argumento. A função assume que esse argumento representa *metros*. Assim, ela

converte o argumento para metros e centímetros, e atribui os valores resultantes para o objeto. Desse modo, a conversão de metros para *Medida* é executada junto com a criação do objeto na instrução (na linha 62):

```
Medida m2(100.25);
```

Por outro lado, a conversão de um tipo definido pelo usuário para um tipo básico é feita através da sobrecarga do operador *cast*, criando-se uma função de conversão. Isso ocorre na chamada à função na linha 76 com a instrução:

```
double pes = double(m1);
```

A implementação de *operator double()* é feita nas linhas 40-45. Esse operador pega o valor do objeto *Medida m1*, converte-o para *double* (metros) e, por fim, o converte para pés, retornando esse valor.

8.6. FUNÇÕES FRIEND

A linguagem C++ possibilita que funções-membros tenham acesso aos dados membros de uma classe (da qual essas funções também fazem parte). Adicionalmente, C++ permite ainda que funções específicas, chamadas de funções *friend*, tenham acesso privilegiado aos dados privados de uma classe.

A declaração de uma função friend é feita na classe usando a palavra-chave friend. A sintaxe para uma declaração de função friend é:

```
class NomeClass
{
  public:
  NomeClasse();
  ...
  friend tipoRetorno funcaoFriend(listaParametros);
}
```

Praticando um Exemplo. O programa da Listagem 8.5 implementa a classe Medida. Modifique esse programa adicionando a ele uma função friend quadrado(Medida) que calcula o quadrado do valor de um objeto Medida. Para tanto, você precisa adicionar na especificação da classe Medida a declaração da função friend, fazendo:

friend double quadrado(Medida);

Você deve ainda implementar essa função convertendo o valor de centímetros para metros e somando a parte de metros antes de calcular o quadrado do objeto Medida. Neste exemplo, você pode criar três objetos m1, m2 e m3, em que m2 é inicializado com Medida m2(12, 50) e m1 é inicializado com valor digitado pelo usuário. Os valores m1 e m2 são somados e o resultado atribuído a m3. Finalmente, o programa deve calcular o quadrado do valor de m3. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 8.10.

Perceba que o objetivo de usar uma função friend em um programa ocorre quando se precisa assegurar um nível de acesso a um grupo de funções de determinada classe. Nesse caso, pode-se declarar funções friend que têm acesso a dados privados de uma classe. Para entender mais, vamos examinar o próximo exemplo.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar uso de funcoes friend
4.
5.  class Medida // Especificacao da classe Medida
6.  {
7.    private:
8.     int metro;
9.     double centimetro;
10.   public:
11.    Medida ()
12.    {
13.      metro = 0;
14.      centimetro = 0.0;
15.    }
16.    Medida(int m, double c) // define medida
17.    {
18.      metro = m;
19.      centimetro = c;
20.    }
21.
22.    void getMedida() // obtem medida do usuario
23.    {
24.      cout << "\nDigite a parte de metros da medida: ";
25.      cin >> metro;
26.      cout << "Digite a parte de centimetros da medida: ";
27.      cin >> centimetro;
28.    }
29.    void mostraMedida() // exibe medida
30.    {
31.      cout << (metro + centimetro/100) << " metros \n";
32.    }
33.    Medida operator + ( Medida );
34.    friend double quadrado(Medida); // funcao friend
35. };
36.   Medida Medida::operator + (Medida m1) // retorna o valor da
      soma
37.   {
```

```
38.   int m = metro + m1.metro;
39.   double c = centimetro + m1.centimetro;
40.   if(centimetro >= 100.0) // testa se centimetro >= 100
41.   {
42.    c -= 100.0; // decrementa centimetro
43.    m++;
44.   }
45.   return Medida(m, c);
46. }
47. double quadrado(Medida m)
48. {
49.   double parteMetro = m.metro + m.centimetro/100; //converte
      para metro
50.   return parteMetro*parteMetro; // converte para pes
51. }
52. int main()
53. {
54.   Medida m1, m3; // cria dois objetos Medida
55.   Medida m2(12, 5.0);
56.   double mQuadrado;
57.
58.   m1.getMedida();
59.   m3 = m2 + m1; // m3 = m2 + m1
60.   mQuadrado = quadrado(m3);
61.
62.   cout << "\nValores atualizados das medidas \n";
63.   cout << "\nMedida 1 = "; m1.mostraMedida(); // exibe valor
      total metros
64.   cout << "\nMedida 2 = "; m2.mostraMedida();
65.   cout << "\nMedida 3 = "; m3.mostraMedida();
66.   cout << "\nQuadrado da medida m3 = " << mQuadrado;
67.   cout << endl << endl;
68.   system("PAUSE");
69.   return 0;
70. }
```

Listagem 8.10

O programa da Listagem 8.10 define a classe *Medida* e ilustra o uso da função friend *quadrado(Medida)* que tem acesso aos dados privados da classe Medida. Essa função calcula e retorna o quadrado do valor do objeto *Medida*. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.10.

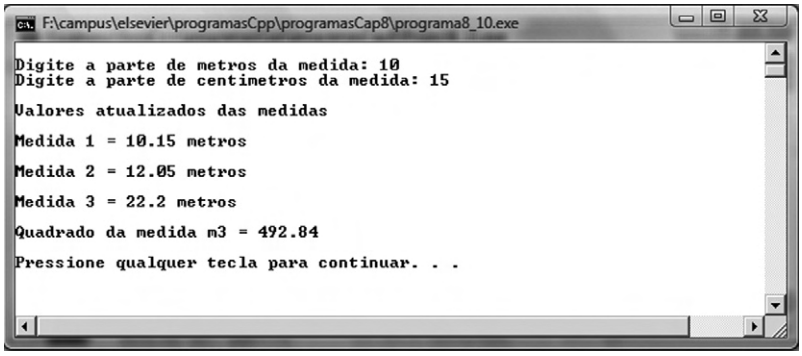

Figura 8.10 – Saída do programa da Listagem 8.10.

8.7. APLICAÇÕES E RESTRIÇÕES DE USO

8.7.1. Conversão entre Objetos de Classes Diferentes

Os métodos vistos anteriormente se aplicam a tipos de dados definidos pelo usuário. Ou seja, você pode utilizar um construtor de um argumento ou uma função de conversão. A diferença depende de você querer colocar a rotina de conversão no especificador da classe do objeto-fonte ou destino. Assim, para entender melhor, vamos examinar um exemplo.

Considere a situação na qual você tenha uma função de conversão localizada na classe do objeto-fonte. Quando a rotina de conversão está na classe-fonte, ela é geralmente implementada como uma função de conversão.

As duas classes usadas no programa da Listagem 8.11 são *Retangular* e *Polar*. A classe *Retangular* é caracterizada por ter seus objetos representando pontos em um plano bidimensional. Entretanto, ela faz uso de um sistema de coordenadas cartesianas, em que a localização de cada ponto é dada por coordenadas x e y.

■ *Observe que a função friend é um tipo especial de função que pode ter acesso a dados privados ou protegidos de uma classe. Esse tipo de função permite implementar operações mais flexíveis do que aquelas oferecidas pela funções-membros de uma classe.*

```
1.  #include <iostream>
2.  #include <math.h> // para sin() e cos()
3.  using namespace std;
4.  // Programa para ilustrar conversao de dados
5.
```

```
6.  class Retangular // Especificacao da classe Medida
7.  {
8.    private:
9.     double coordX; // coordenada x
10.    double coordY; // coordenada y
11.   public:
12.    Retangular() // construtor sem argumento
13.    {
14.      coordX = 0.0;
15.      coordY = 0.0;
16.    }
17.    Retangular(double x, double y) // construtor com 2 argu-
       mentos
18.    {
19.      coordX = x;
20.      coordY = y;
21.    }
22.    void mostraCoord() // exibe coordenadas
23.    {
24.      cout << "(" << coordX << ", " << coordY << ")";
25.    }
26. };
27. class Polar
28. {
29.   private:
30.    double raio;
31.    double angulo;
32.   public:
33.    Polar() // construtor sem argumento
34.    {
35.      raio = 0.0;
36.      angulo = 0.0;
37.    }
38.    Polar(double r, double a) // construtor com 2 argumentos
39.    {
40.      raio = r;
41.      angulo = a;
42.    }
43.    void mostraCoord()
44.    {
45.      cout << "(" << raio << ", " << angulo << ")";
46.    }
47.    operator Retangular() // funcao de conversao polar -> re-
       tangular
48.    {
```

```
49.     double x = raio * cos(angulo);
50.     double y = raio * sin(angulo);
51.     return Retangular(x, y);
52.   }
53. };
54. int main()
55. {
56.  Retangular r;
57.  Polar p(1.0, 0.785398);
58.  r = p; // conversao polar -> retangular
59.  cout << "\nCoordenada polar = "; p.mostraCoord();
60.  cout << "\nCoordenada retangular = "; r.mostraCoord();
61.  cout << endl << endl;
62.  system("PAUSE");
63.  return 0;
64. }
```

Listagem 8.11

Na parte *main()* do programa, definimos um objeto *r* da classe *Retangular* (não inicializado). Também definimos um objeto *p* que é inicializado com raio de 1,0 e ângulo de 0,785398. Em seguida, fazemos a atribuição do valor do objeto *p* para *r*:

```
r = p;
```

Desde que esses objetos são de classes diferentes, a atribuição envolve a conversão fazendo uso da rotina de conversão. Essa função transforma o objeto(*p*) da classe a qual ela é membro (*Polar*) em um objeto da classe *Retangular* e retorna esse objeto para *main(),* atribuindo-o a *r*.

O programa da Listagem 8.11 define as classes *Retangular* e *Polar*, as quais são usadas para ilustrar a conversão de um tipo de dado definido pelo usuário para outro, ou seja, de um objeto para outro. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 8.11.

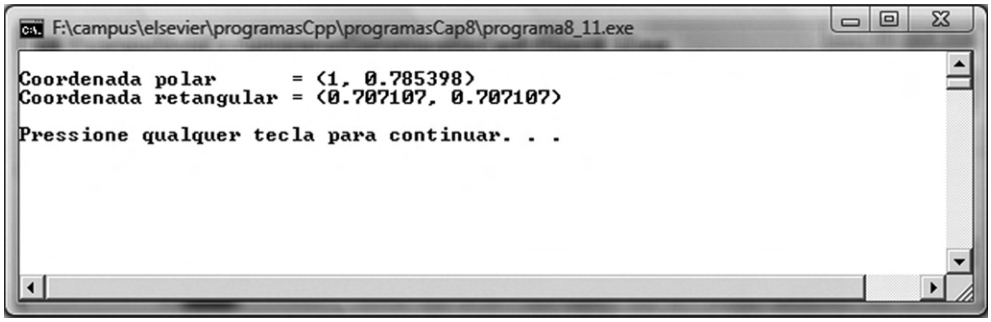

Figura 8.11 – Saída do programa da Listagem 8.11.

■ *Uma questão importante é quando usar um construtor de argumento na classe destino ou uma função de conversão na classe-fonte. Essa escolha é feita pelo programador. Por exemplo, se você compra uma biblioteca de classes, talvez não tenha acesso ao seu código-fonte. Se usar um objeto dessa classe como fonte, precisará utilizar um construtor de um argumento. Se o objeto da biblioteca de classes é o destino, você deve usar a função de conversão na fonte.*

A Tabela 8.1 mostra as conversões de tipos de dados.

A Tabela 8.1

Conversão	Rotina no Destino	Rotina na Fonte
básico para básico	funções de conversão embutidas (predefinidas)	
básico para classe	construtor	não disponível
classe para básico	não disponível	função de conversão
classe para classe	construtor	função de conversão

RESUMO

Neste capítulo, você estudou sobrecarga de operadores. Você teve a oportunidade de aprender como fazer a sobrecarga de operadores unários e binários, além de fazer funções na sobrecarga. Foi feita uma discussão sobre a diferença de sobrecarga de operadores e sobrecarga funcional (estudada no Capítulo 5). Diversos exemplos foram apresentados, mostrando como empregar a sobrecarga de operadores e a conversões de dados. Você ainda teve oportunidade de aprender que, quando precisa assegurar um nível de acesso a um grupo de funções, pode declará-las como funções friend que têm acesso a dados privados de uma classe. Diversos exemplos foram usados para apresentação do conteúdo. No próximo capítulo, você estudará e explorará o uso de herança e como essa propriedade pode ser utilizada na programação orientada a objetos.

QUESTÕES

1. Qual a diferença entre operadores unários e binários? Use exemplos para ilustrar sua resposta.

2. O que são funções friend? Em que situações se pode utilizá-las? Use exemplos para ilustrar sua resposta.

3. É possível fazer a conversão de objetos de classes distintas? Use um exemplo para ilustrar sua resposta.

4. É possível ter uma autoatribuição? Em caso afirmativo, use um exemplo para ilustrar como fazer.

EXERCÍCIOS

1. Faça uma pesquisa visando responder à seguinte questão: em que situações é adequado usar sobrecarga de operadores? Apresente um exemplo para ilustrar sua resposta.

2. Escreva um programa que faça a sobrecarga do operador +=.

3. Escreva um programa que faça a sobrecarga do operador + para que possa fazer a soma de números complexos.

4. Modifique o programa da Listagem 8.10 de modo a implementar uma função friend que determine a área de uma terreno. Para tanto, você deve especificar: friend double area(Medida).