

# Laços e Decisões

*As far as the laws of mathematics refers to reality, they are not certain, as far as they are certain, they do not refer to reality.*

Albert Einstein

## OBJETIVOS

- Aprender e utilizar operadores lógicos e relacionais.
- Identificar situações e utilizar laços.
- Explorar diversas estruturas de controle e situações em que empregá-las.
- Verificar a precedência existente entre os operadores.

O Capítulo 2 apresentou um conjunto de componentes da linguagem C++ que permite elaborar vários programas. Juntamente com esses componentes, características e conceitos da linguagem C++ foram apresentados. Nesse sentido, observa-se que a maioria dos programas não executa suas instruções em uma ordem rigorosa do início ao fim. Isto é, maioria dos programas decide o que fazer de acordo com as circunstâncias da ocasião. Assim, algumas questões surgem: onde devo ter controle iterativo como laços em programas? Qual o mecanismo de controle mais apropriado para o problema que tenho em mãos?

Note que o interesse recai em explorar os mecanismos de controle que você pode utilizar em um programa, e a apresentação desses mecanismos será feita com o uso de exemplos ilustrando várias aplicações. Responder às questões supracitadas compreende os propósitos deste capítulo para que você saiba como identificar qual mecanismo de controle mais adequado pode empregar em dada situação.

3.1. INTRODUÇÃO

Geralmente, a maioria dos programas não executa suas instruções de modo estritamente sequencial. Na realidade, a maioria dos programas (assim como os seres humanos) decide o que fazer de acordo com as circunstâncias. Por exemplo, se você desenvolver um programa cujo objetivo é determinar se um grupo de alunos foi aprovado ou não em uma disciplina, deve ter uma instrução no programa pela qual verificará se a média do aluno foi igual ou maior do que 6. Àqueles que tiverem média igual ou superior a 6, você atribuirá o conceito aprovado e, caso contrário, será feita a atribuição não aprovado.

Perceba que o fluxo de controle do programa *salta* de uma parte do programa para outra dependendo dos cálculos executados pelo programa. As instruções que causam esses saltos ou *jumps* são denominadas instruções de controle. Dentre as principais categorias existem *laços* e *decisões*. O número de vezes que um laço é executado, ou mesmo uma decisão que resulta na execução de parte do código, vai depender de certas expressões terem valor *verdadeiro* ou *falso*. Tais expressões, geralmente, envolvem um operador chamado *operador relacional*, o qual compara dois valores. Nesse sentido, o objetivo é identificar mecanismos de controle e decisão mais adequados para problemas que se tenham em mãos. Na sequência, um conjunto de operadores relacionais é apresentado.

3.2. OPERADORES RELACIONAIS

Computadores são máquinas incríveis que servem aos mais variados propósitos, e os programas podem fazer muito mais do que cálculos, eles também podem fazer comparações que compreendem os pilares para a tomada de decisão. Em C++, há um conjunto de seis operadores relacionais, como mostrado na Tabela 3.1.

Tabela 3.1 – Operadores relacionais

Operador	Significado
>	maior do que
<	menor do que
==	igual a
!=	não igual a
>=	maior do que ou igual a
<=	menor do que ou igual a

### 3.2.1. Operadores Relacionais

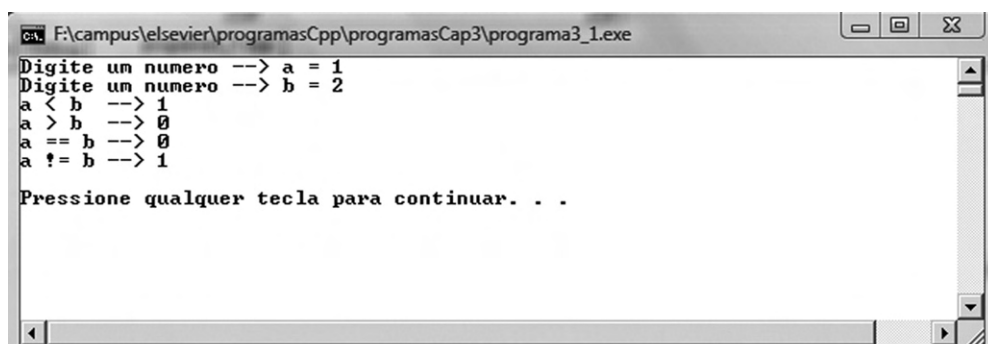
Um operador relacional compara dois valores. Esses valores podem ser tipos de dados predefinidos, como `char`, `int` e `float`, ou podem ser classes definidas pelo usuário, conforme será visto adiante. A comparação envolve relações como *igual a*, *menor do que* e *maior do que*, como ilustrado na Tabela 3.1. O resultado da comparação é do tipo booleano, que pode assumir os valores verdadeiro ou falso. Para compreender melhor, nada como um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário dois números do tipo inteiro, em seguida faça a comparação entre esses dois números (por exemplo, *a* e *b*) e retorne o resultado da comparação informando se *a* é maior, menor ou igual a *b*. Para elaborar esse programa, você deve fazer uso dos operadores relacionais apresentados na Tabela 3.1 e seguir os passos realizados nos programas anteriores. Feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.2.

```
#include <iostream>
1. using namespace std;
2. // Programa para ilustrar o uso de operadores relacionais
3. int main()
4. {
5.     int a, b;
6.     cout << "Digite um numero --> a = ";
7.     cin >> a;
8.     cout << "Digite um numero --> b = ";
9.     cin >> b;
10.    cout << "a < b --> " << (a < b) << endl;
11.    cout << "a > b --> " << (a > b) << endl;
12.    cout << "a == b --> " << (a == b) << endl;
13.    cout << "a != b --> " << (a != b) << endl << endl;
14.    system("PAUSE");
15.    return 0;
16. }
```

**Listagem 3.1**

O programa da Listagem 3.1 mostra o uso de operadores relacionais na comparação de variáveis inteiras. Vamos supor que você o tenha executado no ambiente do DevC++. Nesse caso, se entrar com os valores 1 para a variável ‘*a*’ e 2 para variável ‘*b*’, o resultado da execução será o apresentado na Figura 3.1.



```

F:\campus\elsevier\programasCpp\programasCap3\programa3_1.exe
Digite um numero --> a = 1
Digite um numero --> b = 2
a < b --> 1
a > b --> 0
a == b --> 0
a != b --> 1
Pressione qualquer tecla para continuar. . .
```

Figura 3.1 – Saída do programa da Listagem 3.1.

Esse programa realiza quatro comparações, nas linhas 10-13, entre os dois valores digitados pelo usuário. A primeira expressão é verdadeira, considerando que você digitou o valor 1 para a variável ‘a’ e o valor 2 para a variável ‘b’. A segunda expressão, na linha 11, é falsa porque 1 não é maior do que 2; a terceira (linha 12) também é falsa (porque 1 não é igual a 2); e a quarta (linha 13) é verdadeira.

■ É importante observar que o compilador C++ considera o que for verdadeiro como tendo o valor 1, enquanto, se uma expressão é avaliada como falsa, isso resulta que ela terá o valor 0. Você pode verificar isso na Figura 3.1. Além disso, embora C++ gere um ‘1’ para indicar a condição lógica de verdadeiro, é assumido que qualquer valor diferente de 0 seja verdadeiro.

### 3.3. LAÇOS

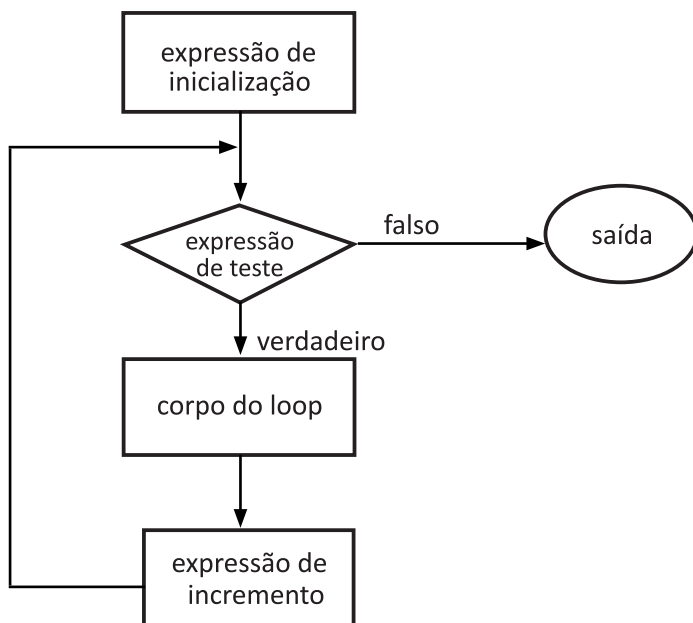
Laços ou *loops* fazem parte de um programa ser repetida um determinado número de vezes (as palavras laço e loop serão utilizadas no livro indistintamente). A repetição continua enquanto uma condição for verdadeira. Quando a condição se tornar falsa, o loop termina e o controle do programa deve prosseguir, passando às instruções seguintes ao loop. Em C++, há três tipos de loop: *for*, *while* e *do*.

#### 3.3.1. Laço for

É um dos mais fáceis. Ele executa uma seção de código um número fixo de vezes. Geralmente, é utilizado quando se sabe antecipadamente (antes de entrar no loop) o número de iterações que um determinado trecho de código será executado, em outras palavras, quantas vezes a seção de código deverá ser executada.

Para entender melhor, considere a Figura 3.2, que ilustra a operação do laço *for*. Observe que a expressão de inicialização é executada apenas uma vez, enquanto

a expressão de teste é executada toda vez que o corpo do laço é executado. Já a expressão de incremento atualiza a variável do laço após cada execução do corpo do laço, enquanto a condição de teste for avaliada como verdadeira. Agora, é hora de um exemplo.



**Figura 3.2** – Operação do laço for.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro ( $n$ ) e, em seguida, faça  $n$  iterações num loop de for. A cada iteração, o programa deve calcular o quadrado do valor (inteiro) de 1 até  $n$  (dentro do laço for), exibindo os respectivos valores. Feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.2.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do loop for
4. int main()
5. {
6.     int i; // define a variavel de loop
7.     int n; // define variavel limite para iteracoes no loop
8.     cout << "Digite um valor: ";
9.     cin >> n;
10.    for(i=1; i <= n; i++) // executa o loop de 1 ate n
  
```

```
11. cout << "Valor do quadrado de " << i << " = " << i * i << endl;  
12. system("PAUSE");  
13. return 0;  
14. }
```

### Listagem 3.2

Quando você executa o programa da Listagem 3.2, o programa solicita que o usuário digite um valor. Vamos supor que você digitou 4. Então, o programa irá para o laço da linha 10 calculando e exibindo o quadrado dos números de 1 até 4, conforme mostrado na Figura 3.3.

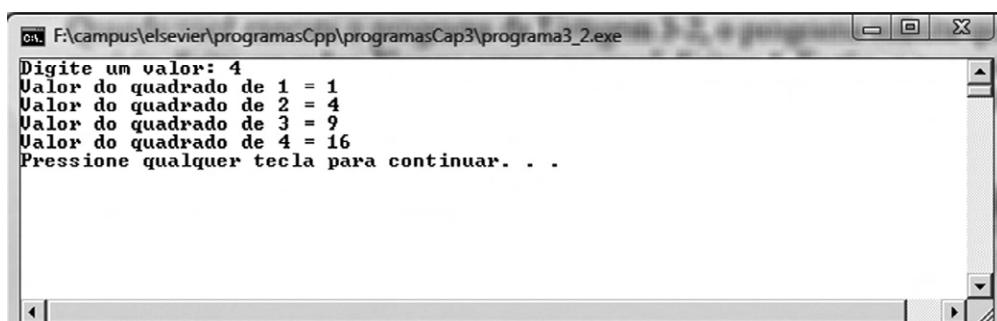


Figura 3.3 – Saída do programa da Listagem 3.2.

### 3.3.2. Instrução for

Observe que a instrução *for* (linha 10) serve para controlar o laço. Ela consiste na palavra-chave *for*, seguida de parênteses que contêm três expressões separadas por ponto-e-vírgula. A primeira expressão é a *inicialização*, a segunda é uma *expressão de teste* e a terceira é uma *expressão de incremento*. O *corpo do loop* é o que é executado enquanto no loop. No exemplo da Listagem 3.2, o corpo do laço está contido na linha 11.

---

■ É importante ressaltar que a expressão de inicialização é executada apenas uma vez, enquanto a expressão de teste é executada toda vez que o corpo do laço é executado. Além disso, a expressão de incremento atualiza a variável do laço após cada iteração (ou seja, após cada execução do corpo do laço).

Se você observar o código da Listagem 3.2, pode verificar que o laço *for* (da linha 10) contém uma única instrução, isto é, a da linha 11. Todavia, você pode também ter a situação em que mais de uma instrução é executada no corpo do laço

ou loop. Nesse caso, múltiplas instruções devem ser delimitadas por chaves. Um programa de exemplo é mostrado na Listagem 3.3. Perceba que não existe ponto-e-vírgula após o fecha-chaves (}) do laço for. A saída do programa da Listagem 3.3 é mostrada na Figura 3.4.

```

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. // Programa para ilustrar a operacao do loop for
5. int main()
6. {
7.     int i; // define a variavel de loop
8.     int n; // define variavel limite para iteracoes no loop
9.
10.    cout << "Digite um valor: ";
11.    cin >> n;
12.    cout << "\nValor " << setw(5) << "Quadrado" << endl;
13.    for(i=1; i <= n; i++) { // executa o loop de 1 ate n
14.        cout << setw(5) << i;
15.        int n2 = i * i; // define variavel n2 que recebe o quadrado
            de n
16.        cout << setw(9) << n2 << endl;
17.    }
18.    system("PAUSE");
19.    Return 0;
20. }
```

Listagem 3.3

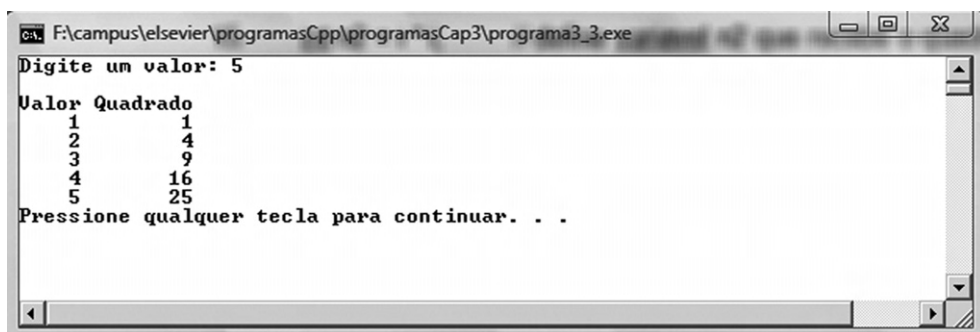


Figura 3.4 – Saída do programa da Listagem 3.3.

■ Observe que a variável *n2* não é visível fora do bloco de código no qual ela foi definida. Visível aqui significa que as instruções do programa podem ter acesso ou “ver” a variável. A vantagem em restringir a visibilidade de uma variável é poder usar o mesmo nome de variável em diferentes blocos de código no mesmo programa.

Inicialmente, o programa da Listagem 3.3 solicita que o usuário digite um valor. No exemplo, foi digitado o valor 5 e, com isso, o programa exibe em formato de duas colunas os números 1 a 5 e seus respectivos valores ao quadrado. Observe que o corpo do laço consiste em três instruções, o que é denominado bloco de código. Perceba que esse bloco de código faz uso da *indentação*, a qual constitui um bom estilo de programação. Note ainda que a variável `n2` é definida na linha 9 e, em seguida, calculada (na linha 15) dentro do bloco.

Nos programas anteriores, você utilizou a expressão de incremento para incrementar uma variável de controle. Contudo, ela também pode decrementar (em vez de incrementar) a variável de um loop, como destacado no exemplo a seguir.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro ( $n$ ) e, em seguida, faça  $n$  iterações num loop de `for` com o objetivo de determinar o valor do fatorial do número  $n$  que digitou. Ao final do laço `for`, o programa deverá exibir o valor do fatorial de  $n$ . Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.4.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do loop for
4. int main()
5. {
6.     unsigned int n; // declara variavel n
7.     unsigned long fatorial = 1; // declara variavel fatorial
        como do tipo long
8.     cout << "Digite um valor: ";
9.     cin >> n; // ler o valor de n
10.    for(int i = n; i > 0; i--)
11.        fatorial *= i; // calcula o fatorial de n
12.    cout << "Fatorial de " << n << " = " << fatorial << endl <<
        endl;
13.    system("PAUSE");
14.    return 0;
15. }
```

**Listagem 3.4**

Executando o programa da Listagem 3.4, você obterá a saída apresentada na Figura 3.5. No exemplo, considera-se que o usuário tenha digitado o valor 5, resultando no fatorial calculado de 120.



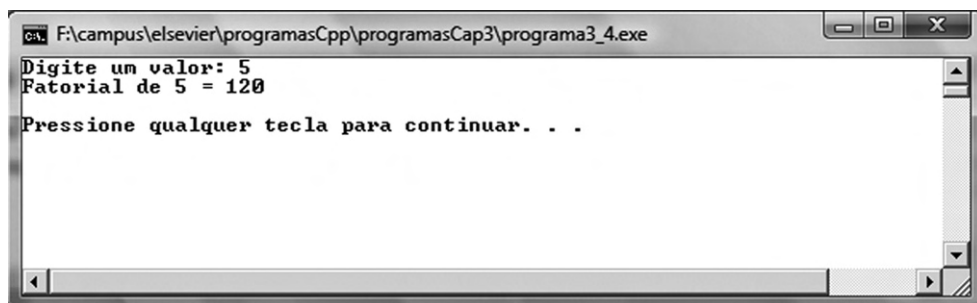


Figura 3.5 – Saída do programa da Listagem 3.4.

Note que  $i$  é inicializado como o valor digitado pelo usuário. Perceba também que  $i$  é definido dentro da instrução *for*. Outra opção que se tem com laços *for* é utilizar múltiplas inicializações e testes, tal como exemplificado a seguir.

```
for( i = 0, c = 100; c < 50; i++, c-- )
{
    // corpo do laço for
}
```

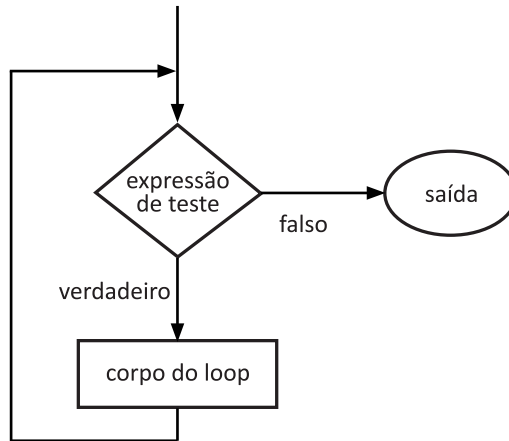
### 3.3.3. Loop while

Se você analisar cuidadosamente o que já estudou, perceberá que o laço *for* executa um bloco de código um número fixo de vezes. Entretanto, você pode se deparar com uma situação na qual não saiba quantas vezes fazer alguma coisa (como, por exemplo, executar uma seção de código) antes de iniciar o laço. Nesse caso, o que fazer?

Uma solução é utilizar o loop *while*. A Figura 3.6 ilustra a operação do loop *while*. Trata-se de um laço simples no qual uma condição é checada e, enquanto ela for verdadeira, o corpo do laço é executado.

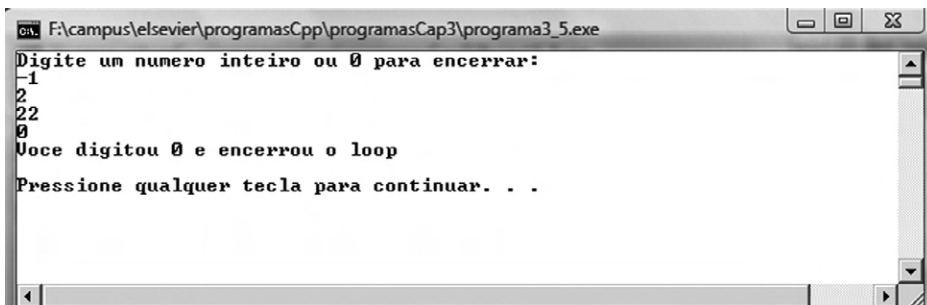
---

■ É importante observar que, embora o uso de múltiplas expressões torne o programa mais conciso, também reduz a habilidade de leitura e entendimento do programa. Portanto, sempre que possível, é uma boa prática de programação usar essas expressões de forma separada (em vez de combinada, como no exemplo anterior).

**Figura 3.6** – Operação do laço for.

A Listagem 3.5 é uma ilustração de como o laço *while* pode ser utilizado. Nesse exemplo, o programa solicita que o usuário digite um número inteiro qualquer ou 0 para encerrar o loop. Enquanto isso não ocorre, o programa fica em loop, aguardando a entrada de outro número. A saída desse programa é mostrada na Figura 3.7.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do loop while
4. int main()
5. {
6.     int n = 1; // inicializar o valor de n!= 0
7.     cout << "Digite um numero inteiro ou 0 para encerrar: \n";
8.     while( n!= 0 ) // executar o laco até n = 0
9.     {
10.         cin >> n;
11.         cout << "Voce digitou " << n << " e encerrou o loop\n" << endl;
12.         system("PAUSE");
13.     }
14. }
```

**Listagem 3.5****Figura 3.7** – Saída do programa da Listagem 3.5.

■ *Note que o loop while pode ser considerado uma versão simplificada do loop for, mas o loop while, diferentemente do for, não tem expressões de inicialização e de incremento.*

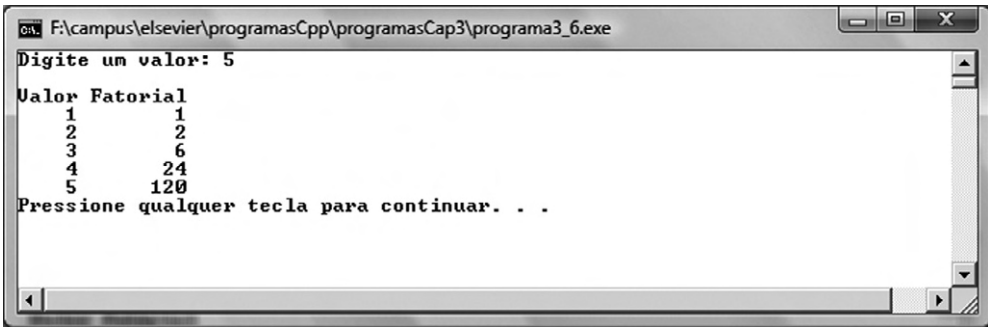
**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro ( $n$ ) e, em seguida, faça  $n$  iterações em loop de while com o objetivo de determinar o valor do fatorial do número  $n$  que digitou. Durante cada iteração do while você deve exibir o valor atual de uma variável  $i$  (que atua como contador) e respectivos valores do fatorial. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.6.

```

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. // Programa para ilustrar a operacao do loop while
5. int main()
6. {
7.     int i = 1; // define e inicializa a variavel de loop
8.     int n; // define variavel limite para iteracoes no loop
9.     int fatorial = 1; // define e inicializa variavel fatorial
10.    cout << "Digite um valor: ";
11.    cin >> n;
12.    cout << "\nValor " << setw(5) << "Fatorial" << endl;
13.    while(i <= n) { // executa o loop de 1 ate n
14.        cout << setw(5) << i;
15.        fatorial *= i;
16.        cout << setw(9) << fatorial << endl;
17.        ++i;
18.    }
19.    system("PAUSE");
20.    return 0;
21. }
```

### Listagem 3.6

A Listagem 3.6 apresenta um programa que faz uso do loop *while* com múltiplas instruções dentro do loop. O programa calcula o fatorial de números de 1 até o valor  $n$  que você tenha digitado. A saída do programa da Listagem 3.6 é mostrada na Figura 3.8.



```
on. F:\campus\elsevier\programasCpp\programasCap3\programa3_6.exe
Digite um valor: 5
Valor Fatorial
1      1
2      2
3      6
4     24
5    120
Pressione qualquer tecla para continuar. . .
```

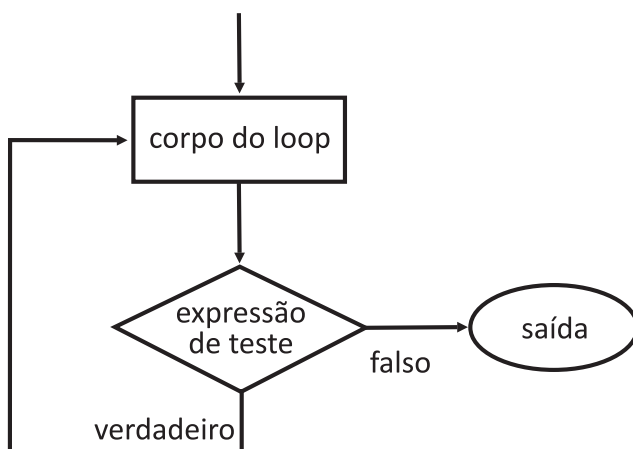
Figura 3.8 – Saída do programa da Listagem 3.6.

### 3.3.4. Loop do

Se você analisar a Listagem 3.6, verá que em um laço *while* a expressão de teste é avaliada apenas no início do laço. Se o resultado da expressão de teste é falso quando o laço é entrado, o corpo do laço não é executado.

Entretanto, há situações em que se pode necessitar que o laço seja executado pelo menos uma vez, não importando o estado inicial da expressão de teste. Nesse caso, a expressão de teste deve ser colocada no fim do laço, e, portanto, em tais situações deve-se usar o laço *do*. Para entender como isso funciona, veja a Figura 3.9, que ilustra a operação do laço *do*. Note que a expressão *while* funciona como expressão de teste (no final do laço) e controla o término do laço. E, para entender melhor, nada melhor do que explorar um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro ( $n$ ) qualquer e, em seguida, o programa deve calcular o quadrado e a raiz quadrada do valor de  $n$ . Após exibir esses dois resultados, o programa deve solicitar se o usuário deseja continuar. Se o usuário digitar s (sim), o programa repete o procedimento; se o usuário digitar n (não), o programa é encerrado. Na solução desse exemplo, você deve utilizar o laço *do* para controlar a execução do programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.7.



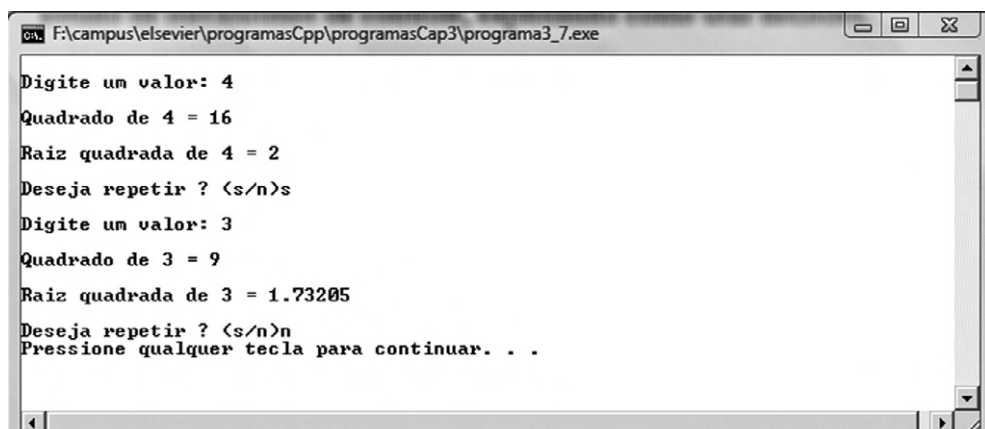
**Figura 3.9** – Operação do laço do.

```

1. #include <iostream>
2. #include <math.h> // necessario para funcao sqrt()
3. using namespace std;
4. // Programa para ilustrar a operacao do loop do
5. int main()
6. {
7.     int i = 1; // define e inicializa a variavel de loop
8.     int n; // define variavel limite para iteracoes no loop
9.     int fatorial = 1; // define e inicializa variavel fatorial
10.    char entrada;
11.    do { // executa o loop de 1 ate n
12.        cout << "\nDigite um valor: ";
13.        cin >> n;
14.        cout << "\nQuadrado de " << n << " = " << n * n << endl;
15.        cout << "\nRaiz quadrada de " << n << " = " << sqrt(n) <<
            endl;
16.        cout << "\nDeseja repetir? (s/n)";
17.        cin >> entrada;
18.    } while (entrada != 'n');
19.    system("PAUSE");
20.    return 0;
21. }
  
```

**Listagem 3.7**

Observe na Listagem 3.7 que a instrução do na linha 11 faz com que as linhas 12-17 sejam executadas pelo menos uma vez, quando na linha 18 é feito um teste com a instrução `while` (`entrada != 'n'`) para checar se esse corpo do laço (linhas 12-17) deve ser executado outra vez. A Figura 3.10 ilustra a saída do programa da Listagem 3.7. Na seção seguinte, você dará continuidade ao estudo de mecanismos de controle, explorando como usar decisões.

A imagem mostra uma janela de terminal com o título "F:\campus\elsevier\programasCpp\programasCap3\programa3\_7.exe". O conteúdo da janela é o seguinte:

```
Digite um valor: 4
Quadrado de 4 = 16
Raiz quadrada de 4 = 2
Deseja repetir ? <s/n>s
Digite um valor: 3
Quadrado de 3 = 9
Raiz quadrada de 3 = 1.73205
Deseja repetir ? <s/n>n
Pressione qualquer tecla para continuar. . .
```

Figura 3.10 – Saída do programa da Listagem 3.7.

## 3.4. DECISÕES

No seu cotidiano, os seres humanos precisam tomar decisões. Por exemplo, você pode ter em algum momento que tomar decisão sobre o que beber: um suco de laranja, um refrigerante ou um café.

Similarmente, os programas precisam tomar decisões. No caso de um programa, uma decisão pode causar um salto (*jump*) de uma parte do programa para outra parte diferente, dependendo do valor de uma expressão.

### 3.4.1. if... else

Decisões em C++ podem ocorrer de várias formas. Dentre elas, a mais importante é a instrução *if... else*, a qual escolhe uma entre duas alternativas. Essa instrução pode vir sem a parte *else* (ou seja, contendo simplesmente o *if*). A expressão *if* é seguida por uma expressão de teste, como no exemplo a seguir:

```
if( a > 0 )
    cout << "a é um valor positivo\n";
```

Note que, se valor da variável for maior do que 0, uma mensagem (a é um valor positivo) será exibida na tela. A Figura 3.11 ilustra a operação da instrução

*if*. Perceba que o corpo do *if* é executado apenas uma vez se a expressão de teste é verdadeiro. Agora, é hora de explorar um exemplo.

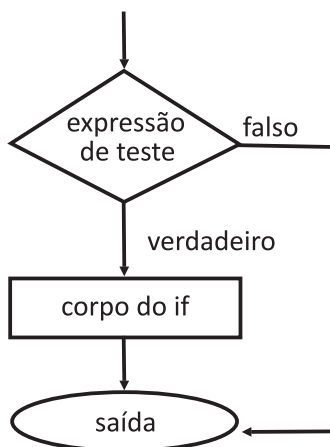


Figura 3.11 – Operação do laço if.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digite um número inteiro ( $n$ ) qualquer. Em seguida você deve verificar se esse número  $n$  é divisível por, utilizando o operador % (resto) e comparando com o 0 (zero). Caso  $n$  seja múltiplo de 3, uma mensagem deve ser exibida na tela. na solução desse exemplo, você deve utilizar o laço *if* para controlar a execução do programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.8.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do loop if
4. int main()
5. {
6.     int n; // define variavel limite para iteracoes no loop
7.     cout << "\nDigite um valor: ";
8.     cin >> n;
9.     if (n % 3 == 0) {
10.         cout << "\nO valor " << n << " que voce digitou e multiplo
            de 3" << endl;
11.         cout << "e tem resto " << n % 3 << endl << endl;
12.     }
13.     system("PAUSE");
14.     return 0;
15. }
  
```

Listagem 3.8

A Listagem 3.8 ilustra o uso do *if* com múltiplas instruções no corpo do laço. Na linha 9, você testa se o valor digitado de *n* é múltiplo de 3, fazendo `if (n % 3 == 0)`. Se essa condição for verdadeira, as instruções das linhas 10 e 11 são executadas, causando a exibição da mensagem de que *n* é múltiplo de 3. O resultado dessa execução é mostrado na Figura 3.12.

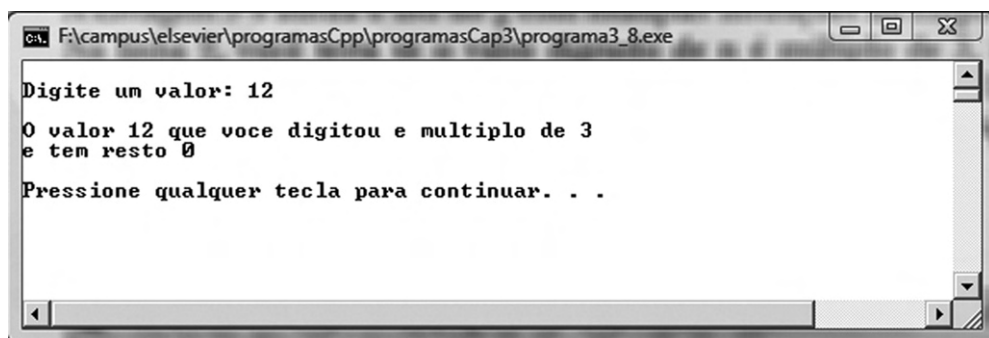


Figura 3.12 – Saída do programa da Listagem 3.8.

### 3.4.2. Aninhamento (*Nesting*) de *ifs* (dentro de laços)

As estruturas de decisão e laço que você tem visto até aqui podem ser aninhadas uma dentro da outra. Como? Você pode aninhar *ifs* dentro de *laços* e vice-versa, *ifs* dentro de *ifs*, e assim por diante. Para entender melhor, vejamos um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro (*n*) qualquer. Em seguida você deve verificar se esse número *n* é primo. Para checar se *n* é primo, você divide *n* sucessivamente por 2 e verifica se o resto é 0. Se essa condição for verdadeira, *n* não é primo. Caso contrário, *n* é primo. Em qualquer situação, você deve exibir o resultado na tela. Na solução desse exemplo, você deve fazer uso de dois laços aninhados para controlar a execução do programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.9.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do loop aninhados
4. int main()
5. {
6.     unsigned long n, i;
7.     cout << "Digite um valor: ";
8.     cin >> n; // ler um valor de entrada
9.     for(i = 2; i <= n/2; i++) // realiza divisoes sucessivas
        por 2
```



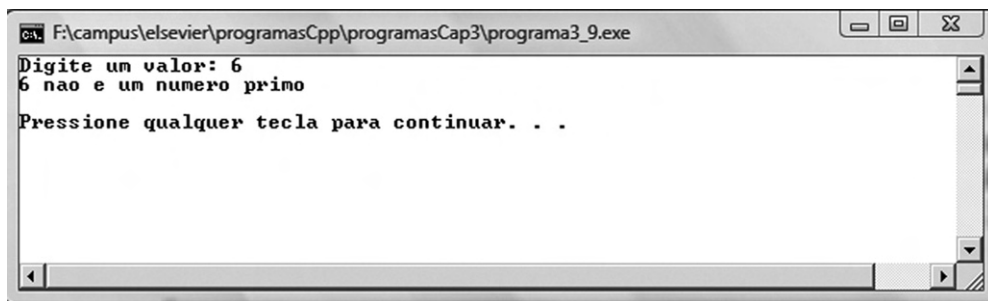
```

10.     if(n % i == 0) { // se o resto é igual a 0, então n é
        divisível por i
11.         cout << n << " não é um número primo" << endl << endl;
12.         system("PAUSE");
13.         return 0;
14.     }
15.     cout << n << " é um número primo" << endl << endl;
16.     system("PAUSE");
17.     return 0;
18. }

```

**Listagem 3.9**

Na Listagem 3.9, você tem um *if* aninhado dentro de um loop *for*. O programa pede para o usuário entrar um número e diz se esse número é um número primo. Cabe lembrar que números primos são inteiros divisíveis por ele mesmo e por 1 apenas. A saída do programa da Listagem 3.9 é exibida na Figura 3.13.



**Figura 3.13** – Saída do programa da Listagem 3.9.

É importante você observar, na Listagem 3.9, que não há chaves (*{}*) para o corpo do loop *for*, isto é, nas linhas 10-14. Isso ocorre porque a instrução *if* (e as instruções de *if*) é considerada como uma única instrução.

Adicionalmente, note que, quando descobre que um número não é primo, o programa é terminado, desde que não haja mais necessidade de provar novamente que o número não é primo. Isso é feito através do uso das instruções das linhas 12 e 13.

### 3.4.3. *if... else*

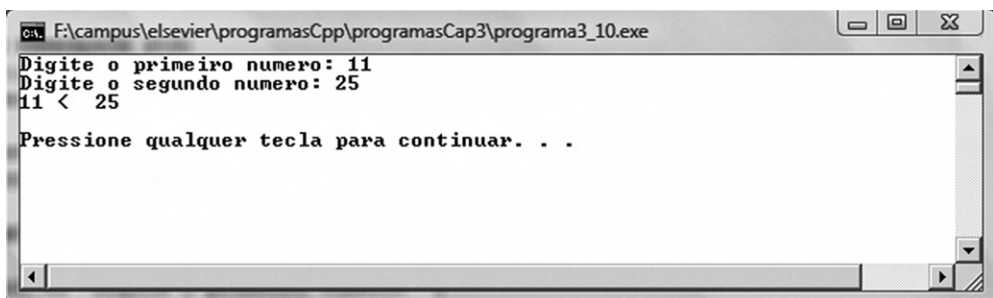
A instrução *if* permite ter uma ou mais ações executadas se sua condição é verdadeira. Se ela é falsa, nada acontece. Todavia, se você desejar fazer alguma coisa quando a condição de *if* é verdadeira e outra coisa quando é falsa, você pode usar *if... else*. Para entender melhor como isso pode ser realizado, vamos explorar um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário a entrada de dois números inteiros ( $n_1$  e  $n_2$ ) quaisquer. Em seguida, seu programa deve verificar qual desses números é maior e exibir o resultado da comparação. Na solução desse exemplo, você deve fazer uso da estrutura *if...else* para controlar a execução do programa. Se a condição *if* for verdadeira,  $n_1$  é maior ou igual a  $n_2$ . Caso contrário, no bloco *else*,  $n_2$  é maior. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.10.

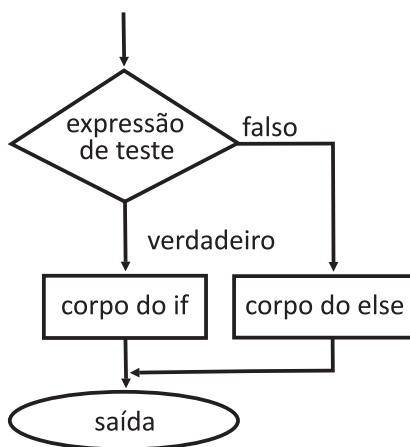
```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar a operacao do if...else
4. int main()
5. {
6.     unsigned long n1, n2;
7.     cout << "Digite o primeiro numero: ";
8.     cin >> n1; // ler primeiro numero de entrada
9.     cout << "Digite o segundo numero: ";
10.    cin >> n2; // ler segundo numero de entrada
11.    if(n1 >= n2) // compara n1 e n2
12.        cout << n1 << " >= " << n2 << endl << endl;
13.    else
14.        cout << n1 << " < " << n2 << endl << endl;
15.    system("PAUSE");
16.    return 0;
17. }
```

**Listagem 3.10**

A saída do programa da Listagem 3.10 é mostrada na Figura 3.14. Observe que a operação *if... else* tem duas porções (*if* e *else*). Isto é, após o teste da condição, o corpo de *if* será executado se essa condição for verdadeira ou o corpo de *else* é executado, como ilustrado na Figura 3.15.



**Figura 3.14** – Saída do programa da Listagem 3.10.



**Figura 3.15** – Operação do laço if...else.

#### 3.4.4. Função `getche()`

Agora, você irá explorar o uso da função `getche()`. Essa função é similar a `getch()`. A função `getche()` ecoa cada caractere digitado (na tela). Já a função `getch()` faz a mesma coisa, porém não ecoa na tela o que foi digitado. Note que ecoar significa tornar visível na tela. Para entender mais, vamos explorar um exemplo.

A Listagem 3.11 apresenta um programa que usa a instrução `if... else` dentro de um loop `while`. Esse programa conta o número de caracteres e de palavras de uma frase que você tenha digitado. Até aqui, você tem utilizado o `cin` para entrada, o qual exige que o usuário sempre pressione a tecla *enter* para informar ao programa que a entrada está completa. No programa da Listagem 3.11, o programa processa cada caractere sem esperar por um *enter*. Entretanto, essa função exige a inclusão do arquivo `conio.h` e não requer qualquer argumento. A Figura 3.16 ilustra a saída da execução do programa.

```

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. // Programa para ilustrar a operacao da funcao getche()
5. int main()
6. {
7.     int numCaracteres = 0;
8.     int numPalavras = 1; // numero de espacos entre palavras
9.     char ch;
10.    cout << "Digite uma frase: ";

```

```
11. while( (ch=getche())!= '\r' ) {      // loop ate que enter
    seja digitado
12. if( ch==' ' )      // testa se é espaço em branco
13.     numPalavras++; // conta palavras
14. else
15.     numCaracteres++; // conta caracteres
16. }
17. cout << "\nNumero de palavras = " << numPalavras << endl <<
    endl;
18. cout << "Numero de caracteres = " << numCaracteres << endl
    << endl;
19. system("PAUSE");
20. return 0;
21. }
```

Listagem 3.11

■ Note que os parênteses ao redor da expressão `(ch = getche())` são necessários porque o operador de atribuição `=` tem precedência menor do que o operador relacional `!=`. Portanto, sem os parênteses, a expressão seria avaliada como `while(ch = (getche()) != 'n')`.

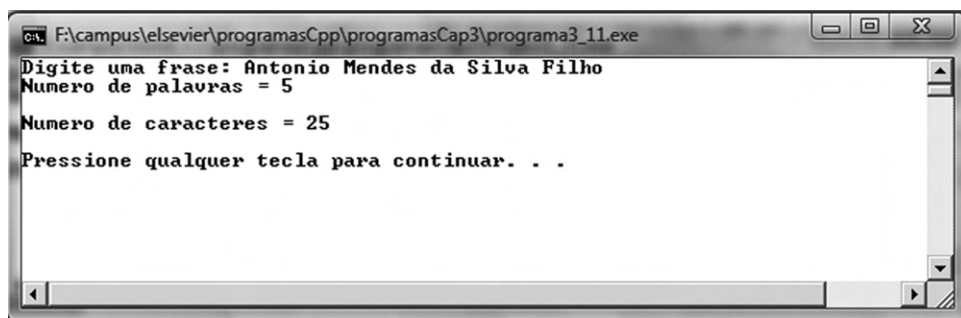


Figura 3.16 – Saída da execução do programa da Listagem 3.11.

### 3.4.5. if... else aninhados

As instruções *if... else* podem ser empregadas de forma aninhada. Entretanto, é preciso total atenção no uso de instruções *if... else* aninhadas porque elas podem causar erros. Portanto, para usar de modo adequado, você deve estar atento para a regra:

*Um else deve estar casado com o último if que não possui o próprio else.*

Para entender melhor, vamos explorar o exemplo da Listagem 3.12, que faz um casamento de um *else* com o *if*errado. O que ocorre se entrarmos com os valores 1, 1 e 2 para *x*, *y* e *z*, respectivamente? O programa responde que *x* e *y* são diferentes, conforme mostrado na Figura 3.17.

```

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. // Programa para ilustrar a operacao da funcao getche()
5. int main()
6. {
7.     int x, y, z;
8.     cout << "Digite 3 numeros x, y e z: ";
9.     cin >> x >> y >> z;
10.    if( x == y )
11.        if( y == z )
12.            cout << "\nx, y e z sao iguais\n" << endl;
13.    else
14.        cout << "\nx e y sao diferentes\n" << endl;
15.    system("PAUSE");
16.    Return 0;
17. }

```

Listagem 3.12

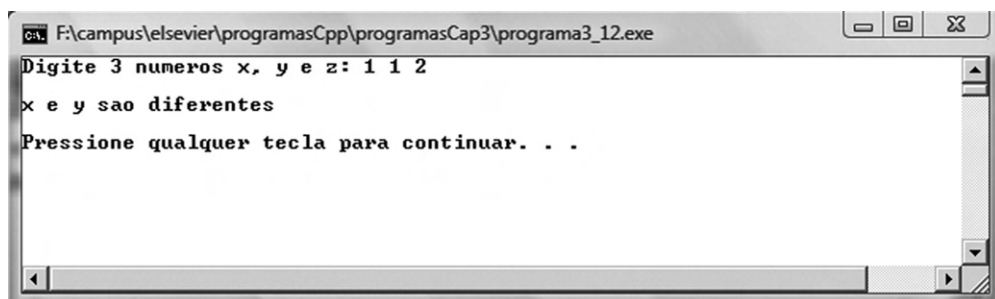


Figura 3.17 – Saída do programa da Listagem 3.12.

É importante destacar que a indentação é enganosa. Portanto, para usar de modo adequado, você deve estar atento para a regra destacada anteriormente que requer que cada *else* esteja casado com o *if* imediatamente anterior (que não possui o próprio *else*). Dessa forma, o modo correto de utilizar é:

```

if( x == y )
    if( y == z )
        cout << "x, y e z sao iguais";
    else
        cout << "y e z sao diferentes"; [YN]

```

Outra forma é:

```

if( x == y )
{

```

```
if( y == z )
    cout << "x, y e z sao iguais";
}
else
    cout << "x e y sao diferentes";
```

Agora, para entender mais, nada melhor do que explorar um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar *n*, *s*, *e* e *w* para permitir ir para norte, sul, leste e oeste, respectivamente. Inicialmente, você assume que ele se encontra na posição  $[x, y] = [0, 0]$ . Se você, por exemplo, digitar 'n', ele irá para a posição  $[0, 1]$ . Em seguida, se você digitar 'o', a nova posição será  $[-1, 1]$ . Quando desejar finalizar, basta teclar Enter. Na solução desse exemplo, você deve fazer uso de múltiplas estruturas de controle em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.13.

```
1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. // Programa para ilustrar o uso de multiplas estruturas de
   controle
5. int main()
6. {
7.     system ("cls"); // Limpa a tela
8.     system ("color F0"); // Torna cor de fundo branca e as le-
   tras pretas
9.     char entrada = 'z';
10.    int x = 0, y = 0;
11.    cout << "Digite n(norte), s(sul), l(leste), o(oeste) ou
   tecle Enter para finalizar\n";
12.    while( entrada!= '\r' ) {
13.        cout << "\nPosicao = [" << x << ", " << y << "]" << endl;
14.        cout << "\nDigite n(norte), s(sul), l(leste), o(oeste) ou
   tecle Enter para finalizar\n";
15.        cout << "\nDirecao = ";
16.        entrada = getche(); // ler entrada
17.        if( entrada == 'n' ) // direcao norte
18.            y++;
19.        else
20.            if( entrada == 's' ) // direcao sul
21.                y--;
22.        else
23.            if( entrada == 'l' ) // direcao leste
```

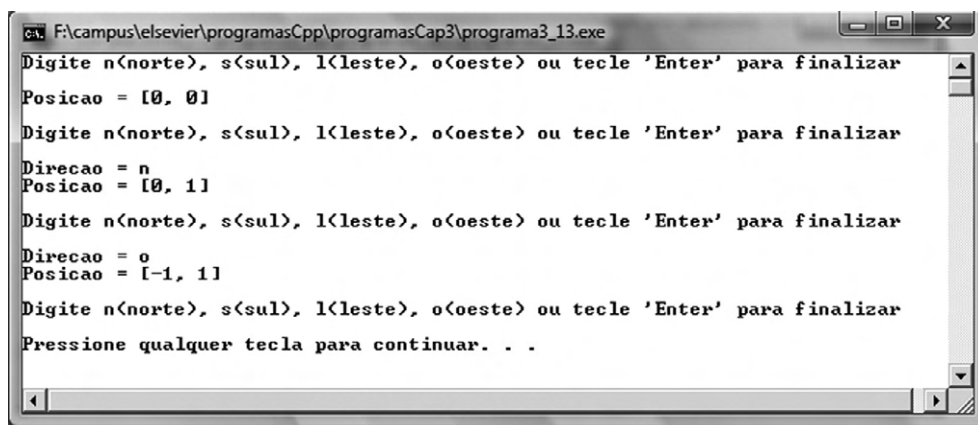
```

24.     x++;
25.     else
26.         if( entrada == 'o' ) // direcao oeste
27.             x--;
28.     }
29.     system("PAUSE");
30.     return 0;
31. }

```

**Listagem 3.13**

Agora, executando o programa da Listagem 3.13, o programa solicita que você digite a direção na qual quer se mover ou para qual coordenada deseja ir. Inicialmente, você assume que a posição inicial é  $[x, y] = [0, 0]$ . Se, por exemplo, digitar 'n', ele irá para a posição  $[0, 1]$ . Em seguida, se você digitar 'o', a nova posição será  $[-1, 1]$ , como ilustrado na Figura 3.18. Quando desejar finalizar, basta teclar Enter.



**Figura 3.18** – Saída do programa da Listagem 3.13.

### 3.4.6. else... if

As instruções *if... else* do programa da Listagem 3.13 não são tão fáceis de ser entendidas, e mais ainda se estiverem aninhadas muito profundamente. Dessa forma, outra maneira de escrever o código da Listagem 3.13 seria usar a construção *else... if*, conforme Listagem 3.14.

```

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. // Programa para ilustrar o uso de multiplas estruturas de
   controle

```

```
5. int main()
6. {
7.     char entrada = 'z';
8.     int x = 0, y = 0;
9.     cout << "Digite n(norte), s(sul), l(leste), o(oeste) ou
    tecle Enter para finalizar\n";
10.    while( entrada!= '\r' ) { // executar ate Enter ser digitado
11.        cout << "\nPosicao = [" << x << ", " << y << "]" << endl;
12.        cout << "\nDigite n(norte), s(sul), l(leste), o(oeste) ou
    tecle Enter para finalizar\n";
13.        cout << "\nDirecao = ";
14.        entrada = getche(); // ler entrada
15.        if( entrada == 'n' ) // direcao norte
16.            y++;
17.        else if( entrada == 's' ) // direcao sul
18.            y--;
19.        else if( entrada == 'l' ) // direcao leste
20.            x++;
21.        else if( entrada == 'o' ) // direcao oeste
22.            x--;
23.    }
24.    system("PAUSE");
25.    return 0;
26. }
```

Listagem 3.14

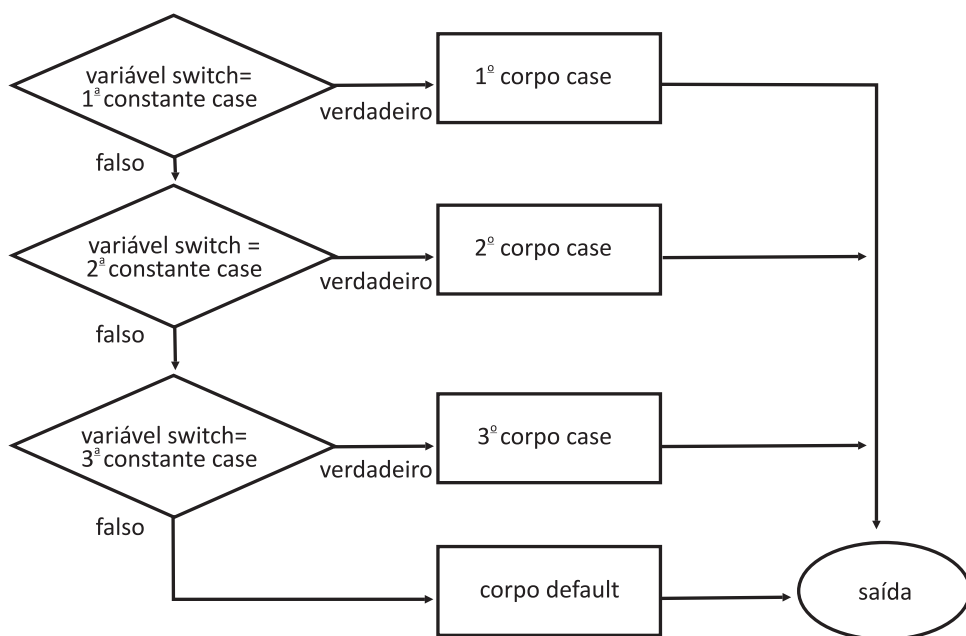
### 3.4.7. Switch

Outra instrução de decisão é o *switch*, que permite ter múltiplas ramificações para múltiplas seções de código, as quais dependem do valor de uma única variável.

Considere um problema que requeira várias decisões e que todas as decisões dependam do valor da mesma variável. Em tal situação, você pode utilizar o *switch* em vez das outras construções vistas anteriormente (*if... else* ou *else... if*). Para tanto, deve usar a palavra-chave *switch* seguida por uma variável entre parênteses como, por exemplo, *switch(variável)*.

Adicionalmente, para cada uma das decisões, você deve ter um *case* que contenha as instruções a serem realizadas se a condição do *case* é satisfeita. Essas instruções são delimitadas por *chaves* (*{}*). Cada palavra-chave *case* é seguida por uma constante e dois-pontos (:). Os tipos de dados das constantes devem casar com o tipo de dado da variável. Essa variável é testada para verificar se ela casa com a constante. Se esse casamento ocorre, o corpo do *case* é executado, conforme ilustrado na Figura 3.19.





**Figura 3.19** – Operação da instrução switch.

Observe que, se não ocorre um casamento entre o valor da variável *switch* e uma das constantes *case*, o controle vai para a primeira instrução seguindo o *case*. Agora, para entender mais, vamos explorar um exemplo.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário a opção de escolher digitar *n*, *s*, *e* e *w*, indicando direções norte, sul, leste e oeste, respectivamente. O usuário também deve ter a opção de sair, teclando Enter. E, se você digitar qualquer outro caractere diferente de *n*, *s*, *e* e *w*, deve receber uma notificação do fato e encerrar o programa. Para a solução desse problema, você deve fazer uso do switch em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.15.

```

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. // Programa para ilustrar o uso de multiplas estruturas de
   controle
5. int main()
6. {

```

```
7.  char entrada = 'z';
8.  cout << "\nDigite n(norte), s(sul), l(leste), o(oeste) ou
    Enter para sair:";
9.  entrada = getche();    // entrada do usuario
10. switch(entrada) {      // selecao da direcao
11.     case 'n':
12.         cout << "\nVoce escolheu direcao Norte\n\n";
13.         break;
14.     case 's':
15.         cout << "\nVoce escolheu direcao Sul\n\n";
16.         break;
17.     case 'l':
18.         cout << "\nVoce escolheu direcao Leste\n\n";
19.         break;
20.     case 'o':
21.         cout << "\nVoce escolheu direcao Oeste\n\n";
22.         break;
23.     case '\r':
24.         cout << "\nVoce escolheu sair do programa\n\n";
25.         break;
26.     default:
27.         cout << "\nVoce escolheu opcao errada\n";
28.         break;
29. }
30. system("PAUSE");
31. return 0;
32. }
```

**Listagem 3.15**

A solução da Listagem 3.15 faz uso de `switch` com uma variável do tipo `char`. Esse programa imprime, por exemplo, uma mensagem `Voce escolheu direcao norte` (conforme linha 12) se o usuário tiver digitado `n`, havendo um casamento com a variável `entrada` do `switch` (da linha 11). Note que a palavra-chave *switch* (linha 10) é seguida por uma variável entre parênteses como, por exemplo, *switch(variável)*. As *chaves* (`{}`) são utilizadas para delimitar várias instruções *case*.

Quando uma constante casa com o valor da variável *switch*, as instruções seguintes à instrução *case* são executadas até que o *break* seja encontrado. Isso é ilustrado na Figura 3.20, ocorrendo quando você executa o programa da Listagem 3.15 e digita 'n'. Quando o *break* é encontrado, o controle do programa vai para primeira instrução após a instrução *switch*. Se não existe o *break*, o controle vai para a(s) próxima(s) instrução(ões) *case*.

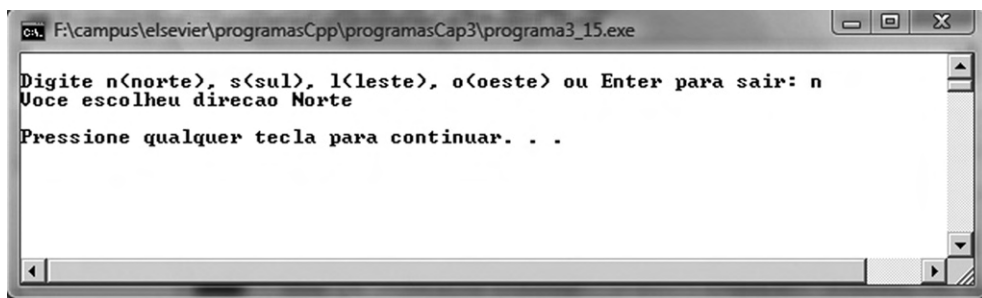


Figura 3.20 – Saída do programa da Listagem 3.15.

■ Note que você pode usar os tipos `int` ou `char` como variáveis de `switch`. Contudo, não pode usar números de ponto flutuante. Você pode ainda utilizar a palavra `default` (linha 26 da Listagem 3.15) para fazer com que uma determinada ação seja realizada, caso nenhuma das variáveis anteriores do `switch` tenha sido satisfeita com as constantes do `case`.

Agora uma possível pergunta que você pode ter em mente é: quando se usar `if... else` (ou `else... if`) e quando se usa `switch`?

Você poderá usar `if... else` em expressões que envolvem variáveis não relacionadas e que sejam complexas. Por outro lado, em uma instrução `switch`, todas as ramificações são selecionadas por uma mesma variável. A única coisa que difere uma ramificação da outra é o valor dessa variável e, portanto, você deve utilizar uma instrução `switch` sempre que uma única variável está sendo testada, especialmente quando o número de possibilidades é pequeno.

### 3.4.8. Operador Condicional

Considere, agora, a situação na qual você usa a instrução `if... else` e atribui à variável `min` o valor de `x` ou `y`, dependendo de qual dos dois valores é o menor. Isso é ilustrado no exemplo a seguir.

```
if ( alpha < beta )
    min = alpha;
else
    min = beta;
```

Para uma situação como essa, você pode utilizar o operador condicional, que é uma forma abreviada de expressar a mesma coisa. Esse operador atua sobre três operandos. O equivalente do fragmento de programa anterior é:

```
min = (x < y)? x: y;
```

A parte que fica à direita do sinal de igual (isto é,  $[x < y]? x : y$ ) é chamada de expressão condicional. Se a expressão de teste ( $x < y$ ) é verdadeira, então a expressão condicional assume o valor do operando que segue o ponto de interrogação. Contudo, se a expressão de teste é falsa, a expressão condicional assume o valor do operando que fica após os dois-pontos. A Figura 3.21 mostra a operação do operador condicional. Agora, nada melhor do que um exemplo para entender mais como utilizá-lo.

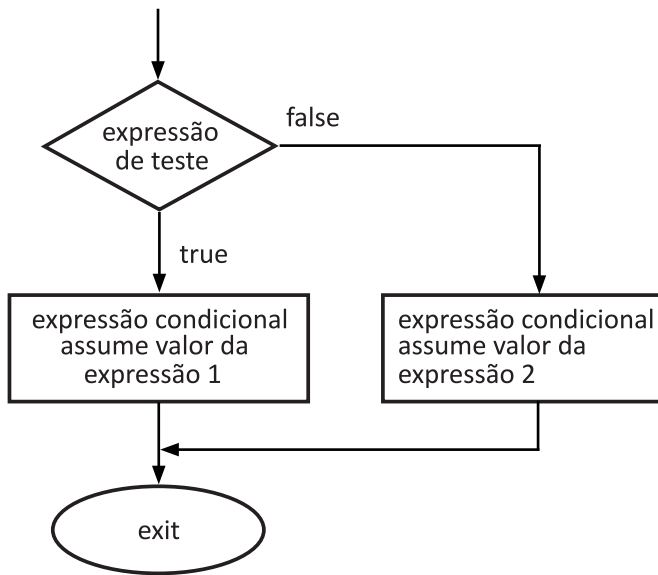


Figura 3.21 – Operação do operador condicional.

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar um número inteiro qualquer. Após isso, o programa irá testar se o número digitado é par ou não, dividindo  $n$  por 2 e comparando-o com 0 e apresentando a resposta par ou ímpar. Para a solução desse problema, você deve fazer uso do operador condicional em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.16.

■ Observe que, embora o uso do operador condicional torne o código mais conciso, sua utilização não é obrigatória. Seu uso é, na realidade, opcional. Usar um `if...else` apenas requer algumas linhas a mais (e pode tornar o código mais compreensível).

```
1. #include <iostream>
2. // #include <conio.h>
3. using namespace std;
```

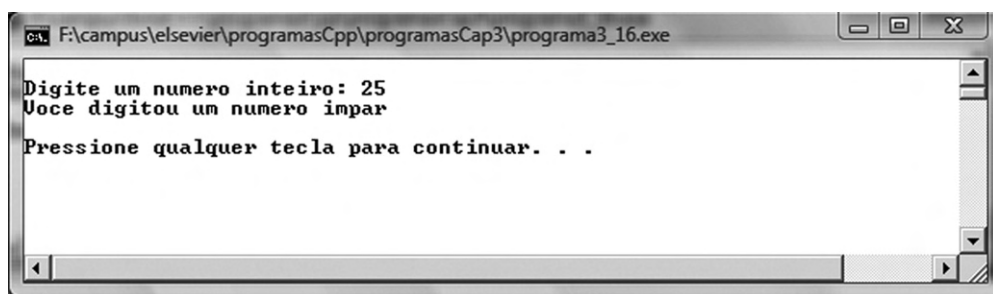
```

4. // Programa para ilustrar o uso do operador condicional
5. int main()
6. {
7.     int n;
8.     cout << "\nDigite um numero inteiro: ";
9.     cin >> n; // entrada do usuario
10.    cout << "Voce digitou um numero " << ( n % 2 == 0? "par":
        "impar") << endl << endl;
11.    system("PAUSE");
12.    return 0;
13. }

```

**Listagem 3.16**

Executando o programa da Listagem 3.16 e digitando, por exemplo, 25, o programa determina se  $n$  é par ou ímpar, dividindo-o por 2 e comparando o resultado com 0. Em seguida, o programa exibe a resposta como ilustrado na Figura 3.22.



**Figura 3.22** – Saída do programa da Listagem 3.16.

## 3.5. OPERADORES LÓGICOS

Até agora, vimos duas famílias de operadores: primeiro, os operadores *aritméticos* (+, −, \*, / e %) e, segundo, os operadores *relacionais* (<, >, <=, >=, == e !=). Em seguida, veremos a família de operadores *lógicos*. Esse tipo de operador permitirá combinar, logicamente, valores booleanos (verdadeiro/falso).

### 3.5.1. Operador Lógico E (AND)

Os operadores lógicos combinam expressões booleanas em C++. Para compreender melhor, vamos explorar alguns exemplos que usam operadores lógicos.

Primeiramente, vamos considerar o operador E ou AND, representado por `&&`. A expressão de teste será verdade apenas se os operandos forem verdadeiros. Assim, por exemplo, se você deseja checar se ambos os valores digitados pelo usuário são diferentes de 0, deve escrever:

```
if (( x!= 0) && ( y!= 0))  
    cout << "x e y são diferentes de 0 (zero)";
```

Se tanto o valor de *x* quanto o de *y* forem iguais a 0, uma mensagem será exibida na tela. Então, vamos explorar isso em um exemplo?

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar dois números inteiros. Após isso, o programa irá testar se ambos os números digitados são múltiplos de 3, exibindo uma mensagem de acordo com o resultado do teste. Para a solução desse problema, você deve fazer uso do operador lógico E (`&&`) em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.17.

```
1. #include <iostream>  
2. using namespace std;  
3. // Programa para ilustrar o uso do operador logico &&  
4. int main()  
5. {  
6.     int n1, n2;  
7.     cout << "\nDigite um numero inteiro n1 = ";  
8.     cin >> n1; // entrada do usuario  
9.     cout << "\nDigite um outro inteiro n2 = ";  
10.    cin >> n2; // entrada do usuario  
11.    if ( n1 % 3 == 0 && n2 % 3 == 0)  
12.        cout << "\nn1 e n2 sao multiplos de 3\n" << endl;  
13.    else  
14.        cout << "\nn1 e n2 nao sao multiplos de 3\n" << endl;  
15.    system("PAUSE");  
16.    Return 0;  
17. }
```

**Listagem 3.17**

Observe que o propósito do operador lógico E (*AND*) `&&` é juntar as duas expressões relacionais para obter esse resultado. Executando o programa, você obtém a saída mostrada na Figura 3.23.

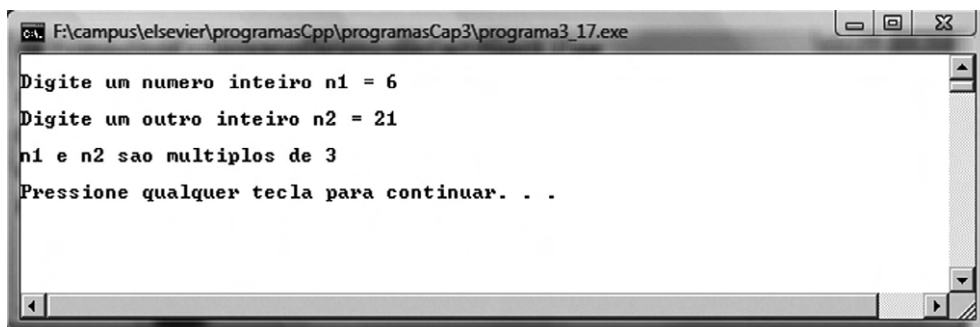


Figura 3.23 – Saída do programa da Listagem 3.17.

■ É importante observar que os operadores relacionais têm precedência maior do que os operadores lógicos. Assim, não é necessário parênteses ao redor de expressões relacionais ( $n1 \% 3 == 0$ ) ou ( $n2 \% 3 == 0$ ), como ocorre na linha 11, ( $n1 \% 3 == 0 \&\& n2 \% 3 == 0$ ), da Listagem 3.17.

A Tabela 3.2 apresenta os três operadores lógicos em C++.

Tabela 3.2

Operador	Efeito
&&	E lógico
	OU lógico
!	NÃO lógico

### 3.5.2. Operador Lógico OU (OR)

Agora, vamos considerar o operador OU ou OR, representado por `||`. A expressão de teste será verdade se pelo menos um dos operando for verdadeiro. Assim, por exemplo, se você deseja checar se algum dos valores digitados pelo usuário é igual a 0, deve escrever:

```
if (( x == 0) || ( y == 0))
    cout << "você digitou um 0 (zero)";
```

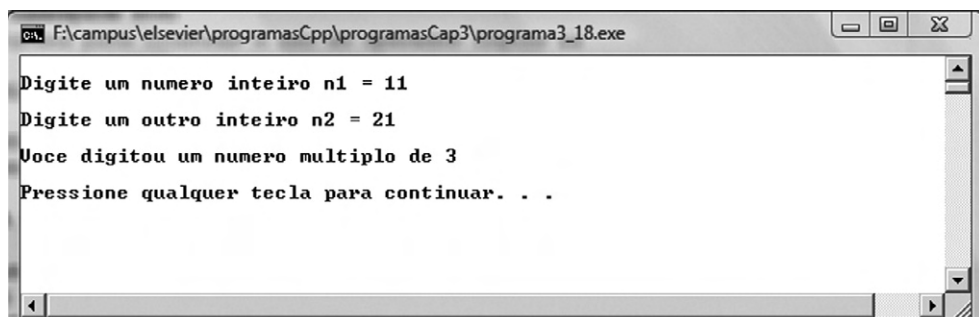
Se qualquer um dos valores digitados for igual a 0 (zero), uma mensagem é exibida na tela. Então, vamos explorar isso em um exemplo?

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar dois números inteiros. Após isso, o programa irá testar se algum desses números é múltiplo de 3, exibindo uma mensagem de acordo. Para a solução desse problema, você deve fazer uso do operador lógico OR (`||`) em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.18.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar o uso do operador logico | |
4. int main()
5. {
6.     int n1, n2;
7.     cout << "\nDigite um numero inteiro n1 = ";
8.     cin >> n1; // entrada do usuario
9.     cout << "\nDigite um outro inteiro n2 = ";
10.    cin >> n2; // entrada do usuario
11.    if ( n1 % 3 == 0 || n2 % 3 == 0)
12.        cout << "\nVoce digitou um numero multiplo de 3\n" << endl;
13.    else
14.        cout << "\nOs numeros digitados nao sao multiplos de 3\n"
            << endl;
15.    system("PAUSE");
16.    Return 0;
17. }
```

**Listagem 3.18**

O operador lógico OU (OU), representado por `||`, da linha 11 avalia duas expressões relacionais, verificando se algum dos valores digitados `n1` ou `n2` é múltiplo de 3. Se, por exemplo, você digitar os números 11 e 21, o programa exibirá a mensagem Voce digitou um numero multiplo de 3, conforme mostrado na Figura 3.24.

**Figura 3.24** – Saída do programa da Listagem 3.18.

### 3.5.3. Operador Lógico NÃO (NOT)

O operador NÃO ou NOT é representado por `!`. Esse operador requer apenas um operando unário. Utilizando esse operador obtém-se como resultado o valor lógico inverso do seu operando. Isto é, se alguma coisa é verdadeira, ele a torna falsa e vice-versa. Por exemplo, se  $(x == 1)$  é verdadeiro, então  $!(x == 1)$  é falso.



### 3.6. EXPRESSÕES LÓGICAS

Na solução de problemas, é comum você se deparar com situações nas quais precisa utilizar uma combinação de operadores lógicos e relacionais para poder formular uma condição mais complexa. Você pode, por exemplo, ter expressões como:

```
x < 0 | | x > 1000
(x!= 0 | | x! = 1) && (y!= 0 && z > 0)
```

**Praticando um Exemplo.** Escreva um programa em C++ que solicite ao usuário digitar dois números do tipo inteiro. Após isso, o programa irá testar se esses números são múltiplos de 2 e também de 3, exibindo uma mensagem se isso ocorrer. Para a solução desse problema, você deve fazer uso combinado de operadores relacionais e de uma composição de operadores lógicos AND (&&) em seu programa. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.19.

As expressões lógicas, também denominadas expressões booleanas, compreendem expressões que fazem uso dos operadores lógicos, geralmente acompanhadas dos operadores relacionais. Para entender mais, vamos explorar um exemplo.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar o uso de expressões lógicas
4. int main()
5. {
6.     unsigned int n1, n2;
7.     cout << "\nDigite um numero inteiro n1 = ";
8.     cin >> n1; // entrada do usuario
9.     cout << "\nDigite um outro inteiro n2 = ";
10.    cin >> n2; // entrada do usuario
11.    if (( n1 % 2 == 0 && n2 % 2 == 0) && ( n1 % 3 == 0 && n2 %
12.        3 == 0))
13.        cout << "\nVoce digitou numeros multiplos de 2 e 3 " <<
14.            endl << endl;
15.    else
16.        cout << "\nOs numeros digitados nao sao multiplos de 2 e
17.            3" << endl << endl;
18.    system("PAUSE");
19.    Return 0;
20. }
```

Listagem 3.19

É possível fazer uma composição de vários operadores lógicos combinados com operadores relacionais, como ocorre na linha 11 da Listagem 3.19. Execute o programa e digite, por exemplo, os números 30 e 72. O programa exibirá a mensagem *Voce digitou numeros multiplos de 2 e 3*, conforme mostrado na Figura 3.25.

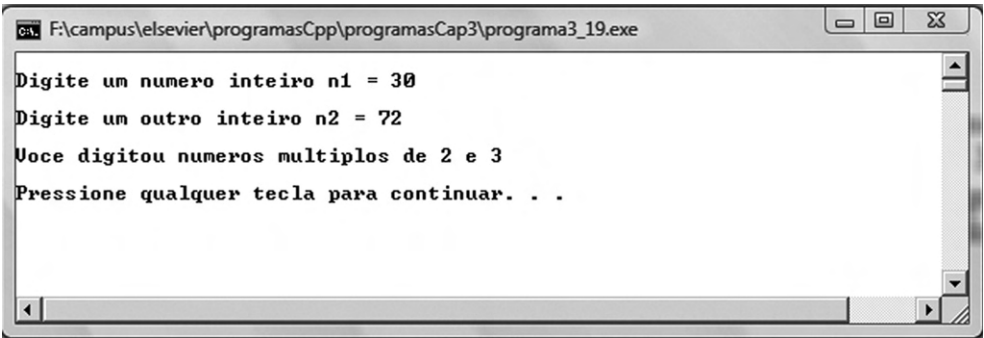


Figura 3.25 – Saída do programa da Listagem 3.19.

■ É importante ressaltar que você deve ter muita atenção quando for utilizar expressões lógicas que resultem sempre em verdadeiro ou sempre em falso. Considere, por exemplo, a expressão  $(i > 100 \ \&\& \ i < 0)$ , que sempre tem valor falso, pois a variável *i* não pode ser maior do que 100 e menor do que 0 simultaneamente.

### 3.7. PRECEDÊNCIA DE OPERADORES

A Tabela 3.3, que sumariza a precedência dos operadores, está apresentada em ordem decrescente, ou seja, os operadores da parte superior da tabela têm precedência maior do que aqueles que se encontram abaixo dele. Por exemplo, o operador relacional `<=` (menor ou igual que) tem precedência maior do que o operador lógico `&&` (E lógico).

Tabela 3.3

Tipo de Operador	Operadores
Unário	!, ++, --
Aritmético	*, /, %, +, -
Relacional	<, >, <=, >=, ==, !=
Lógico	&&,
Condicional	?:
Atribuição ( <i>assignment</i> )	=, +=, -=, *=, /=, %=

## 3.8. MAIS MECANISMOS DE CONTROLE

### 3.8.1. Break

A instrução *break* causa uma saída do processamento de um *loop* da mesma forma que o faz em uma instrução *switch* (conforme já visto anteriormente). Em tal situação, a instrução que é executada após encontrar um *break* é a instrução imediatamente seguinte ao loop. Isso é ilustrado na Figura 3.26.

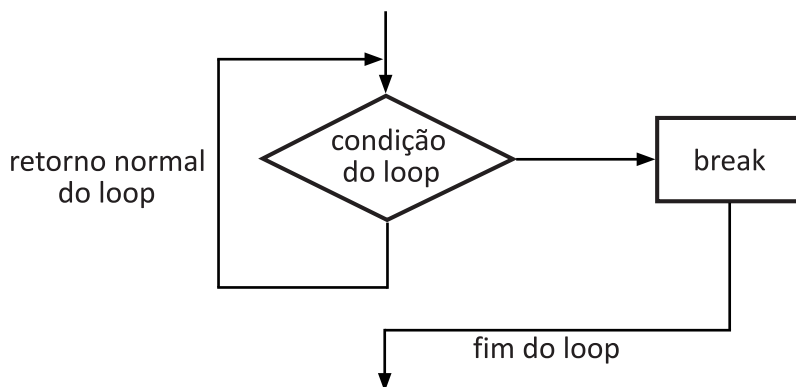


Figura 3.26 – Operação da instrução *break*.

### 3.8.2. Continue

Você viu que a instrução *break* faz com que um loop seja abandonado. Porém, há situações em que você pode precisar retornar ao início do loop devido a algum fato inesperado que ocorreu. Para tais casos, podemos usar a instrução *continue*. A Figura 3.27 ilustra essa operação. Para entender mais, veja o exemplo seguinte, que usa *continue* juntamente com operadores lógicos e relacionais.

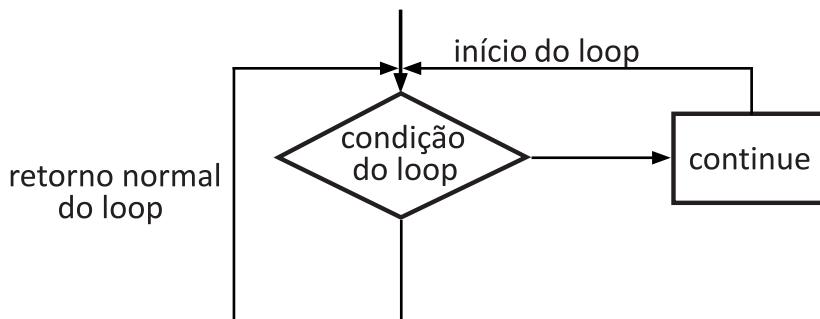


Figura 3.27 – Operação da instrução *continue*.

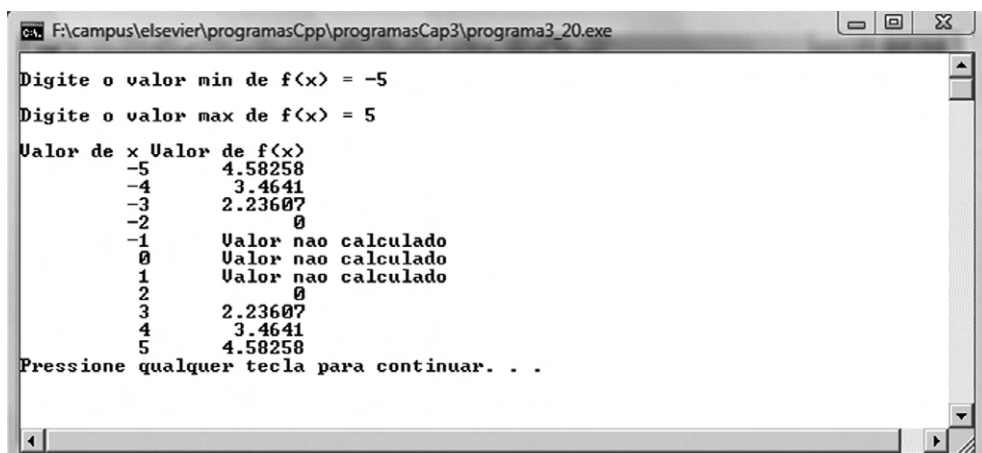
**Praticando um Exemplo.** Escreva um programa em C++ que calcule  $y = f(x)$  para valores de  $x$  na faixa mín a máx. A função  $f(x) = \sqrt{x^2 - 4}$  e o usuário é solicitado a entrar com dois números do tipo inteiro mín (valor mínimo) e máx (valor máximo) que serão computados para função  $f(x)$ . Após isso, o programa irá calcular os valores de  $f(x)$ , exceto para valores na faixa de  $-1$  a  $1$ , que resultam em valores complexos. Portanto, o programa evita calcular esses valores. Para a solução desse problema, você deve fazer uso combinado de operadores relacionais e lógicos, além da instrução *continue*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 3.20.

```
1. #include <iostream>
2. #include <iomanip>
3. #include <math.h>
4. using namespace std;
5. // Programa para ilustrar o uso do continue
6. int main()
7. {
8.     int min, max;
9.     double x, y;
10.    cout << "\nDigite o valor min de f(x) = ";
11.    cin >> min; // entrada do usuario
12.    cout << "\nDigite o valor max de f(x) = ";
13.    cin >> max; // entrada do usuario
14.    cout << "\nValor de x " << setw(11) << "Valor de f(x)" << endl;
15.    for (int i = min; i <= max; i++) {
16.        if (i > -2 && i < 2) {
17.            cout << setw(11) << i << setw(25) << "Valor nao calculado" << endl;
18.            continue;
19.        }
20.        x = (double)i;
21.        y = sqrt(x*x - 4);
22.        cout << setw(11) << x << setw(13) << y << endl;
23.    }
24.    system("PAUSE");
25.    Return 0;
26. }
```

**Listagem 3.20**

O programa da Listagem 3.20 faz uso dos operadores lógicos combinados com operadores relacionais, além de também utilizar a instrução *continue* na linha 18.

Execute o programa e digite, por exemplo, os valores  $-5$  para mín e  $5$  para máx e, então, o programa exibirá a saída mostrada na Figura 3.28.



```

F:\campus\elsevier\programasCpp\programasCap3\programa3_20.exe
Digite o valor min de f(x) = -5
Digite o valor max de f(x) = 5
Valor de x  Valor de f(x)
-5          4.58258
-4          3.4641
-3          2.23607
-2          0
-1          Valor nao calculado
0           Valor nao calculado
1           Valor nao calculado
2           0
3           2.23607
4           3.4641
5           4.58258
Pressione qualquer tecla para continuar. . .
  
```

Figura 3.28 – Saída do programa da Listagem 3.20.

## RESUMO

Neste capítulo, você verificou que os programas não executam as instruções em uma ordem estritamente sequencial e que, por causa disso, os programadores fazem uso de mecanismos de controle e decisão para definir o comportamento de um programa. Aqui você teve a oportunidade de conhecer estruturas de controle e decisão, explorando o uso dos operadores lógicos e relacionais, e entender a relação de precedência que eles possuem. Novos conceitos foram apresentados e diversos exemplos foram realizados, ilustrando o comportamento desses mecanismos de controle e decisão. No próximo capítulo, você irá estudar e explorar estruturas e mecanismos de acesso a membros de uma estrutura e seu relacionamento com classes, que constituem a base do paradigma da programação orientada a objetos.

## QUESTÕES

1. O que são laços (ou loops)? Apresente exemplos onde você poderia necessitar usar laços.
2. Qual a diferença entre os laços *while* e *for*? Apresente exemplos onde você poderia necessitar desses comandos de controle.
3. Em que situação você utiliza o parâmetro *default* no comando de controle *switch*? Use um exemplo para ilustrar sua resposta.
4. Qual a ordem de precedência dos operadores suportados pela linguagem C++?

## EXERCÍCIOS

1. Faça uma pesquisa visando responder à seguinte questão: há operadores que tornam o código mais conciso e eficiente? Em que situações seu uso é interessante?
2. Escreva um programa em C++ que gere a saída a seguir.
 

```
*****
*****
*****
*****
*****
*****
```
3. Escreva um programa em C++ que gere a saída a seguir.
 

```
*      *****
**     *****
***    *****
****   *****
*****  *****
*****  *
```
4. Escreva um programa em C++ que gere a saída a seguir.
 

```
*
* *
* * *
* * * *
* * *
* * *
* *
*
```
5. Escreva um programa em C++ que gere a saída a seguir.
  - a) 2008 100
  - b) 2009 400
  - c) 2010 700
  - d) 2011 1000
6. Escreva um programa em C++ que permita que o usuário entre com um valor de temperatura em Fahrenheit e faça a conversão desse valor para Celsius e mostre o resultado um número qualquer de vezes. Use a fórmula a seguir, considerando ftemp e ctemp como temperatura em Fahrenheit e Celsius, respectivamente. Permita ao usuário digitar um valor de sentinela (como, por exemplo, -1) para encerrar a conversão.
 
$$ctemp = (ftemp - 32) * 5 / 9;$$
7. Modifique o programa do Exercício 5 de modo que ele solicite ao usuário para entrar com uma quantia em dólares e faça a conversão desse valor para reais. Permita ao usuário digitar um valor de sentinela (como, por exemplo, -1) para encerrar a conversão.