

Capítulo 5

Funções

Do not worry about your difficulties in Mathematics.

I can assure you mine are still greater.

Albert Einstein

OBJETIVOS

- Entender e utilizar funções.
- Compreender os mecanismos de passagem de parâmetros.
- Saber como utilizar argumentos de referência.
- Entender e utilizar funções sobrecarregadas e inline.

O Capítulo 4 apresentou estruturas que permitem criar novos tipos e introduziu novos conceitos da linguagem C++. Neste capítulo, você estudará funções, que servem para agrupar várias instruções em uma unidade (a função). Essa unidade pode ser chamada de outros trechos do programa. A principal razão para utilizar funções é auxiliar na organização conceitual de um programa. Dividir um programa em funções é um dos principais princípios da programação estruturada. Entretanto, a programação orientada a objetos oferece outra forma de organizar os programas, como discutido no Capítulo 6. Outra razão para usar funções é o fato de elas reduzirem o tamanho do programa. Qualquer sequência de instruções que apareça em um programa mais de uma vez é candidata para se tornar uma função. Porém, o código da função é armazenado em apenas um local de memória, mesmo que ela seja executada muitas vezes. Este capítulo apresenta funções e ilustra seu uso com diversos exemplos. Nesse sentido, questões a serem exploradas são: como definir e usar uma função? Qual o mecanismo de passagem de parâmetros na chamada de uma função e de retorno de dados? Responder a essas questões é o objetivo deste capítulo e, para tanto, exemplos são usados para ilustrar situações em que é adequado o uso de funções.

5.1. INTRODUÇÃO

Um dos principais princípios para o desenvolvimento de sistemas, e mais especificamente de software, compreende a decomposição estrutural. Nesse sentido, a função é entidade utilizada para organizar o software em unidades (ou funções). Utilizar função para organizar o software é a base da programação estruturada. Função também é usada na programação orientada a objetos, embora ofereça outra forma de organizar o software.

5.1.1. Função

Uma função agrupa muitas instruções em uma unidade (ou entidade) e lhe atribui um nome. Cabe destacar que essa unidade pode ser chamada de outras partes do programa. Além disso, qualquer sequência de instruções que apareça em um programa mais de uma vez é candidata para se tornar uma função. Fazendo isso, você consegue também reduzir o tamanho do programa.

5.1.2. Sintaxe de Função

Embora a função seja executada muitas vezes, o código da função é armazenado em apenas um local de memória. O objetivo principal de uma função é ajudar a organizar o código. Dessa forma, você deve declarar uma função, como ilustrado no fragmento de código a seguir:

```
#include <iostream>;
tipoRetorno nomeFuncao (arg1, arg2,...) {
...; // corpo da função
}
int main()
{...; // corpo da funcao principal main() }
onde:
```

- `tipoRetorno` é o tipo de dado retornado pela função;
- `nomeFuncao` é o nome atribuído à função;
- `arg1, arg2,...` são os argumentos (ou parâmetros) passados à função quando ela é chamada; cada argumento deve ser precedido pelo seu respectivo tipo;
- `corpo da funcao` compreende às instruções da função.

Note que a declaração da função precede o seu uso. Para entender mais, vamos explorar um exemplo.

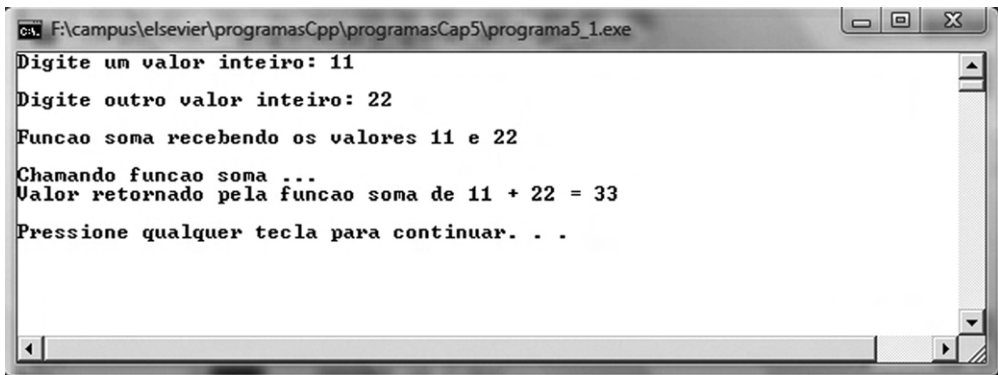
Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar dois valores inteiros e, em seguida, o programa principal chama uma função, denominada soma, passando os parâmetros que o usuário digitou. A partir desse ponto, o controle do programa é passado à função soma que deve efetuar a soma e retornar o resultado calculado. Para elaborar esse programa, você deve criar uma função soma e chamá-la a partir da função main(), passando os dados digitados pelo usuário. Procure seguir a sintaxe apresentada anteriormente para função. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.1.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar o uso de funcao
4. int soma(int x, int y) {
5.     cout << "\nFuncao soma recebendo os valores " << x << " e
        << y << endl;
6.     return (x + y);
7. }
8. int main()
9. {
10.     int a, b, c;
11.     cout << "Digite um valor inteiro: ";
12.     cin >> a;
13.     cout << "\nDigite outro valor inteiro: ";
14.     cin >> b;
15.     c = soma(a, b);
16.     cout << "\nChamando funcao soma... ";
17.     cout << "\nValor retornado pela funcao soma de " << a << "
        << b << " = " << c << endl << endl;
18.     system("PAUSE");
19.     Return 0;
20. }
```

Listagem 5.1

O programa da Listagem 5.1 ilustra o uso de uma simples função soma que faz a adição de dois números inteiros fornecidos pelo usuário. Execute o programa e digite, por exemplo, os valores 11 e 22; depois, o programa exibirá a saída mostrada na Figura 5.1.



```
cmd: F:\campus\elsevier\programasCpp\programasCap5\programa5_1.exe
Digite um valor inteiro: 11
Digite outro valor inteiro: 22
Funcao soma recebendo os valores 11 e 22
Chamando funcao soma ...
Valor retornado pela funcao soma de 11 + 22 = 33
Pressione qualquer tecla para continuar. . .
```

Figura 5.1 – Saída do programa da Listagem 5.1.

Observe que o programa consiste em duas funções: *main()* e *soma()*. Outros programas usam apenas a função *main()*. Que outros componentes são necessários para adicionar uma função ao programa? São três: a declaração da função, a(s) chamada(s) à função e a definição da função.

Da mesma forma que não podemos usar uma variável sem a declarar para informar ao compilador, também não podemos usar uma função sem declará-la antes. Portanto, você deve declarar a função no início do programa como ocorre no programa, nas linhas 4-7, da Listagem 5.1.

Declaração de Função. Vale ressaltar que a declaração simplesmente diz ao compilador que a função aparecerá em algum ponto (mais à frente) do programa. A palavra-chave *int* na linha 4 do programa da Listagem 5.1 especifica que a função possui valor do tipo *int* a ser retornado, e os parâmetros entre parênteses indicam que a função requer argumentos que devem ser recebidos. A declaração de uma função é também chamada de **protótipo**, desde que ele oferece um modelo para a função.

A função *soma()* é chamada uma vez no programa anterior. A sintaxe da chamada de uma função é similar à sua declaração, exceto que não se usa o tipo de dado do valor de retorno da função. Chamar uma função faz a função ser executada, e o controle do programa é transferido para a função (chamada), as instruções na definição da função são executadas e depois o controle retorna para a instrução seguinte à chamada de função.

A definição da função está no código da função *soma* (linhas 4-7). A definição consiste no declarador da função (linha 4) e no corpo da função (linhas 5-7).

■ É importante observar que a chamada da função deve concordar com a declaração, ou seja, elas devem possuir o mesmo nome, ter os mesmos tipos de argumentos, os argumentos devem estar na mesma ordem, além de terem o mesmo tipo de retorno. Note, ainda, que a declaração da função não termina com ponto-e-vírgula (linha 7).

As funções têm o objetivo principal de organizar o código de modo a prover maior modularidade na implementação de um software. Nesse sentido, há mecanismos que permitem a passagem de parâmetros quando uma função é chamada, assunto apresentado na próxima seção.

5.2. PASSAGEM DE ARGUMENTOS EM FUNÇÕES

5.2.1. Passagem de Argumentos

Em C++, geralmente, a passagem de argumentos é feita por valor. Um argumento é um fragmento de dado (p. ex., um valor inteiro) passado de um programa para a função. Note que argumentos permitem a uma função operar com diferentes valores ou fazer diferentes coisas, dependendo dos requisitos do programa que chama essa função. Para entender mais, vamos explorar um exemplo.

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um caractere e um valor inteiro. Esses dois valores (do tipo char e int, respectivamente) são usados pela função `exibeLinha(c, n)` na chamada à função. Essa função deve exibir na tela uma linha de `n` caracteres `c` (informado pelo usuário). Note que essa função deve ser chamada para montar uma tabela de notas de alunos, como ilustrado na Figura 5.2. Portanto, a função será chamada três vezes. Para elaborar esse programa, você deve criar uma função `exibeLinha(c, n)` e chamá-la a partir da função `main()`, passando os dados digitados pelo usuário. Procure seguir a sintaxe utilizada para função no exemplo anterior. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.2.

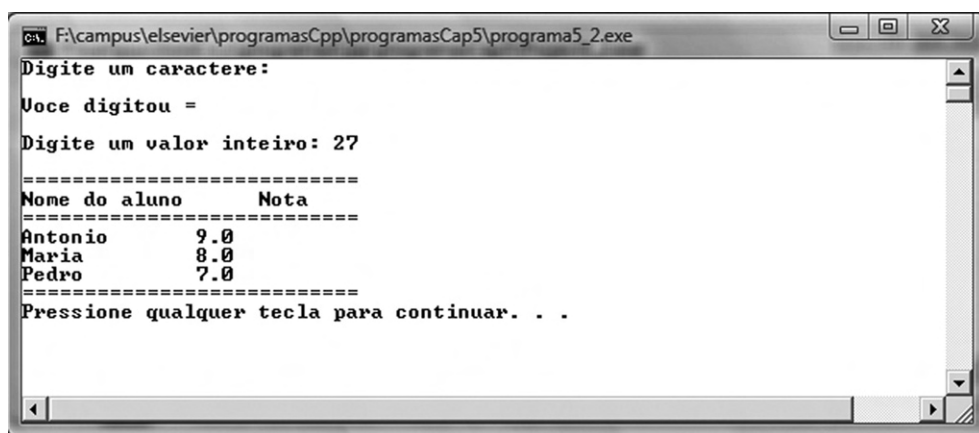
```
=====
Nome do aluno      Nota
=====
Antonio            9.0
Maria              8.0
Pedro              7.0
=====
```

Figura 5.2 – Formatação de dados desejada na saída do programa.

```
1. #include <iostream>
2. #include <conio.h> // para getch()
3. #include <iomanip> // para setw()
4. using namespace std;
5. // Programa para ilustrar o uso de funcao
6. void exibeLinha(char c, int n) // definicao da funcao
7. {
8.     cout << "\n";
9.     for(int i=0; i<n; i++)
10.         cout << c;
11.     cout << endl;
12. }
13. int main()
14. {
15.     void exibeLinha(char c, int n); // prototipo da funcao
16.     int n;
17.     char c = 'x';
18.     cout << "Digite um caractere: ";
19.     c = getch();
20.     cout << "\n\nVoce digitou " << c << endl;
21.     cout << "\nDigite um valor inteiro: ";
22.     cin >> n;
23.     exibeLinha(c, n); // chamada da funcao
24.     cout << "Nome do aluno " << setw(10) << "Nota ";
25.     exibeLinha(c, n); // chamada da funcao
26.     cout << "Antonio " << setw(15) << "9.0 " << endl;
27.     cout << "Maria " << setw(15) << "8.0 " << endl;
28.     cout << "Pedro " << setw(15) << "7.0 ";
29.     exibeLinha(c, n); // chamada da funcao
30.     system("PAUSE");
31.     return 0;
32. }
```

Listagem 5.2

O programa da Listagem 5.2 ilustra a passagem de argumentos na chamada à função `exibeLinha(c, n)`. Execute o programa e digite como entrada, por exemplo, o caractere `=` e o inteiro `27`. Então, o programa exibirá a saída mostrada na Figura 5.3.



```

c:\ F:\campus\elsevier\programasCpp\programasCap5\programa5_2.exe
Digite um caractere:
Voce digitou =
Digite um valor inteiro: 27
=====
Nome do aluno      Nota
=====
Antonio           9.0
Maria              8.0
Pedro              7.0
=====
Pressione qualquer tecla para continuar. . .
  
```

Figura 5.3 – Saída do programa da Listagem 5.2.

No programa da Listagem 5.2, foi feito uso de argumentos para passar o caractere = a ser impresso e o número (n) vezes que esse caractere deve ser repetido, imprimindo-o na tela. A função utilizada é `exibeLinha(c, n)`.

Os itens entre parênteses são os tipos de dados que serão enviados para `exibeLinha(c, n)`, ou seja, *char* e *int*. Na chamada à função, valores específicos obtidos do usuário nas linhas 20 e 23 são inseridos no lugar apropriado entre parênteses. A instrução `exibeLinha('=' , 27)` faz a função `exibeLinha(c, n)` imprimir uma linha de 27 caracteres '='. O primeiro argumento deve ser do tipo *char*, e o segundo argumento do tipo *int*.

■ É importante observar que os tipos de dados usados como argumentos devem concordar com aqueles especificados na declaração e definição da função.

Vale também ressaltar que o programa chamador é encarregado de fornecer os argumentos para a função chamada. As variáveis usadas na função para receber os valores de argumento são chamadas de parâmetros ou argumentos.

Passar argumentos dessa forma, em que uma função cria cópias dos argumentos passados a ela, é chamado de *passagem por valor*. Depois, veremos como ocorre a passagem por referência.

Usar funções é essencial na solução de problemas, principalmente quando se precisa quebrar um problema grande em partes menores (mais simples) e resolver as partes para depois obter a solução do todo. Em tais situações, o uso de funções recursivas é apropriado.

Funções Recursivas. Funções recursivas são funções que obtêm um resultado através de várias chamadas à própria função. Entretanto, as chamadas recursivas devem ser limitadas para evitar o uso excessivo de memória. Nesse sentido, deve-se estar atento para que a função recursiva verifique a condição de término de uma recursão.

Um exemplo simples de função recursiva é a função fatorial. Assim, o fatorial de n é expresso como:

$$n! = 1 * 2 * 3 * \dots * (n-2) * (n-1) * n$$

Agora, para entender e aplicar isso, nada melhor do que explorar um exemplo.

Praticando um Exemplo. Escreva um programa em C++ que calcule o número de permutações e o número de combinações. Para tanto, você sabe que o número de combinações é dado por:

$$C_{m,n} = m! / ((m - n)! * n!)$$

Enquanto o número de permutações é dado por:

$$P_{m,n} = m! / (m - n)!$$

O programa deve solicitar que o usuário digite valores inteiros m e n , sendo os valores de m na faixa de 5 a 50 e os valores de n na faixa de 2 a 10. Após o usuário entrar com esses valores, as funções *combinação* e *permutação* devem ser chamadas para calcular e exibir os valores obtidos. Para elaborar esse programa, você deve criar uma função *combinação*(m , n) e *permutação*(m , n) e chamá-la a partir da função *main*(), passando os dados digitados pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.3.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a criacao e uso de uma funcao re-
   cursiva
4. const int MIN = 2;
5. const int MAX = 50;
6.
7. double fatorial(int j)
8. {
9.     if (j > 1)
10.         return double(j) * fatorial(j-1);
11.     else

```



```

12.     return j;
13. }
14.
15. double permutacao (int m, int n)
16. {
17.     return fatorial(m) / fatorial(m - n);
18. }
19.
20. double combinacao (int m, int n)
21. {
22.     return permutacao(m, n) / fatorial(n);
23. }
24. int main()
25. {
26.     double fatorial(int j);
27.     double permutacao(int m, int n);
28.     double combinacao(int m, int);
29.     int m, n;
30.
31.     do {
32.         cout << "Digite um inteiro entre " << MIN << " e " << MAX
33.             << ": ";
34.         cin >> m;
35.     } while (m < MIN || m > MAX );
36.     do {
37.         cout << "Agora digite um inteiro entre " << MIN << " e "
38.             << m << ": ";
39.         cin >> n;
40.         cout << "\n";
41.     } while (n < MIN || n > m);
42.     cout << "Numero de permutacoes P[" << m << ", " << n << "]"
43.         << " << permutacao(m, n) << endl;
44.     cout << "Numero de combinacoes C[" << m << ", " << n << "]"
45.         << " << combinacao(m, n) << endl << endl;
46.     system("PAUSE");
47.     return 0;
48. }

```

Listagem 5.3

O programa da Listagem 5.3 ilustra o uso de funções recursivas e chamadas às funções `permutacao` e `combinacao`. Execute o programa e digite como entrada, por exemplo, os valores 20 para `m` e 2 para `n`. Então, o programa exibirá a saída mostrada na Figura 5.4.

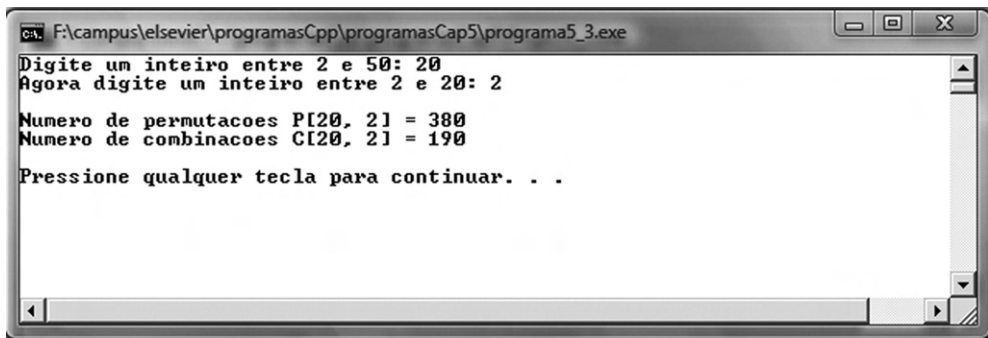


Figura 5.4 – Saída do programa da Listagem 5.3.

A função *main()* do programa recebe dois valores de *m* e *n* do tipo *int* que o usuário digita e usa esses valores na chamada às funções *permutacao(m, n)* e *combinacao(m, n)* nas linhas 40 e 41, respectivamente.

■ É importante observar que outros tipos, como por exemplo as estruturas, também podem ser usados como argumentos para funções. Em um programa, as variáveis do tipo estrutura devem ser tratadas com qualquer outra variável.

5.3. RETORNO DE VALORES DE FUNÇÕES

Quando uma função termina sua execução, ela pode retornar um valor para o programa chamador. Geralmente, esse valor retornado é uma resposta para o problema resolvido pela função. Para entender melhor como isso funciona, vamos explorar um exemplo.

Praticando um Exemplo. Escreva um programa em C++ que calcule a área de um círculo. Para tanto, você sabe que a área de um círculo é dada por $\pi * r^2$.

O programa deve solicitar que o usuário digite o valor (real) do raio e com esse valor a função *areaCirculo(r)* deve ser chamada para calcular e exibir o valor da área obtida. Para elaborar esse programa, você pode utilizar a função *acos(x)* que retorna o valor do arco cosseno de *x*, onde *x* deve estar no intervalo $[-1, 1]$. Para esse exemplo, faça *x* = -1, o que faz a função *acos(x)* retornar o valor da constante π (pi). Note que você deve criar uma função *areaCirculo(r)* e chamá-la a partir da função *main()*, passando o dado digitado pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.4.

```

1. #include <iostream>
2. #include <math.h> // necessário para funcao acos(x)
3. using namespace std;
4. // Programa para ilustrar o retorno de uma funcao
5. const double PI = acos(-1.0); // funcao arco cosseno
6.
7. double areaCirculo(double raio)
8. {
9.     return PI * raio * raio;
10. }
11. int main()
12. {
13.     double areaCirculo(double r);
14.     double r;
15.     cout << "Digite o valor do raio do circulo: ";
16.     cin >> r;
17.     cout << "\nArea do circulo de raio " << r << ": " <<
        areaCirculo(r) << endl << endl;
18.     system("PAUSE");
19.     Return 0;
20. }

```

Listagem 5.4

O programa da Listagem 5.4 ilustra como os valores podem ser retornados em funções. Execute o programa e digite como entrada, por exemplo, o valor 5,5 para o raio. Então, o programa exibirá a saída mostrada na Figura 5.5.

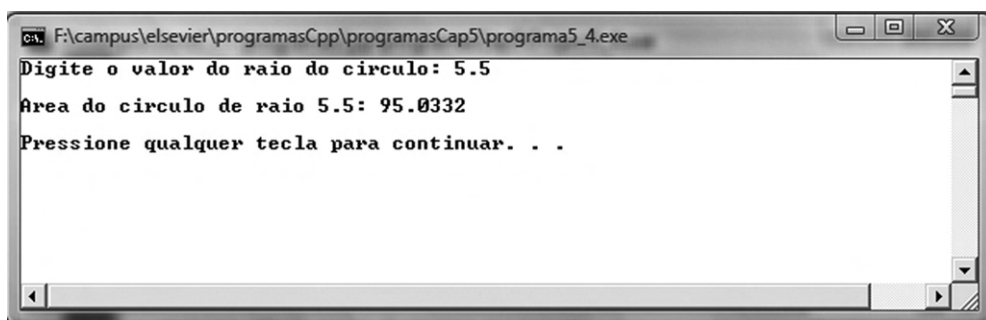


Figura 5.5 – Saída do programa da Listagem 5.4.

Quando uma função retorna um valor, o tipo de dado desse valor precisa ser especificado. Perceba que o tipo de dado *double* é colocado no exemplo antes do nome da função na declaração e na definição. O primeiro *double* especifica o tipo de dado de retorno, enquanto o *double* que está entre parênteses especifica o tipo de dado do argumento que é passado para a função (linha 7).

Observe que, enquanto muitos argumentos podem ser passados para uma função, apenas um argumento pode ser retornado dela. Isso é uma limitação quando você precisa retornar mais informação.

5.3.1. Definição de Tipo de Retorno

Você deve sempre incluir o tipo de retorno de uma função na declaração da função. Se, por acaso, a função não precisa retornar valor, então use *void* para indicar isso. Do contrário, o compilador assumirá que a função retorna um valor do tipo *int*. Contudo, não é aconselhável tirar proveito desse fato. É recomendável sempre definir o tipo de dado de retorno, mesmo que ele seja um do tipo *int* (inteiro). Adicionalmente, é recomendável usar parênteses ao redor da expressão de *return* (valor a retornar).

5.4. ARGUMENTOS DE REFERÊNCIA

Uma referência fornece um *alias* (isto é, uma espécie de pseudônimo ou outro tipo de denominação que se dá para uma variável). Você pode declarar um argumento ou parâmetro de referência colocando o caractere & após o tipo do argumento (ou parâmetro).

Além dos argumentos de referência, a linguagem C++ provê suporte às variáveis de referência. Para manipular as variáveis de referência, você deve utilizar seu *alias*. Uma forma genérica para você declarar uma variável de referência é ilustrada no exemplo a seguir.

```
tipoDado& variavelReferencia;
```

5.4.1. Variável de Referência

Cabe destacar que a variável de referência pode ser inicializada quando você a declara. Contudo, você deve assegurar que a variável de referência seja inicializada ou a ela seja atribuída outra variável. Isso é mostrado no exemplo a seguir.

```
int a = 10;
int b = 20;
int& ref_a;
ref_a = a;
int& ref_b = b;
```

O exemplo anterior ilustra como utilizar variável de referência. Nesse sentido, vale ressaltar que um dos mais importantes usos para referências é na passagem de argumentos para funções. Até aqui, você tem visto exemplos de argumentos de funções passadas por valor, em que a função chamada cria uma nova variável do mesmo

tipo do argumento e copia o valor do argumento nessa variável. Como observado nos exemplos anteriores, a função não pode ter acesso à variável original do programa chamador, mas apenas a uma cópia dela.

■ *É importante observar que passar argumento por valor é útil quando a função não precisa modificar a variável original no programa chamador. Perceba que isso impede que a função altere o valor da variável original.*

5.4.2. Passagem de Argumento por Referência

Passar argumentos por referência, contudo, utiliza um mecanismo diferente. Em vez de um valor ser passado para a função, uma referência à variável original (no programa chamador) é passada. Na realidade, o endereço da variável é passado.

A principal vantagem de passar argumento por referência é que a função pode ter acesso às variáveis no programa chamador. Um dos benefícios é que a função pode retornar mais de um valor para o programa chamador. Para entender mais, convindo-o a explorar o próximo exemplo.

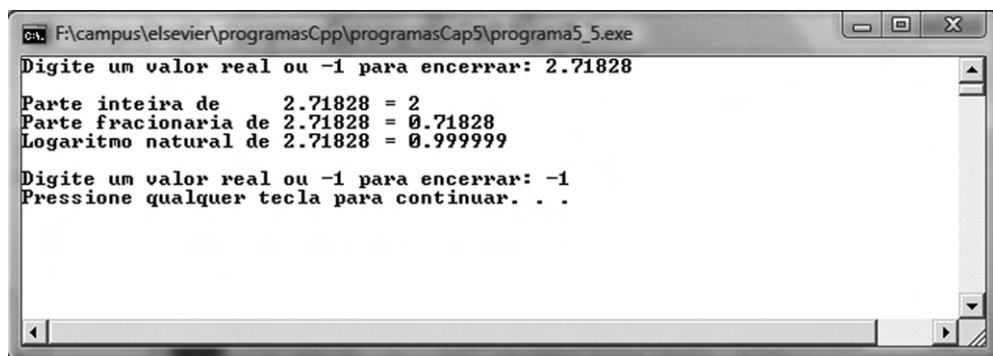
Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um valor real (que contenha partes inteira e fracionária). Adicionalmente, o programa deve obter o valor do logaritmo natural (neperiano) do valor real que você digitou. Esse programa deve implementar duas funções: `obtemIntFracao(double, double&, double&)` e `obtemLogNatural(double, double&)`, que separa as partes real e fracionária do número real e calcula o logaritmo natural do valor (real) n digitado. Para elaborar esse programa, você pode utilizar a função `log(n)`, que retorna o valor do logaritmo natural de n , onde n é um número real. Para esse exemplo, faça $n = -2.71828$ (que é o valor aproximado da constante de Euler ou base do logaritmo natural). Isso faz a função `log(n)` retornar o valor 1.0. Note que você deve criar essas funções e chamá-las a partir da função `main()`, passando o dado digitado pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.5.

```
1. #include <iostream>
2. #include <cmath> // necessario para funcao log(x)
3. using namespace std;
4. // Programa para ilustrar uso de passagem de argumento de
   referencia
5. void obtemIntFracao(double n, double& nInt, double& nFracao)
```

```
6. {
7.   nInt = double( int(n) ); // obtem para inteira
8.   nFracao = n - nInt; // obtem parte fracionaria
9. }
10.
11 void obtemLogNatural (double n, double& nLog)
12. {
13.   nLog = log(n);
14. }
15.
16 int main()
17. {
18.   void obtemIntFracao(double, double&, double&); // prototipo
    da funcao
19.   void obtemLogNatural(double, double&);
20.   double nReal, nInt, nFracao, nLog;
21.
22.   cout << "Digite um valor real ou -1 para encerrar: ";
23.   cin >> nReal;
24.   while ( nReal!= -1 ) {
25.     obtemIntFracao(nReal, nInt, nFracao); // obtem parte inteira
    e fracionaria
26.     obtemLogNatural(nReal, nLog); // obtem logaritmo natural
    de nReal
27.     cout << "\nParte inteira de " << nReal << " = " << nInt <<
    endl;
28.     cout << "Parte fracionaria de " << nReal << " = " << nFra-
    cao << endl;
29.     cout << "Logaritmo natural de " << nReal << " = " << nLog
    << endl << endl;
30.     cout << "Digite um valor real ou -1 para encerrar: ";
31.     cin >> nReal;
32.   };
33.   system("PAUSE");
34.   Return 0;
35. }
```

Listagem 5.5

O programa da Listagem 5.5 ilustra como você pode utilizar argumentos (ou parâmetros) de referência e passá-los em funções. Execute o programa e digite como entrada, por exemplo, o valor 2.71828 (isto é, a constante de Euler) e, então, o programa exibirá a saída mostrada na Figura 5.6.



```

F:\campus\elsevier\programasCpp\programasCap5\programa5_5.exe
Digite um valor real ou -1 para encerrar: 2.71828
Parte inteira de 2.71828 = 2
Parte fracionaria de 2.71828 = 0.71828
Logaritmo natural de 2.71828 = 0.999999
Digite um valor real ou -1 para encerrar: -1
Pressione qualquer tecla para continuar. . .

```

Figura 5.6 – Saída do programa da Listagem 5.5.

O programa da Listagem 5.5 pede ao usuário para digitar um número do tipo real como, por exemplo, 2.71828, e o programa separa a parte inteira da parte fracionária. Assim, o programa retornará 2 como parte inteira e 0,71828 como parte fracionária. Isso é feito pela função *obtemIntFracao()* na linha 25. A função *obtemIntFracao()* acha a parte inteira convertendo o número *n* (argumento passado) em uma variável do tipo *int* através de um *cast*. Essa conversão ocorre porque apenas a parte inteira é armazenada. A parte fracionária é simplesmente o número original menos a parte inteira. Além disso, na linha 26, o programa chama a função *obtemLogNatural()*, que utiliza a função *log(n)* para retornar o valor do logaritmo natural de *n*.

Agora, como os valores são retornados?

5.4.3. Usando Referência para Retornar Valores

return poderia ser usado se apenas um valor fosse retornado, não dois ou mais. Isso é resolvido usando argumentos de referência. Os argumentos de referência são indicados através do sinal & (E lógico) logo após o tipo de dado. O sinal & indica que, por exemplo, *nInt* é um *alias* (outra denominação ou nome) para a variável passada como argumento.

Assim, quando usamos *nInt* na função *obtemIntFracao()*, estamos nos referindo à parte inteira de *nInt* em *main()*. O sinal & pode significar *referência a*. Perceba ainda que o sinal & não é usado na chamada da função.

Note ainda que, enquanto *nInt* e *nFracao* são passadas por referência, a variável *nReal* é passada por valor. *nInt* e *nFracao* são nomes diferentes para o mesmo lugar de memória (assim como para *nLog*). Por outro lado, desde que o valor de *nReal* é

passado por valor, ele é copiado em *n*. Ele pode ser passado por valor desde que a função não precise modificar esse valor.

5.5. FUNÇÕES SOBRECARGADAS

Sobrecarga de funções é uma característica importante das linguagens orientadas a objetos e permite que se declarem múltiplas funções que tenham o mesmo nome mas diferente lista de parâmetros.

Uma função sobrecarregada executa diferentes atividades dependendo do(s) tipo(s) de dado(s) passado(s) a ela. Considere a função `calculaQuadrado()`, que deve retornar o quadrado de um valor *n* que você tenha digitado. Em tal situação, você pode digitar um número inteiro ou real e o programa deve calcular o quadrado do valor a depender do tipo que você está passando. Isso acarreta que você (programador) não necessita lembrar de dois nomes e locais dessas duas funções. Perceba que é muito mais conveniente se você tiver o mesmo nome para as duas funções, mesmo que cada uma delas tenha diferentes argumentos. Para entender mais e explorar como isso é possível, veja o exemplo seguinte.

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um valor real e em seguida calcula e retorna o quadrado do valor digitado. A seguir, o programa pede que você digite outro valor inteiro, e ele calcula e retorna o quadrado do valor inteiro digitado. A chamada a função `calculaQuadrado(n)`. Para elaborar esse programa, você deve sobrecarregar a função `calculaQuadrado(n)` que, dependendo do valor de *n*, pode calcular o quadrado de um número inteiro ou real. Note que você deve criar essas funções e chamá-las a partir da função `main()`, passando diferentes argumentos digitados pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.6.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de sobrecarga de funcao
4.
5. int calculaQuadrado(int x)
6. {
7.     return x * x; // calcula o quadrado de x (inteiro)
8. }
9.
10. double calculaQuadrado(double x)
11. {
12.     return x * x; // calcula o quadrado de x (double)
```



```

13. }
14.
15. int main()
16. {
17.     int calculaQuadrado(int n); // prototipo da funcao
18.     double calculaQuadrado(double m);
19.     int n;
20.     double m;
21.     cout << "Digite um valor inteiro: ";
22.     cin >> n;
23.     cout << "\nO quadrado de " << n << " = " << calculaQuadrado(n)
        << endl << endl;
24.     cout << "Digite um valor real: ";
25.     cin >> m;
26.     cout << "\nO quadrado de " << m << " = " << calculaQuadrado(m)
        << endl << endl;
27.     system("PAUSE");
28.     return 0;
29. }

```

Listagem 5.6

O programa da Listagem 5.6 ilustra como você pode utilizar sobrecarga de função, o que permite reutilizar o mesmo nome de função para a lista de argumentos diferentes. Execute o programa e digite como entrada, por exemplo, o valor 4 (como inteiro) e o valor 4,4 (como real); o programa exibirá a saída mostrada na Figura 5.7.

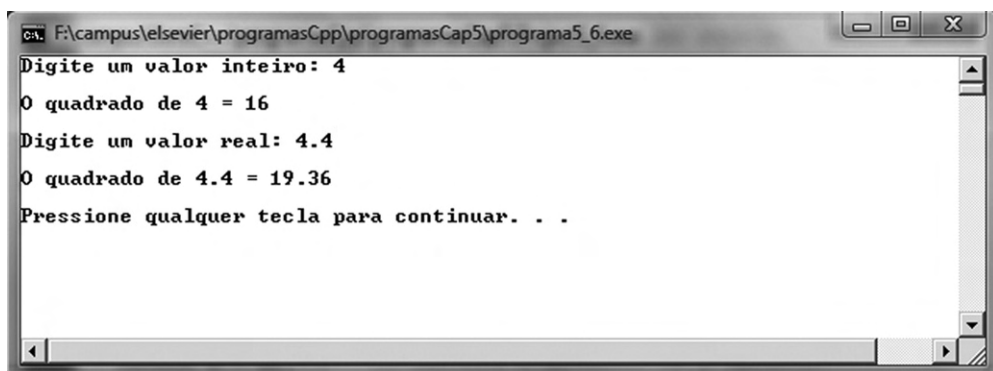


Figura 5.7 – Saída do programa da Listagem 5.6.

O programa da Listagem 5.6 possui duas funções com o mesmo nome `calculaQuadrado(n)`. Existem duas declarações, duas chamadas de função e duas definições de função. Como o compilador entende isso? Ele usa o número de argu-

mentos e tipos de dados para distinguir uma função de outra. No exemplo anterior, criamos duas funções com o mesmo nome, porém com diferentes tipos de argumentos. Adicionalmente, o compilador pode diferenciar entre funções sobrecarregadas com diferentes números de argumentos.

■ *Note que a sobrecarga de função permite declarar múltiplas funções que têm o mesmo nome, mas com lista de parâmetros diferente. A lista de parâmetro é também chamada de assinatura da função.*

5.6. FUNÇÕES INLINE

Usar funções adequadamente em um programa é essencial. As funções economizam espaço de memória porque todas as chamadas à função fazem com que o mesmo fragmento de código seja executado. Assim, o corpo da função não precisa ser duplicado na memória. Quando um compilador encontra uma chamada de função, ele geralmente passa o controle para a função (chamada). Ao final da função (chamada), o controle é passado de volta para o programa na instrução seguinte à chamada (da função).

Embora essa sequência de eventos possa economizar memória, ela consome mais tempo. Deve haver uma instrução para o salto à função, instruções para salvar registros, instruções para colocar argumentos na pilha no programa chamador e removê-los da pilha na função (se houver argumentos), instruções para restaurar os registros e uma instrução para retornar ao programa chamador. Tais instruções fazem com que o programa fique lento.

Objetivando economizar tempo de execução em funções pequenas, podemos colocar o código do corpo da função junto com o programa chamador. Ou seja, toda vez que houver uma chamada à função no código-fonte, o código da função é inserido em vez de fazer um salto para a função.

Seções longas de um código repetido deveriam ser utilizadas como funções normais. Nesse caso, a economia em termos de espaço de memória será maior quando comparada com a velocidade de execução. Perceba que, se uma função é muito pequena, as instruções necessárias para chamá-la podem consumir também espaço de memória e haverá consumo de tanto tempo quanto de espaço em memória.

5.6.1. Problema com Inserção Repetida de Função

Em tais casos, você poderia simplesmente repetir o código necessário no programa através da inserção das instruções da função sempre que necessário. O problema com essa inserção repetida do mesmo código é que você pode perder os benefícios

de organização e clareza de programa quando usa funções. O programa pode ocupar menos espaço ou ser mais rápido, porém a listagem fica maior e mais complexa.

5.6.2. Função inline

A solução para isso é usar função *inline*. Esse tipo de função é escrito como uma função normal no arquivo-fonte, mas compila no código *inline* em vez da função. O arquivo fica bem organizado e fácil de ler, desde que a função seja mostrada como uma entidade separada.

Entretanto, quando o programa é compilado, o corpo da função é inserido no programa em todos os locais onde houver uma chamada de função. As funções que são pequenas (com uma ou duas instruções, por exemplo) são candidatas a se tornarem *inline*. Para fazer uso de função inline, você deve seguir a sintaxe a seguir:

```
inline tipoRetorno nomeFuncao(listaParametros)
```

Para entender mais, veja o próximo exemplo, que ilustra o uso de função inline.

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar uma quantia em dólares que você deseja que seja convertida para reais. Seu programa deve implementar uma função inline, já que a função é pequena (isto é, contém apenas uma instrução). Após a entrada da quantia em dólares, o programa calcula e retorna a quantia convertida em reais. Chame a função inline de dolarParaReal(n). Essa função deve retornar um número do tipo double. Note que você deve criar essa função e chamá-la a partir da função main(), passando o dado (argumento) digitado pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.7.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de funcao inline
4.
5. inline double dolarParaReal (double dolar) // funcao precede
   a chamada
6. {
7.     return 1.75 * dolar;
8. }
9. int main()
10. {
11.     double dolarParaReal(double dolar); // prototipo da funcao
12.     double real, dolar;
13.     cout << "Digite uma quantia em dolar US$ ";
14.     cin >> dolar;
```

```
15. cout << "\nConvertendo dolar para reais... " << endl;  
16. cout << "\nUS$ " << dolar << " corresponde a R$ " <<  
    dolarParaReal(dolar) << endl;  
17. system("PAUSE");  
18. return 0;  
19. }
```

Listagem 5.7

O programa da Listagem 5.7 ilustra como se pode utilizar a função inline, que permite inserir função, geralmente pequena, contendo uma ou duas instruções, tornando mais eficiente o tempo de execução. Execute o programa e digite como entrada, por exemplo, a quantia de US\$ 105,40 como entrada; depois o programa exibirá a saída mostrada na Figura 5.8.

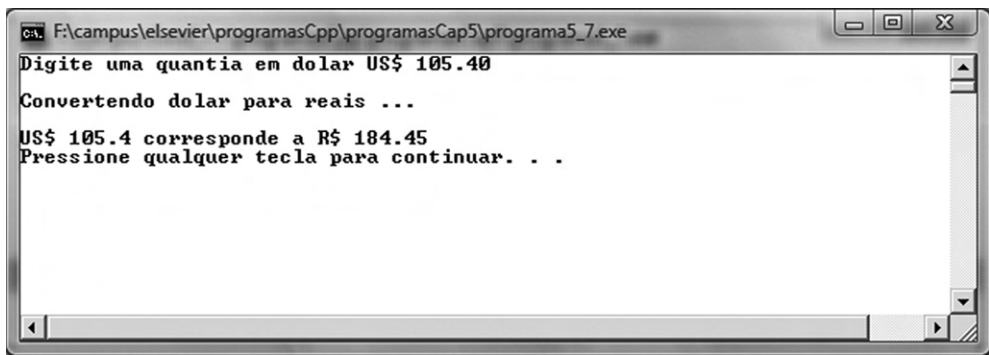


Figura 5.8 – Saída do programa da Listagem 5.7.

■ *Note que, para usar funções inline, se torna necessário apenas colocar a palavra-chave inline na definição da função. Contudo, é necessário ter tanto a declaração quanto a definição antes da chamada à função.*

Perceba que o uso de funções inline é apropriado quando se têm funções pequenas. Então, vamos ver mais um exemplo interessante.

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um número inteiro n e o programa, em seguida, retorne o quadrado e o cubo de n . Seu programa deve implementar duas funções inline. Após a entrada do valor n , o programa chama as funções inline `quadrado(n)` e `cubo(n)`. As funções inline devem retornar valores do tipo `int`. Note que você deve criar essas funções e chamá-las a partir da função `main()`, passando o dado (argumento) n digitado pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.8.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de funcao inline
4. inline int quadrado(int n) // funcao deve preceder as chamadas
5. {
6.     return n * n;
7. }
8. inline int cubo(int n) // funcao deve preceder as chamadas
9. {
10.    return n * n * n;
11. }
12. int main()
13. {
14.    int quadrado(int num); // prototipo da funcao
15.    int cubo(int num); // prototipo da funcao
16.    int num;
17.    cout << "Digite um numero inteiro: ";
18.    cin >> num;
19.    cout << "\nO quadrado de " << num << " = " << quadrado(num)
20.        << endl;
21.    cout << "\nO cubo de " << num << " = " << cubo(num) << endl
22.        << endl;
23.    system("PAUSE");
24.    return 0;
25. }

```

Listagem 5.8

O programa da Listagem 5.8 apresenta outro exemplo de uso da função inline, permitindo inserir as funções `quadrado (num)` e `cubo (num)`, chamadas nas linhas 19 e 20, respectivamente. Execute o programa e digite como entrada, por exemplo, o valor 4; depois o programa exibirá a saída mostrada na Figura 5.9.

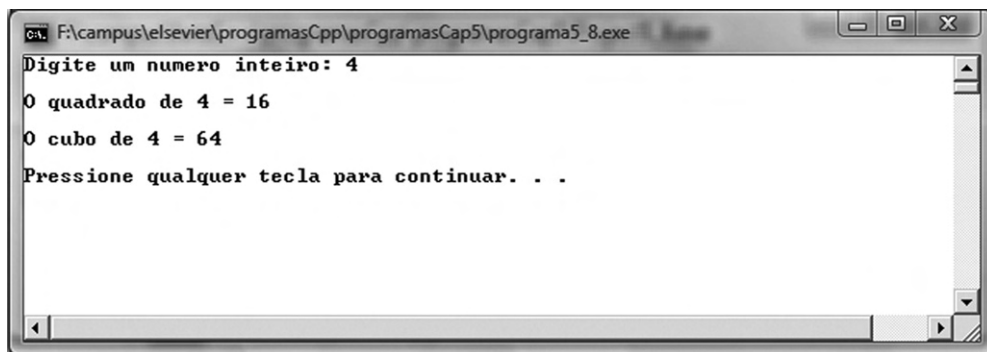


Figura 5.9 – Saída do programa da Listagem 5.8.

5.7. RETORNO POR REFERÊNCIA

5.7.1. Referência

Tipicamente, uma referência é entendida como uma *referência* que é passada para uma função. Nesse sentido, uma função que retorna uma referência funciona com um *alias* para aquela variável (que está sendo referenciada).

Em C++, uma função pode retornar uma referência. Quando uma função retorna uma referência, ela retorna uma espécie de ponteiro ou apontador implícito para o valor de retorno.

5.7.2. Função Retornando Referência

Uma pergunta que você pode estar fazendo é: quando se deve fazer uma função retornar uma referência?

Uma das principais motivações para fazer isso é quando a quantidade de informação a ser retornada é grande e, portanto, torna-se muito mais eficiente uma referência em vez de retornar uma cópia. Isso leva a aplicações interessantes. Para entender mais, vamos explorar o exemplo seguinte.

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um número inteiro n (que corresponde ao número de dias), e o programa, em seguida, retorna a quantidade de horas desses dias, multiplicando n por 24. Seu programa deve implementar a função `obterHoras(n)`. Após a entrada do valor n , o programa chama a função `obterHoras(n)`, que deve retornar uma referência do tipo `double`. Para a função `obterHoras(n)`, utilize a sintaxe `double & obterHoras(int d)`, onde d corresponde à quantidade de dias digitada pelo usuário. Note que você deve criar essa função e chamá-la a partir da função `main()`, passando o dado (argumento) dias digitado pelo usuário. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.9.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar retorno de referencia
4. double & obterHoras(int d)
5. {
6.     double h = 24.0 * d;
7.     double &horas = h;
8.     return horas;
9. }
10. int main()
11. {
```

```

12. int dias;
13. cout << "Digite a quantidade de dias: ";
14. cin >> dias;
15.
16. double horas = obterHoras(dias);
17. cout << "\nQuantidade de horas de " << dias << " dias = "
    << horas << endl << endl;
18. system("PAUSE");
19. return 0;
20. }

```

Listagem 5.9

O programa da Listagem 5.9 apresenta um exemplo de retorno de referência a uma função, tornando o mecanismo de retorno de informação mais eficiente. Execute o programa e digite como entrada, por exemplo, o valor 4; depois o programa exibirá a saída mostrada na Figura 5.10.

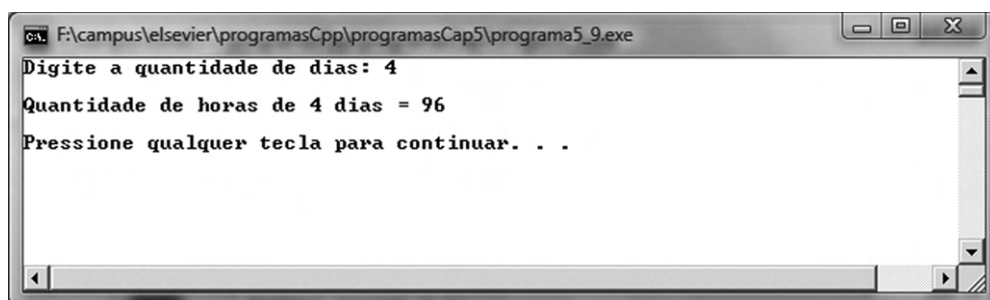


Figura 5.10 – Saída do programa da Listagem 5.9.

■ Note que, no corpo da função (linhas 5-9 da Listagem 5.9), você define o comportamento desejado para a função. Uma recomendação importante é que a função deve retornar a referência para o tipo adequado. Quando retornando um valor, você não deve preceder o nome da variável como no exemplo apresentado.

5.8. TÓPICOS COMPLEMENTARES

5.8.1. Argumentos Default

Uma função pode ser chamada sem ter todos os seus argumentos especificados. Para isso acontecer, a declaração de função deve fornecer os valores default para aqueles argumentos não especificados. Para tanto, veja o exemplo a seguir.

```
void imprimeCaracteres(char='#', int = 50);
```

Praticando um Exemplo. Escreva um programa em C++ que solicite ao usuário digitar um caractere qualquer e, em seguida, chama a função `imprimeCaracteres()`. Essa função possui dois parâmetros, sendo o primeiro do tipo `char` e o segundo do tipo `int`. No programa, você chama a função três vezes com os comandos a seguir:

```
imprimeCaracteres();  
imprimeCaracteres(ch);  
imprimeCaracteres('$', 25);
```

Em seu programa, utilize a sintaxe `void imprimeCaracteres(char='#', int = 50)` para declarar o protótipo da função. Note que você deve criar essa função e chamá-la a partir da função `main()`, passando os dados (argumentos), conforme descrito. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 5.10.

No exemplo, você especifica o protótipo da função e fornece dois argumentos ('#' e 50) *default* que podem ser usados, caso não passe qualquer dos parâmetros na chamada à função `imprimeCaracteres`. A seguir, vamos explorar um exemplo que ilustra o uso de argumentos default.

```
1. #include <iostream>  
2. #include <conio.h> // necessario para getch()  
3. using namespace std;  
4. // Programa para ilustrar uso de argumento default  
5. void imprimeCaracteres(char c, int n) // imprime sequencia  
   repetida de caracteres  
6. {  
7.     cout << "\\Imprimindo " << n << " caracteres " << c << "\\n\\n";  
8.     for(int i=0; i<n; i++)  
9.         cout << c; // imprime n caracteres 'c'  
10.    cout << endl << endl;  
11. }  
12. int main()  
13. {  
14.    void imprimeCaracteres(char='#', int = 50); // prototipo da  
       funcao com argumenos default  
15.    char ch;  
16.    cout << "Digite um caractere: ";  
17.    ch = getch();  
18.    cout << endl;  
19.    imprimeCaracteres(); // imprime 50 vezes o caractere '#'  
20.    imprimeCaracteres(ch); // imprime 50 vezes o caractere di-  
    gitado pelo usuario
```



```

21.  imprimeCaracteres('$', 25); // imprime 25 vezes o caractere
    '$'
22.  system("PAUSE");
23.  return 0;
24. }

```

Listagem 5.10

■ Note que os argumentos default são úteis quando, após escrever uma função, o programador deseja aumentar a capacidade de uma função adicionando um novo argumento. Usar argumentos default significa que as chamadas às funções existentes podem continuar a utilizar o número antigo de argumentos, ao passo que novas chamadas de funções podem usar mais.

O programa da Listagem 5.10 apresenta um exemplo de como você pode utilizar funções com argumentos default. Isso permite o uso de valores default em situações em que não sejam passados parâmetros na chamada à função. Execute o programa e digite o caractere 7 quando solicitado; depois o programa exibirá a saída mostrada na Figura 5.11.

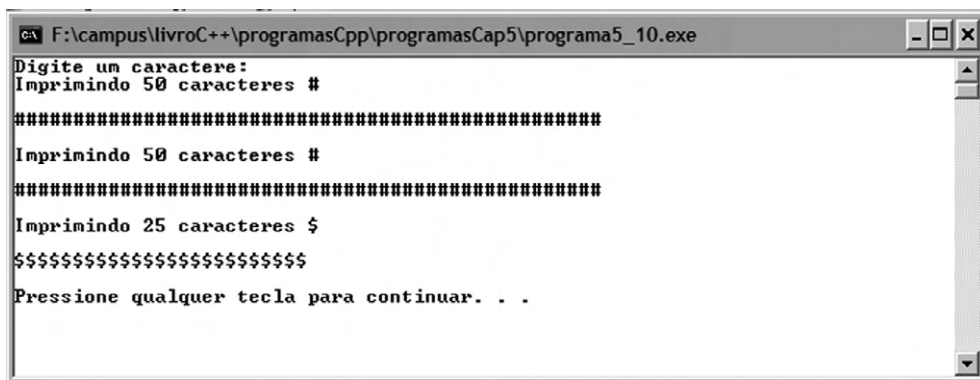


Figura 5.11 – Saída do programa da Listagem 5.10.

No programa da Listagem 5.10, a função `imprimeCaracteres()` tem dois argumentos e ela é chamada três vezes nas linhas 20-22. Quando a função é chamada e nenhum argumento é passado a ela (como na linha 20), ela usa os valores default. Tais valores default são especificados no protótipo para `imprimeCaracteres()`, como ocorre na linha 14.

Observe que, se um argumento está ausente quando uma função é chamada, é assumido que ele é o último (ou os últimos) na lista de argumentos, como ocorre na linha 21.

5.8.2. Categorias de Variáveis e Armazenamento (ou Memória)

A categoria de memória de uma variável determina quais partes do programa podem ter acesso à variável e por quanto tempo ela existirá. Exemplos de categorias de armazenamento (ou variáveis) compreendem *automatic*, *external*, *static* e *register*. Aqui, as três primeiras são apresentadas.

Até aqui, as variáveis que temos usado nos programas têm sido definidas dentro das funções nas quais são utilizadas. Ou seja, a definição ocorre dentro do corpo da função delimitado por um par de chaves (`{}`).

5.8.3. Variável *automatic*

As variáveis definidas dentro do corpo da função são chamadas de *automatic*. Na realidade, a palavra-chave *auto* pode ser usada para especificar uma variável automática. Assim, você poderia ter:

```
void qualquerFunção() {  
    auto int qualquerVariável;  
    // tem o mesmo efeito que int qualquerVariável  
    auto float outraVariável;  
    // tem o mesmo efeito que float outraVariável  
    ...  
}
```

Contudo, desde que essa é a categoria default, não há necessidade de usar a palavra-chave *auto* (as variáveis definidas em uma função serão do tipo *automatic* por default). Adicionalmente, as variáveis do tipo *automatic* possuem duas importantes características: tempo de vida (ou existência) e visibilidade.

Uma variável do tipo *automatic* não será criada até o instante em que a função onde ela é definida seja chamada (ou, mais precisamente, até o instante em que a função é executada).

No fragmento de programa mostrado, as variáveis *qualquerVariável* e *outraVariável* não existem até o instante em que a função *qualquerFunção()* seja chamada. Quando o controle do programa é passado para *qualquerFunção()*, as variáveis são criadas, e espaço na memória é reservado para ela.

Depois que a função é abandonada e o controle retornado ao programa chamador, as variáveis são destruídas e seus valores são perdidos. O nome *auto* é usado porque as variáveis são automaticamente criadas quando a função é chamada, e automaticamente destruídas quando a função termina sua execução.

5.8.4. Tempo de Vida de Variável

O período de tempo entre a criação e a destruição da variável é denominado tempo de vida ou existência (duração). A razão de se limitar o tempo de vida de variáveis é para economizar espaço em memória. Assim, quando uma função não está sendo executada, as variáveis dessa função não são necessárias e, em consequência, tem-se mais memória disponível para outras funções em uso.

5.8.5. Visibilidade de Variável

Outra característica das variáveis é a *visibilidade*. A visibilidade de uma variável descreve onde no programa ela pode ser acessada. A palavra *escopo* é também usada para descrever a visibilidade. O escopo de uma variável é a parte do programa onde a variável é visível.

As variáveis automáticas são apenas visíveis (isto é, elas podem ser acessadas) dentro do corpo da função na qual estão definidas. Considere o fragmento de programa a seguir com as duas funções:

```
void função1() {
    int x1;        // variáveis automáticas
    float x2;
    x1 = 1;        // OK
    x2 = 2;        // OK
    x3 = 3;        // Errado: não é visível à função1()
}

void função2() {
    int x3;        // variável automática
    x1 = 11;       // Errado: não é visível à função2()
    x2 = 22;       // Errado: não é visível à função2()
    x3 = 33;       // OK
}
```

A variável *x3* é invisível para *função1()*, e as variáveis *x1* e *x2* são invisíveis para a *função2()*. Limitar a visibilidade de variáveis ajuda a organizar e modularizar o programa. Assim, podemos ficar seguros de que variáveis em uma função estão livres de ser alteradas por outras funções (desde que as outras funções não as enxerguem).

5.8.6. Variáveis Locais

Esse é um aspecto importante, tanto na programação estruturada quanto na orientada a objetos. No caso de variáveis do tipo automática, há coincidência do tempo de vida e visibilidade, uma vez que as variáveis existem quando a função está

sendo executada e são visíveis apenas dentro da função. As variáveis automáticas são também chamadas de *locais*.

Outra categoria de armazenamento é *external*. Enquanto variáveis do tipo *automatic* são definidas dentro de funções, variáveis do tipo *external* são definidas fora de qualquer função (ou externa).

5.8.7. Variável Externa

Uma variável *externa* (ou do tipo *external*) é visível a todas as funções de um programa. Mais precisamente, ela é visível a todas as funções que seguem a definição da variável na listagem. Geralmente, usamos variáveis externas a fim de que elas sejam visíveis a todas as funções. Assim, elas são colocadas no início da listagem.

5.8.8. Variáveis Globais

As variáveis externas são também chamadas de *globais*. O fragmento de programa contém duas funções com acesso a uma variável externa. A função *leCaractere()* lê os caracteres do teclado fazendo uso da função de biblioteca *getch()*, a qual é igual a *getche()* exceto que ela não ecoa o caractere digitado na tela.

```
...
char c = 'a'; // variavel externa (global)
void leCaractere() {
    ch = getch();
}
void leCaractere();
void main()
{
    while( c!= '\r' ) {
        leCaractere();
    }
}
...
```

Perceba ainda que a variável *c* não é definida em qualquer das funções (*leCaractere()* e *main()*). Observe que ela é definida no início do programa e, portanto, ela é externa. Assim, qualquer função que a segue pode ter acesso a ela.

A variável externa pode ser acessada por qualquer função no programa que apareça após sua definição. Todavia, conforme mencionado no Capítulo 1, variáveis globais (externas) criam problemas na organização de um programa, pois funções erradas podem ter acesso a elas ou as funções podem ter acesso a elas de maneira

incorreta. Em programas orientados a objeto, a necessidade de variáveis desse tipo é menor.

Uma variável externa pode ser inicializada no programa ou, do contrário, ser inicializada automaticamente para 0. Isso é diferente de variável automática que se não for inicializada pode conter algum valor arbitrário (lixo) quando de sua criação.

Adicionalmente, as variáveis externas existem durante a vida do programa (isto é, enquanto o programa está sendo executado). Isso significa que o uso de memória ocorre durante todo o tempo de execução do programa.

As variáveis externas são visíveis no arquivo onde são definidas, a partir do ponto onde são definidas. Se, no programa anterior, *c* fosse definida dentro do da função *leCaractere()*, *c* seria visível a essa função, mas não à função *main()*.

Outra categoria de variável é *static*. Nesse caso, o foco recai sobre variáveis do tipo *static automatic*. As variáveis do tipo *static external* são usadas em programas com vários arquivos.

As variáveis *static automatic* têm a visibilidade de uma variável local, porém tempo de vida (ou existência) de uma variável externa. As variáveis *static automatic* são usadas quando é necessário para uma função lembrar de um valor, mesmo que ela não esteja sendo executada (isto é, entre chamadas de função).

No fragmento de programa adiante, a função *calculaMedia()* calcula um valor de média. As variáveis do tipo *static* retêm os valores de total e contador da função *calculaMedia()* após esta retornar o controle para o programa chamador.

Além disso, as variáveis são inicializadas apenas uma vez quando da execução do programa (isto é, quando a função é chamada pela primeira vez). Essas variáveis não são reinicializadas nas chamadas subsequentes (como ocorre com as variáveis do tipo *automatic*).

```
...
double calculaMedia(double x) {
    static float total = 0; // variaveis static
    static int j = 0; // variavel contador
    j++; // increment count
    total += x; // atualiza total
    return total / j; // retorna media atualizada
}
float getavg(float); // prototype
void main()
{
    double c = 1;
    double media;
```

```
while( c!= 0 )
{
    cout << "Digite um numero: ";
    cin >> c;
    media = calculaMedia(c);
    cout << "Média atualizada " << media << endl;
}
}
```

Finalizando, a Tabela 5.1 resume o objetivo, o valor inicial, o tempo de vida e a visibilidade das variáveis automatic, static automatic e external.

Tabela 5.1 – Sumário das categorias de variáveis

	automatic	static auto	external
Objetivo	Variáveis usadas por uma única função	Idêntico a auto, porém deve reter o valor quando a função termina a execução	Variáveis usadas por várias funções
Valor inicial	Não inicializada	0	0
Tempo de vida	Função	Programa	Programa
Visibilidade	Função	Função	Arquivo

RESUMO

Neste capítulo, você aprendeu que funções servem para agrupar várias instruções em uma unidade (a função). Essa unidade pode ser chamada de outros trechos do programa. O uso de funções ajuda a organizar e modularizar o código. Você teve a oportunidade de criar funções e explorar os mecanismos de passagem de parâmetros por valor e por referência. Além disso, estudou e desenvolveu programas com funções sobrecarregadas e funções inline que podem tornar o código mais compreensível e eficiente. Por fim, foram apresentados diversos conceitos que você pode usar quando criando funções. Além disso, novos conceitos foram apresentados, e diversos exemplos foram realizados ilustrando o uso de funções. No próximo capítulo, você estudará e explorará classes e objetos.

QUESTÕES

1. O que são funções? Para que servem? Use exemplos para ilustrar sua resposta.
2. O que ocorre se os tipos de dados usados como argumentos de uma função não estão em acordo com aqueles usados na definição da função?

3. O que é uma função recursiva? Use um exemplo para ilustrar sua resposta.
4. O que é uma variável de referência? Em que situações ela pode ser empregada? Use um exemplo para ilustrar sua resposta.
5. O que são funções sobrecarregadas? Em que situações elas podem ser empregadas? Use um exemplo para ilustrar sua resposta.
6. O que são funções inline? Em que situações elas podem ser empregadas? Use um exemplo para ilustrar sua resposta.

EXERCÍCIOS

Neste capítulo, você teve a oportunidade de estudar o uso de funções em um programa orientado a objetos.

1. Faça uma pesquisa visando responder à seguinte questão: em quais situações a passagem de parâmetro por referência é adequada? Apresente um exemplo para ilustrar sua resposta.
2. Escreva uma função que calcule e exiba a média aritmética das notas digitadas de um conjunto de N alunos. O programa deve aceitar as entradas de novas notas para controlar a execução do programa.
3. Escreva uma função que implemente a conversão de moedas de dólares para reais e que possa ser chamada sempre que o usuário desejar.
4. Escreva uma função que solicite que o usuário digite as coordenadas de dois pontos $p1$ e $p2$, e retorne a distância entre esses dois pontos.
5. Escreva uma função que solicite que o usuário digite dois números $n1$ e $n2$ e retorne o maior.