CHAPTER 5
STATEMENTS

CONTENTS

Like most languages, C++ provides statements for conditional execution, loops that repeatedly execute the same body of code, and jump statements that interrupt the flow of control. This chapter looks in detail at the statements supported by C++.

Statements are executed sequentially. Except for the simplest programs, sequential execution is inadequate. Therefore, C++ also defines a set of ***flow-of-control*** statements that allow more complicated execution paths.

5.1 Simple Statements

Most statements in C++ end with a semicolon. An expression, such as `ival + 5`, becomes an **expression statement** when it is followed by a semicolon. Expression statements cause the expression to be evaluated and its result discarded:

```
ival + 5;        // rather useless expression statement
cout << ival;    // useful expression statement
```

The first statement is pretty useless: The addition is done but the result is not used. More commonly, an expression statement contains an expression that has a side effect—such as assigning a new value to a variable, or printing a result—when it is evaluated.

Null Statements

The simplest statement is the empty statement, also known as a **null statement**. A null statement is a single semicolon:

```
;    // null statement
```

A null statement is useful where the language requires a statement but the program's logic does not. Such usage is most common when a loop's work can be done within its condition. For example, we might want to read an input stream, ignoring everything we read until we encounter a particular value:

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ; // null statement
```

This condition reads a value from the standard input and implicitly tests `cin` to see whether the read was successful. Assuming the read succeeded, the second part of the condition tests whether the value we read is equal to the value in `sought`. If we found the value we want, the `while` loop is exited. Otherwise, the condition is evaluated again, which reads another value from `cin`.

Best Practices

Null statements should be commented. That way anyone reading the code can see that the statement was omitted intentionally.

Beware of Missing or Extraneous Semicolons

Because a null statement is a statement, it is legal anywhere a statement is expected. For this reason, semicolons that might appear illegal are often nothing more than null statements. The following fragment contains two statements—the expression statement and the null statement:

```
ival = v1 + v2;; // ok: second semicolon is a superfluous null statement
```

Although an unnecessary null statement is often harmless, an extra semicolon following the condition in a `while` or `if` can drastically alter the programmer's intent. For example, the following code will loop indefinitely:

```
// disaster: extra semicolon: loop body is this null statement
while (iter != svec.end()) ; // the while body is the empty statement
    ++iter;      // increment is not part of the loop
```

Contrary to the indentation, the increment is not part of the loop. The loop body is the null statement formed by the semicolon that follows the condition.

WARNING

Extraneous null statements are not always harmless.

Compound Statements (Blocks)

A **compound statement**, usually referred to as a **block**, is a (possibly empty) sequence of statements and declarations surrounded by a pair of curly braces. A block is a scope (§ 2.2.4, p. 48). Names introduced inside a block are accessible only in that block and in blocks nested inside that block. Names are visible from where they are defined until the end of the (immediately) enclosing block.

Compound statements are used when the language requires a single statement but the logic of our program needs more than one. For example, the body of a `while` or `for` loop must be a single statement, yet we often need to execute more than one statement in the body of a loop. We do so by enclosing the statements in curly braces, thus turning the sequence of statements into a block.

As one example, recall the `while` loop in the program in § 1.4.1 (p. 11):

```
while (val <= 10) {
    sum += val;  // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

The logic of our program needed two statements but a `while` loop may contain only one statement. By enclosing these statements in curly braces, we made them into a single (compound) statement.

Note

A block is *not* terminated by a semicolon.

We also can define an empty block by writing a pair of curlies with no statements. An empty block is equivalent to a null statement:

```
while (cin >> s && s != sought)
    { } // empty block
```

Exercises Section 5.1

Exercise 5.1: What is a null statement? When might you use a null statement?

Exercise 5.2: What is a block? When might you might use a block?

Exercise 5.3: Use the comma operator (§ 4.10, p. 157) to rewrite the `while` loop from § 1.4.1 (p. 11) so that it no longer requires a block. Explain whether this rewrite improves or diminishes the readability of this code.

5.2 Statement Scope

We can define variables inside the control structure of the `if`, `switch`, `while`, and `for` statements. Variables defined in the control structure are visible only within that statement and are out of scope after the statement ends:

```
while (int i = get_num()) // i is created and initialized on each iteration
    cout << i << endl;
i = 0;   // error: i is not accessible outside the loop
```

If we need access to the control variable, then that variable must be defined outside the statement:

```
// find the first negative element
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // we know that all elements in v are greater than or equal to zero
```

The value of an object defined in a control structure is used by that structure. Therefore, such variables must be initialized.

Exercises Section 5.2

Exercise 5.4: Explain each of the following examples, and correct any problems you detect.

```
(a) while (string::iterator iter != s.end()) { /* . . . */ }
(b) while (bool status = find(word)) { /* . . . */ }
    if (!status) { /* . . . */ }
```

5.3 Conditional Statements

C++ provides two statements that allow for conditional execution. The `if` statement determines the flow of control based on a condition. The `switch` statement evaluates an integral expression and chooses one of several execution paths based on the expression's value.

5.3.1 The `if` Statement

An **`if` statement** conditionally executes another statement based on whether a specified condition is true. There are two forms of the `if`: one with an `else` branch and one without. The syntactic form of the simple `if` is

```
if (condition)
       statement
```

An **`if else` statement** has the form

```
if (condition)
       statement
else
       statement2
```

In both versions, *condition* must be enclosed in parentheses. *condition* can be an expression or an initialized variable declaration (§ 5.2, p. 174). The expression or variable must have a type that is convertible (§ 4.11, p. 159) to `bool`. As usual, either or both *statement* and *statement2* can be a block.

If *condition* is `true`, then *statement* is executed. After *statement* completes, execution continues with the statement following the `if`.

If *condition* is `false`, *statement* is skipped. In a simple `if`, execution continues with the statement following the `if`. In an `if else`, *statement2* is executed.

Using an `if else` Statement

To illustrate an `if` statement, we'll calculate a letter grade from a numeric grade. We'll assume that the numeric grades range from zero to 100 inclusive. A grade of 100 gets an "A++," grades below 60 get an "F," and the others range in clumps of ten: grades from 60 to 69 inclusive get a "D," 70 to 79 a "C," and so on. We'll use a `vector` to hold the possible letter grades:

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

To solve this problem, we can use an `if else` statement to execute different actions for failing and passing grades:

```
// if grade is less than 60 it's an F, otherwise compute a subscript
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

Depending on the value of `grade`, we execute the statement after the `if` or the one after the `else`. In the `else`, we compute a subscript from a grade by reducing the grade to account for the larger range of failing grades. Then we use integer division (§ 4.2, p. 141), which truncates the remainder, to calculate the appropriate `scores` index.

Nested `if` Statements

To make our program more interesting, we'll add a plus or minus to passing grades. We'll give a plus to grades ending in 8 or 9, and a minus to those ending in 0, 1, or 2:

```
if (grade % 10 > 7)
    lettergrade += '+';      // grades ending in 8 or 9 get a +
else if (grade % 10 < 3)
    lettergrade += '-';      // those ending in 0, 1, or 2 get a -
```

Here we use the modulus operator (§ 4.2, p. 141) to get the remainder and decide based on the remainder whether to add plus or minus.

We next will incorporate the code that adds a plus or minus to the code that fetches the letter grade from scores:

```
// if failing grade, no need to check for a plus or minus
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10];   // fetch the letter grade
    if (grade != 100)   // add plus or minus only if not already an A++
        if (grade % 10 > 7)
            lettergrade += '+';     // grades ending in 8 or 9 get a +
        else if (grade % 10 < 3)
            lettergrade += '-';     // grades ending in 0, 1, or 2 get a -
}
```

Note that we use a block to enclose the two statements that follow the first `else`. If the `grade` is `60` or more, we have two actions that we need to do: Fetch the letter grade from `scores`, and conditionally set the plus or minus.

Watch Your Braces

It is a common mistake to forget the curly braces when multiple statements must be executed as a block. In the following example, contrary to the indentation, the code to add a plus or minus happens unconditionally:

```
if (grade < 60)
    lettergrade = scores[0];
else  // WRONG: missing curly
    lettergrade = scores[(grade - 50)/10];
    // despite appearances, without the curly brace, this code is always executed
    // failing grades will incorrectly get a - or a +
    if (grade != 100)
        if (grade % 10 > 7)
            lettergrade += '+';     // grades ending in 8 or 9 get a +
        else if (grade % 10 < 3)
            lettergrade += '-';     // grades ending in 0, 1, or 2 get a -
```

Uncovering this error may be very difficult because the program looks correct.

To avoid such problems, some coding styles recommend always using braces after an `if` or an `else` (and also around the bodies of `while` and `for` statements).

Doing so avoids any possible confusion. It also means that the braces are already in place if later modifications of the code require adding statements.

Best Practices

Many editors and development environments have tools to automatically indent source code to match its structure. It is a good idea to use such tools if they are available.

Dangling `else`

When we nest an `if` inside another `if`, it is possible that there will be more `if` branches than `else` branches. Indeed, our grading program has four `if`s and two `else`s. The question arises: How do we know to which `if` a given `else` belongs?

This problem, usually referred to as a **dangling `else`**, is common to many programming languages that have both `if` and `if else` statements. Different languages solve this problem in different ways. In C++ the ambiguity is resolved by specifying that each `else` is matched with the closest preceding unmatched `if`.

Programmers sometimes get into trouble when they write code that contains more `if` than `else` branches. To illustrate the problem, we'll rewrite the innermost `if else` that adds a plus or minus using a different set of conditions:

```
// WRONG: execution does NOT match indentation; the else goes with the inner if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
else
    lettergrade += '-'; // grades ending in 3, 4, 5, 6, or 7 get a minus!
```

The indentation in our code indicates that we intend the `else` to go with the outer `if`—we intend for the `else` branch to be executed when the `grade` ends in a digit less than `3`. However, despite our intentions, and contrary to the indentation, the `else` branch is part of the inner `if`. This code adds a `'-'` to grades ending in `3` to `7` inclusive! Properly indented to match the actual execution, what we wrote is:

```
// indentation matches the execution path, not the programmer's intent
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
    else
        lettergrade += '-';  // grades ending in 3, 4, 5, 6, or 7 get a minus!
```

Controlling the Execution Path with Braces

We can make the `else` part of the outer `if` by enclosing the inner `if` in a block:

```
// add a plus for grades that end in 8 or 9 and a minus for those ending in 0, 1, or 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
} else                      // curlies force the else to go with the outer if
    lettergrade += '-'; // grades ending in 0, 1, or 2 will get a minus
```

Statements do not span block boundaries, so the inner `if` ends at the close curly before the `else`. The `else` cannot be part of the inner `if`. Now, the nearest unmatched `if` is the outer `if`, which is what we intended all along.

Exercises Section 5.3.1

Exercise 5.5: Using an `if–else` statement, write your own version of the program to generate the letter grade from a numeric grade.

Exercise 5.6: Rewrite your grading program to use the conditional operator (§ 4.7, p. 151) in place of the `if–else` statement.

Exercise 5.7: Correct the errors in each of the following code fragments:

```
(a) if (ival1 != ival2)
        ival1 = ival2
    else ival1 = ival2 = 0;
(b) if (ival < minval)
        minval = ival;
        occurs = 1;
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
```

Exercise 5.8: What is a "dangling `else`"? How are `else` clauses resolved in C++?

5.3.2 The `switch` Statement

A **`switch` statement** provides a convenient way of selecting among a (possibly large) number of fixed alternatives. As one example, suppose that we want to count how often each of the five vowels appears in some segment of text. Our program logic is as follows:

- Read every character in the input.
- Compare each character to the set of vowels.
- If the character matches one of the vowels, add 1 to that vowel's count.
- Display the results.

For example, when we run the program on the text of this chapter, the output is

```
Number of vowel a:  3195
Number of vowel e:  6230
Number of vowel i:  3102
Number of vowel o:  3289
Number of vowel u:  1033
```

We can solve our problem most directly using a `switch` statement:

```
// initialize counters for each vowel
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

A `switch` statement executes by evaluating the parenthesized expression that follows the keyword `switch`. That expression may be an initialized variable declaration (§ 5.2, p. 174). The expression is converted to integral type. The result of the expression is compared with the value associated with each `case`.

If the expression matches the value of a `case` label, execution begins with the first statement following that label. Execution continues normally from that statement through the end of the `switch` or until a `break` statement.

We'll look at `break` statements in detail in § 5.5.1 (p. 190), but, briefly, a `break` interrupts the current control flow. In this case, the `break` transfers control out of the `switch`. In this program, the `switch` is the only statement in the body of a `while`. Breaking out of this `switch` returns control to the enclosing `while`. Because there are no other statements in that `while`, execution continues at the condition in the `while`.

If no match is found, execution falls through to the first statement following the `switch`. As we already know, in this example, exiting the `switch` returns control to the condition in the `while`.

The `case` keyword and its associated value together are known as the **`case` label**. `case` labels must be integral constant expressions (§ 2.4.4, p. 65):

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: //  error: noninteger as case label
    case ival: //  error: nonconstant as case label
    // ...
```

It is an error for any two `case` labels to have the same value. There is also a special-case label, `default`, which we cover on page 181.

Control Flow within a `switch`

It is important to understand that execution flows across `case` labels. After a `case` label is matched, execution starts at that label and continues across all the remaining `case`s or until the program explicitly interrupts it. To avoid executing code for subsequent `case`s, we must explicitly tell the compiler to stop execution. Under most conditions, the last statement before the next `case` label is `break`.

However, there are situations where the default `switch` behavior is exactly what is needed. Each `case` label can have only a single value, but sometimes we have two or more values that share a common set of actions. In such instances, we omit a `break` statement, allowing the program to *fall through* multiple `case` labels.

For example, we might want to count only the total number of vowels:

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    //  any occurrence of a, e, i, o, or u increments vowelCnt
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
       ++vowelCnt;
       break;
}
```

Here we stacked several `case` labels together with no intervening `break`. The same code will be executed whenever `ch` is a vowel.

Because C++ programs are free-form, `case` labels need not appear on a new line. We can emphasize that the `case`s represent a range of values by listing them all on a single line:

```
switch (ch)
{
    //  alternative legal syntax
    case 'a': case 'e': case 'i': case 'o': case 'u':
       ++vowelCnt;
       break;
}
```

Best Practices

Omitting a `break` at the end of a `case` happens rarely. If you do omit a `break`, include a comment explaining the logic.

Forgetting a `break` Is a Common Source of Bugs

It is a common misconception to think that only the statements associated with the matched `case` label are executed. For example, here is an *incorrect* implementation of our vowel-counting `switch` statement:

```
// warning: deliberately incorrect!
switch (ch) {
    case 'a':
        ++aCnt;    // oops: should have a break statement
    case 'e':
        ++eCnt;    // oops: should have a break statement
    case 'i':
        ++iCnt;    // oops: should have a break statement
    case 'o':
        ++oCnt;    // oops: should have a break statement
    case 'u':
        ++uCnt;
}
```

To understand what happens, assume that the value of `ch` is `'e'`. Execution jumps to the code following the `case 'e'` label, which increments `eCnt`. Execution *continues* across the `case` labels, incrementing `iCnt`, `oCnt`, and `uCnt` as well.

Best Practices

Although it is not necessary to include a `break` after the last label of a `switch`, the safest course is to provide one. That way, if an additional `case` is added later, the `break` is already in place.

The `default` Label

The statements following the **`default` label** are executed when no `case` label matches the value of the `switch` expression. For example, we might add a counter to track how many nonvowels we read. We'll increment this counter, which we'll name `otherCnt`, in the `default` case:

```
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            ++vowelCnt;
            break;
        default:
            ++otherCnt;
            break;
    }
}
```

In this version, if ch is not a vowel, execution will start at the default label and we'll increment otherCnt.

Best Practices

It can be useful to define a default label even if there is no work for the default case. Defining an empty default section indicates to subsequent readers that the case was considered.

A label may not stand alone; it must precede a statement or another case label. If a switch ends with a default case that has no work to do, then the default label must be followed by a null statement or an empty block.

Variable Definitions inside the Body of a `switch`

As we've seen, execution in a switch can jump across case labels. When execution jumps to a particular case, any code that occurred inside the switch before that label is ignored. The fact that code is bypassed raises an interesting question: What happens if the code that is skipped includes a variable definition?

The answer is that it is illegal to jump from a place where a variable with an initializer is out of scope to a place where that variable is in scope:

```
case true:
    // this switch statement is illegal because these initializations might be bypassed
    string file_name; // error: control bypasses an implicitly initialized variable
    int ival = 0;     // error: control bypasses an explicitly initialized variable
    int jval;         // ok: because jval is not initialized
    break;
case false:
    // ok: jval is in scope but is uninitialized
    jval = next_num(); // ok: assign a value to jval
    if (file_name.empty()) // file_name is in scope but wasn't initialized
        // ...
```

If this code were legal, then any time control jumped to the false case, it would bypass the initialization of file_name and ival. Those variables would be in scope. Code following false could use those variables. However, these variables would not have been initialized. As a result, the language does not allow us to jump over an initialization if the initialized variable is in scope at the point to which control transfers.

If we need to define and initialize a variable for a particular case, we can do so by defining the variable inside a block, thereby ensuring that the variable is out of scope at the point of any subsequent label.

```
case true:
    {
       // ok: declaration statement within a statement block
       string file_name = get_file_name();
       // ...
    }
    break;
case false:
       if (file_name.empty())   // error: file_name is not in scope
```

EXERCISES SECTION 5.3.2

Exercise 5.9: Write a program using a series of `if` statements to count the number of vowels in text read from `cin`.

Exercise 5.10: There is one problem with our vowel-counting program as we've implemented it: It doesn't count capital letters as vowels. Write a program that counts both lower- and uppercase letters as the appropriate vowel—that is, your program should count both `'a'` and `'A'` as part of `aCnt`, and so forth.

Exercise 5.11: Modify our vowel-counting program so that it also counts the number of blank spaces, tabs, and newlines read.

Exercise 5.12: Modify our vowel-counting program so that it counts the number of occurrences of the following two-character sequences: `ff`, `fl`, and `fi`.

Exercise 5.13: Each of the programs in the highlighted text on page 184 contains a common programming error. Identify and correct each error.

5.4 Iterative Statements

Iterative statements, commonly called loops, provide for repeated execution until a condition is true. The `while` and `for` statements test the condition before executing the body. The `do while` executes the body and then tests its condition.

5.4.1 The `while` Statement

A **`while` statement** repeatedly executes a target statement as long as a condition is true. Its syntactic form is

```
while (condition)
         statement
```

In a `while`, *statement* (which is often a block) is executed as long as *condition* evaluates as `true`. *condition* may not be empty. If the first evaluation of *condition* yields `false`, *statement* is not executed.

The condition can be an expression or an initialized variable declaration (§ 5.2, p. 174). Ordinarily, the condition itself or the loop body must do something to change the value of the expression. Otherwise, the loop might never terminate.

Note

Variables defined in a `while` condition or `while` body are created and destroyed on each iteration.

Using a `while` Loop

A `while` loop is generally used when we want to iterate indefinitely, such as when we read input. A `while` is also useful when we want access to the value of the loop control variable after the loop finishes. For example:

CODE FOR EXERCISE 5.13

```
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
    char ch = next_text();
    switch (ch) {
        case ’a’: aCnt++;
        case ’e’: eCnt++;
        default: iouCnt++;
    }

(b) unsigned index = some_value();
    switch (index) {
        case 1:
            int ix = get_value();
            ivec[ ix ] = index;
            break;
        default:
            ix = ivec.size()-1;
            ivec[ ix ] = index;
    }

(c) unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }

(d) unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
```

```
vector<int> v;
int i;
// read until end-of-file or other input failure
while (cin >> i)
    v.push_back(i);
// find the first negative element
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // we know that all elements in v are greater than or equal to zero
```

The first loop reads data from the standard input. We have no idea how many times this loop will execute. The condition fails when `cin` reads invalid data, encounters some other input failure, or hits end-of-file. The second loop continues until we find a negative value. When the loop terminates, `beg` is either equal to `v.end()`, or it denotes an element in `v` whose value is less than zero. We can use the state of `beg` outside the `while` to determine further processing.

EXERCISES SECTION 5.4.1

Exercise 5.14: Write a program to read `string`s from standard input looking for duplicated words. The program should find places in the input where one word is followed immediately by itself. Keep track of the largest number of times a single repetition occurs and which word is repeated. Print the maximum number of duplicates, or else print a message saying that no word was repeated. For example, if the input is

```
how now now now brown cow cow
```

the output should indicate that the word `now` occurred three times.

5.4.2 Traditional `for` Statement

The syntactic form of the **`for` statement** is:

```
for (init-statement condition; expression)
    statement
```

The `for` and the part inside the parentheses is often referred to as the `for` header.

init-statement must be a declaration statement, an expression statement, or a null statement. Each of these statements ends with a semicolon, so the syntactic form can also be thought of as

```
for (initializer; condition; expression)
    statement
```

In general, *init-statement* is used to initialize or assign a starting value that is modified over the course of the loop. *condition* serves as the loop control. As long as *condition* evaluates as `true`, *statement* is executed. If the first evaluation of *condition* yields `false`, *statement* is not executed. *expression* usually modifies the variable(s) initialized in *init-statement* and tested in *condition*. *expression* is evaluated after each iteration of the loop. As usual, *statement* can be either a single or a compound statement.

Execution Flow in a Traditional `for` Loop

Given the following `for` loop from § 3.2.3 (p. 94):

```
// process characters in s until we run out of characters or we hit a whitespace
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
```

the order of evaluation is as follows:

1. *init-statement* is executed once at the start of the loop. In this example, `index` is defined and initialized to zero.
2. Next, *condition* is evaluated. If `index` is not equal to `s.size()` and the character at `s[index]` is not whitespace, the `for` body is executed. Otherwise, the loop terminates. If the condition is `false` on the first iteration, then the `for` body is not executed at all.
3. If the condition is `true`, the `for` body executes. In this case, the `for` body makes the character at `s[index]` uppercase.
4. Finally, *expression* is evaluated. In this example, `index` is incremented by 1.

These four steps represent the first iteration of the `for` loop. Step 1 is executed only once on entry to the loop. Steps 2, 3, and 4 are repeated until the condition evaluates as `false`—that is, when we encounter a whitespace character in `s`, or `index` is greater than `s.size()`.

Note

It is worth remembering that the visibility of any object defined within the `for` header is limited to the body of the `for` loop. Thus, in this example, `index` is inaccessible after the `for` completes.

Multiple Definitions in the `for` Header

As in any other declaration, *init-statement* can define several objects. However, *init-statement* may be only a single declaration statement. Therefore, all the variables must have the same base type (§ 2.3, p. 50). As one example, we might write a loop to duplicate the elements of a `vector` on the end as follows:

```
// remember the size of v and stop when we get to the original last element
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

In this loop we define both the index, `i`, and the loop control, `sz`, in *init-statement*.

Omitting Parts of the `for` Header

A `for` header can omit any (or all) of *init-statement*, *condition*, or *expression*.

We can use a null statement for *init-statement* when an initialization is unnecessary. For example, we might rewrite the loop that looked for the first negative number in a `vector` so that it uses a `for`:

```
auto beg = v.begin();
for ( /* null */; beg != v.end() && *beg >= 0; ++beg)
    ; // no work to do
```

Note that the semicolon is necessary to indicate the absence of *init-statement*—more precisely, the semicolon represents a null *init-statement*. In this loop, the `for` body is also empty because all the work of the loop is done inside the `for` condition and expression. The condition decides when it's time to stop looking and the expression increments the iterator.

Omitting *condition* is equivalent to writing `true` as the condition. Because the condition always evaluates as `true`, the `for` body must contain a statement that exits the loop. Otherwise the loop will execute indefinitely:

```
for (int i = 0; /* no condition */ ; ++i) {
    // process i; code inside the loop must stop the iteration!
}
```

We can also omit *expression* from the `for` header. In such loops, either the condition or the body must do something to advance the iteration. As an example, we'll rewrite the `while` loop that read input into a `vector` of `int`s:

```
vector<int> v;
for (int i; cin >> i; /* no expression */ )
   v.push_back(i);
```

In this loop there is no need for an expression because the condition changes the value of `i`. The condition tests the input stream so that the loop ends when we've read all the input or encounter an input error.

5.4.3 Range `for` Statement

The new standard introduced a simpler `for` statement that can be used to iterate through the elements of a container or other sequence. The syntactic form of the **range `for` statement** is:

```
for (declaration : expression)
    statement
```

expression must represent a sequence, such as a braced initializer list (§ 3.3.1, p. 98), an array (§ 3.5, p. 113), or an object of a type such as `vector` or `string` that has `begin` and `end` members that return iterators (§ 3.4, p. 106).

declaration defines a variable. It must be possible to convert each element of the sequence to the variable's type (§ 4.11, p. 159). The easiest way to ensure that the

Exercises Section 5.4.2

Exercise 5.15: Explain each of the following loops. Correct any problems you detect.

(a)
```
for (int ix = 0; ix != sz; ++ix)  { /* ... */ }
if (ix != sz)
     // ...
```
(b)
```
int ix;
for (ix != sz; ++ix) { /* ... */ }
```
(c)
```
for (int ix = 0; ix != sz; ++ix, ++ sz)  { /* ... */ }
```

Exercise 5.16: The `while` loop is particularly good at executing while some condition holds; for example, when we need to read values until end-of-file. The `for` loop is generally thought of as a step loop: An index steps through a range of values in a collection. Write an idiomatic use of each loop and then rewrite each using the other loop construct. If you could use only one loop, which would you choose? Why?

Exercise 5.17: Given two `vector`s of `int`s, write a program to determine whether one `vector` is a prefix of the other. For `vector`s of unequal length, compare the number of elements of the smaller `vector`. For example, given the `vector`s containing `0`, `1`, `1`, and `2` and `0`, `1`, `1`, `2`, `3`, `5`, `8`, respectively your program should return `true`.

types match is to use the `auto` type specifier (§ 2.5.2, p. 68). That way the compiler will deduce the type for us. If we want to write to the elements in the sequence, the loop variable must be a reference type.

On each iteration, the control variable is defined and initialized by the next value in the sequence, after which *statement* is executed. As usual, *statement* can be a single statement or a block. Execution ends once all the elements have been processed.

We have already seen several such loops, but for completeness, here is one that doubles the value of each element in a `vector`:

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// range variable must be a reference so we can write to the elements
for (auto &r : v)    // for each element in v
    r *= 2;          // double the value of each element in v
```

The `for` header declares the loop control variable, `r`, and associates it with `v`. We use `auto` to let the compiler infer the correct type for `r`. Because we want to change the value of the elements in `v`, we declare `r` as a reference. When we assign to `r` inside the loop, that assignment changes the element to which `r` is bound.

A range `for` is defined in terms of the equivalent traditional `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // r must be a reference so we can change the element
    r *= 2;         // double the value of each element in v
}
```

Now that we know how a range `for` works, we can understand why we said in § 3.3.2 (p. 101) that we cannot use a range `for` to add elements to a `vector` (or other container). In a range for, the value of end() is cached. If we add elements to (or remove them from) the sequence, the value of end might be invalidated (§ 3.4.1, p. 110). We'll have more to say about these matters in § 9.3.6 (p. 353).

5.4.4 The `do while` Statement

A **`do while` statement** is like a while but the condition is tested after the statement body completes. Regardless of the value of the condition, we execute the loop at least once. The syntactic form is as follows:

```
do
        statement
while (condition);
```

Note

A do while ends with a semicolon after the parenthesized condition.

In a do, *statement* is executed before *condition* is evaluated. *condition* cannot be empty. If *condition* evaluates as false, then the loop terminates; otherwise, the loop is repeated. Variables used in *condition* must be defined outside the body of the do while statement.

We can write a program that (indefinitely) does sums using a do while:

```
// repeatedly ask the user for a pair of numbers to sum
string rsp;   // used in the condition; can't be defined inside the do
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin  >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

The loop starts by prompting the user for two numbers. It then prints their sum and asks whether the user wishes to do another sum. The condition checks that the user gave a response. If not, or if the input starts with an n, the loop is exited. Otherwise the loop is repeated.

Because the condition is not evaluated until after the statement or block is executed, the do while loop does not allow variable definitions inside the condition:

```
do {
    // . . .
    mumble(foo);
} while (int foo = get_foo()); // error: declaration in a do condition
```

If we could define variables in the condition, then any use of the variable would happen *before* the variable was defined!

Exercises Section 5.4.4

Exercise 5.18: Explain each of the following loops. Correct any problems you detect.

(a)
```
do
    int v1, v2;
    cout << "Please enter two numbers to sum:" ;
    if (cin >> v1 >> v2)
        cout << "Sum is: " << v1 + v2 << endl;
while (cin);
```
(b)
```
do {
    // ...
} while (int ival = get_response());
```
(c)
```
do {
    int ival = get_response();
 } while (ival);
```

Exercise 5.19: Write a program that uses a `do while` loop to repetitively request two `strings` from the user and report which `string` is less than the other.

5.5 Jump Statements

Jump statements interrupt the flow of execution. C++ offers four jumps: `break`, `continue`, and `goto`, which we cover in this chapter, and the `return` statement, which we'll describe in § 6.3 (p. 222).

5.5.1 The `break` Statement

A **`break` statement** terminates the nearest enclosing `while`, `do while`, `for`, or `switch` statement. Execution resumes at the statement immediately following the terminated statement.

A `break` can appear only within an iteration statement or `switch` statement (including inside statements or blocks nested inside such loops). A `break` affects only the nearest enclosing loop or `switch`:

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // process up to the first blank
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
              if (*it == ' ')
                   break; // #1, leaves the for loop
              // ...
        }
        // break #1 transfers control here
        // remaining '-' processing:
        break; // #2, leaves the switch statement
        case '+':
            // ...
        } // end switch
       // end of switch: break #2 transfers control here
    } // end while
```

The `break` labeled #1 terminates the `for` loop that follows the hyphen `case` label. It does not terminate the enclosing `switch` statement and in fact does not even terminate the processing for the current case. Processing continues with the first statement following the `for`, which might be additional code to handle a hyphen or the `break` that completes that section.

The `break` labeled #2 terminates the `switch` but does not terminate the enclosing `while` loop. Processing continues after that `break` by executing the condition in the `while`.

EXERCISES SECTION 5.5.1

Exercise 5.20: Write a program to read a sequence of `string`s from the standard input until either the same word occurs twice in succession or all the words have been read. Use a `while` loop to read the text one word at a time. Use the `break` statement to terminate the loop if a word occurs twice in succession. Print the word if it occurs twice in succession, or else print a message saying that no word was repeated.

5.5.2 The `continue` Statement

A **`continue` statement** terminates the current iteration of the nearest enclosing loop and immediately begins the next iteration. A `continue` can appear only inside a `for`, `while`, or `do while` loop, including inside statements or blocks nested inside such loops. Like the `break` statement, a `continue` inside a nested loop affects only the nearest enclosing loop. Unlike a `break`, a `continue` may appear inside a `switch` only if that `switch` is embedded inside an iterative statement.

A `continue` interrupts the current iteration; execution stays inside the loop. In the case of a `while` or a `do while`, execution continues by evaluating the condition. In a traditional `for` loop, execution continues at the *expression* inside the `for` header. In a range `for`, execution continues by initializing the control variable from the next element in the sequence.

For example, the following loop reads the standard input one word at a time. Only words that begin with an underscore will be processed. For any other value, we terminate the current iteration and get the next input:

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // get another input
    // still here? the input starts with an underscore; process buf . . .
}
```

EXERCISES SECTION 5.5.2

Exercise 5.21: Revise the program from the exercise in § 5.5.1 (p. 191) so that it looks only for duplicated words that start with an uppercase letter.

5.5.3 The goto Statement

A **goto statement** provides an unconditional jump from the goto to a another statement in the same function.

Best Practices: Programs should not use gotos. gotos make programs hard to understand and hard to modify.

The syntactic form of a goto statement is

```
goto label;
```

where *label* is an identifier that identifies a statement. A **labeled statement** is any statement that is preceded by an identifier followed by a colon:

```
end: return;  // labeled statement; may be the target of a goto
```

Label identifiers are independent of names used for variables and other identifiers. Hence, a label may have the same identifier as another entity in the program without interfering with the other uses of that identifier. The goto and the labeled statement to which it transfers control must be in the same function.

As with a switch statement, a goto cannot transfer control from a point where an initialized variable is out of scope to a point where that variable is in scope:

```
    // . . .
    goto end;
    int ix = 10; // error: goto bypasses an initialized variable definition
end:
    // error: code here could use ix but the goto bypassed its declaration
    ix = 42;
```

A jump backward over an already executed definition is okay. Jumping back to a point before a variable is defined destroys the variable and constructs it again:

```
// backward jump over an initialized variable definition is okay
  begin:
    int sz = get_size();
    if (sz <= 0) {
          goto begin;
    }
```

Here sz is destroyed when the goto executes. It is defined and initialized anew when control passes back through its definition after the jump back to begin.

EXERCISES SECTION 5.5.3

Exercise 5.22: The last example in this section that jumped back to `begin` could be better written using a loop. Rewrite the code to eliminate the `goto`.

5.6 `try` Blocks and Exception Handling

Exceptions are run-time anomalies—such as losing a database connection or encountering unexpected input—that exist outside the normal functioning of a program. Dealing with anomalous behavior can be one of the most difficult parts of designing any system.

Exception handling is generally used when one part of a program detects a problem that it cannot resolve and the problem is such that the detecting part of the program cannot continue. In such cases, the detecting part needs a way to signal that something happened and that it cannot continue. Moreover, the detecting part needs a way to signal the problem without knowing what part of the program will deal with the exceptional condition. Having signaled what happened, the detecting part stops processing.

A program that contains code that might raise an exception (usually) has another part to handle whatever happened. For example, if the problem is invalid input, the handling part might ask the user to provide correct input. If the database was lost, the handling part might alert an operator.

Exception handling supports this cooperation between the detecting and handling parts of a program. In C++, exception handling involves

- **`throw` expressions**, which the detecting part uses to indicate that it encountered something it can't handle. We say that a `throw` **raises** an exception.
- **`try` blocks**, which the handling part uses to deal with an exception. A `try` block starts with the keyword `try` and ends with one or more **`catch` clauses**. Exceptions thrown from code executed inside a `try` block are usually handled by one of the `catch` clauses. Because they "handle" the exception, `catch` clauses are also known as **exception handlers**.
- A set of **`exception` classes** that are used to pass information about what happened between a `throw` and an associated `catch`.

In the remainder of this section, we'll introduce these three components of exception handling. We'll also have more to say about exceptions in § 18.1 (p. 772).

5.6.1 A `throw` Expression

The detecting part of a program uses a `throw` expression to raise an exception. A `throw` consists of the keyword `throw` followed by an expression. The type of the expression determines what kind of exception is thrown. A `throw` expression is usually followed by a semicolon, making it into an expression statement.

As a simple example, recall the program in § 1.5.2 (p. 23) that added two objects of type `Sales_item`. That program checked whether the records it read referred to the same book. If not, it printed a message and exited.

```
Sales_item item1, item2;
cin >> item1 >> item2;
// first check that item1 and item2 represent the same book
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0;    // indicate success
} else {
    cerr << "Data must refer to same ISBN"
              << endl;
    return -1;   // indicate failure
}
```

In a more realistic program, the part that adds the objects might be separated from the part that manages the interaction with a user. In this case, we might rewrite the test to throw an exception rather than returning an error indicator:

```
// first check that the data are for the same item
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// if we're still here, the ISBNs are the same
cout << item1 + item2 << endl;
```

In this code, if the ISBNs differ, we throw an expression that is an object of type `runtime_error`. Throwing an exception terminates the current function and transfers control to a handler that will know how to handle this error.

The type `runtime_error` is one of the standard library exception types and is defined in the `stdexcept` header. We'll have more to say about these types in § 5.6.3 (p. 197). We must initialize a `runtime_error` by giving it a `string` or a C-style character string (§ 3.5.4, p. 122). That string provides additional information about the problem.

5.6.2 The `try` Block

The general form of a `try` block is

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

A `try` block begins with the keyword `try` followed by a block, which, as usual, is a sequence of statements enclosed in curly braces.

Following the try block is a list of one or more catch clauses. A catch consists of three parts: the keyword catch, the declaration of a (possibly unnamed) object within parentheses (referred to as an **exception declaration**), and a block. When a catch is selected to handle an exception, the associated block is executed. Once the catch finishes, execution continues with the statement immediately following the last catch clause of the try block.

The *program-statements* inside the try constitute the normal logic of the program. Like any other blocks, they can contain any C++ statement, including declarations. As with any block, variables declared inside a try block are inaccessible outside the block—in particular, they are not accessible to the catch clauses.

Writing a Handler

In the preceding example, we used a throw to avoid adding two Sales_items that represented different books. We imagined that the part of the program that added two Sales_items was separate from the part that communicated with the user. The part that interacts with the user might contain code something like the following to handle the exception that was thrown:

```
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that the ISBNs must match and prompt for another pair
        cout << err.what()
             << "\nTry Again?  Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break;      // break out of the while loop
    }
}
```

The ordinary logic of the program that manages the interaction with the user appears inside the try block. This part of the program is wrapped inside a try because it might throw an exception of type runtime_error.

This try block has a single catch clause, which handles exceptions of type runtime_error. The statements in the block following the catch are executed if code inside the try block throws a runtime_error. Our catch handles the error by printing a message and asking the user to indicate whether to continue. If the user enters 'n', then the break is executed and we exit the while. Otherwise, execution falls through to the closing brace of the while, which transfers control back to the while condition for the next iteration.

The prompt to the user prints the return from err.what(). We know that err has type runtime_error, so we can infer that what is a member function (§ 1.5.2, p. 23) of the runtime_error class. Each of the library exception classes defines a member function named what. These functions take no arguments and return a C-style character string (i.e., a const char*). The what member of `runtime_error` returns a copy of the `string` used to initialize the particular object. If the code described in the previous section threw an exception, then this `catch` would print

```
Data must refer to same ISBN
Try Again?  Enter y or n
```

Functions Are Exited during the Search for a Handler

In complicated systems, the execution path of a program may pass through multiple `try` blocks before encountering code that throws an exception. For example, a `try` block might call a function that contains a `try`, which calls another function with its own `try`, and so on.

The search for a handler reverses the call chain. When an exception is thrown, the function that threw the exception is searched first. If no matching `catch` is found, that function terminates. The function that called the one that threw is searched next. If no handler is found, that function also exits. That function's caller is searched next, and so on back up the execution path until a `catch` of an appropriate type is found.

If no appropriate `catch` is found, execution is transferred to a library function named **`terminate`**. The behavior of that function is system dependent but is guaranteed to stop further execution of the program.

Exceptions that occur in programs that do not define any `try` blocks are handled in the same manner: After all, if there are no `try` blocks, there can be no handlers. If a program has no `try` blocks and an exception occurs, then `terminate` is called and the program is exited.

CAUTION: WRITING EXCEPTION SAFE CODE IS *Hard*

It is important to realize that exceptions interrupt the normal flow of a program. At the point where the exception occurs, some of the computations that the caller requested may have been done, while others remain undone. In general, bypassing part of the program might mean that an object is left in an invalid or incomplete state, or that a resource is not freed, and so on. Programs that properly "clean up" during exception handling are said to be *exception safe*. Writing exception safe code is surprisingly hard, and (largely) beyond the scope of this language Primer.

Some programs use exceptions simply to terminate the program when an exceptional condition occurs. Such programs generally don't worry about exception safety.

Programs that do handle exceptions and continue processing generally must be constantly aware of whether an exception might occur and what the program must do to ensure that objects are valid, that resources don't leak, and that the program is restored to an appropriate state.

We will occasionally point out particularly common techniques used to promote exception safety. However, readers whose programs require robust exception handling should be aware that the techniques we cover are insufficient by themselves to achieve exception safety.

5.6.3 Standard Exceptions

The C++ library defines several classes that it uses to report problems encountered in the functions in the standard library. These exception classes are also intended to be used in the programs we write. These classes are defined in four headers:

- The `exception` header defines the most general kind of exception class named `exception`. It communicates only that an exception occurred but provides no additional information.
- The `stdexcept` header defines several general-purpose exception classes, which are listed in Table 5.1.
- The `new` header defines the `bad_alloc` exception type, which we cover in § 12.1.2 (p. 458).
- The `type_info` header defines the `bad_cast` exception type, which we cover in § 19.2 (p. 825).

Table 5.1: Standard Exception Classes Defined in `<stdexcept>`

`exception`	The most general kind of problem.
`runtime_error`	Problem that can be detected only at run time.
`range_error`	Run-time error: result generated outside the range of values that are meaningful.
`overflow_error`	Run-time error: computation that overflowed.
`underflow_error`	Run-time error: computation that underflowed.
`logic_error`	Error in the logic of the program.
`domain_error`	Logic error: argument for which no result exists.
`invalid_argument`	Logic error: inappropriate argument.
`length_error`	Logic error: attempt to create an object larger than the maximum size for that type.
`out_of_range`	Logic error: used a value outside the valid range.

The library exception classes have only a few operations. We can create, copy, and assign objects of any of the exception types.

We can only default initialize (§ 2.2.1, p. 43) `exception`, `bad_alloc`, and `bad_cast` objects; it is not possible to provide an initializer for objects of these exception types.

The other exception types have the opposite behavior: We can initialize those objects from either a `string` or a C-style string, but we *cannot* default initialize them. When we create objects of any of these other exception types, we must supply an initializer. That initializer is used to provide additional information about the error that occurred.

The exception types define only a single operation named `what`. That function takes no arguments and returns a `const char*` that points to a C-style character string (§ 3.5.4, p. 122). The purpose of this C-style character string is to provide some sort of textual description of the exception thrown.

The contents of the C-style string that `what` returns depends on the type of the exception object. For the types that take a string initializer, the `what` function returns that string. For the other types, the value of the string that `what` returns varies by compiler.

Exercises Section 5.6.3

Exercise 5.23: Write a program that reads two integers from the standard input and prints the result of dividing the first number by the second.

Exercise 5.24: Revise your program to throw an exception if the second number is zero. Test your program with a zero input to see what happens on your system if you don't `catch` an exception.

Exercise 5.25: Revise your program from the previous exercise to use a `try` block to `catch` the exception. The `catch` clause should print a message to the user and ask them to supply a new number and repeat the code inside the `try`.

Chapter Summary

C++ provides a limited number of statements. Most of these affect the flow of control within a program:

- `while`, `for`, and `do while` statements, which provide iterative execution.
- `if` and `switch`, which provide conditional execution.
- `continue`, which stops the current iteration of a loop.
- `break`, which exits a loop or `switch` statement.
- `goto`, which transfers control to a labeled statement.
- `try` and `catch`, which define a `try` block enclosing a sequence of statements that might throw an exception. The `catch` clause(s) are intended to handle the exception(s) that the enclosed code might throw.
- `throw` expression statements, which exit a block of code, transferring control to an associated `catch` clause.
- `return`, which stops execution of a function. (We'll cover `return` statements in Chapter 6.)

In addition, there are expression statements and declaration statements. An expression statement causes the subject expression to be evaluated. Declarations and definitions of variables were described in Chapter 2.

Defined Terms

block Sequence of zero or more statements enclosed in curly braces. A block is a statement, so it can appear anywhere a statement is expected.

break statement Terminates the nearest enclosing loop or `switch` statement. Execution transfers to the first statement following the terminated loop or `switch`.

case label Constant expression (§ 2.4.4, p. 65) that follows the keyword `case` in a `switch` statement. No two `case` labels in the same `switch` statement may have the same value.

catch clause The `catch` keyword, an exception declaration in parentheses, and a block of statements. The code inside a `catch` clause does whatever is necessary to handle an exception of the type defined in its exception declaration.

compound statement Synonym for block.

continue statement Terminates the current iteration of the nearest enclosing loop. Execution transfers to the loop condition in a `while` or `do`, to the next iteration in a range `for`, or to the expression in the header of a traditional `for` loop.

dangling else Colloquial term used to refer to the problem of how to process nested `if` statements in which there are more `if`s than `else`s. In C++, an `else` is always paired with the closest preceding unmatched `if`. Note that curly braces can be used to effectively hide an inner `if` so that the programmer can control which `if` a given `else` should match.

default label `case` label that matches any otherwise unmatched value computed in the `switch` expression.

do while statement Like a `while`, except that the condition is tested at the end of the loop, not the beginning. The statement inside the `do` is executed at least once.

exception classes Set of classes defined by the standard library to be used to represent errors. Table 5.1 (p. 197) lists the general-purpose exception classes.

exception declaration The declaration in a `catch` clause. This declaration specifies the type of exceptions the `catch` can handle.

exception handler Code that deals with an exception raised in another part of the program. Synonym for `catch` clause.

exception safe Term used to describe programs that behave correctly when exceptions are thrown.

expression statement An expression followed by a semicolon. An expression statement causes the expression to be evaluated.

flow of control Execution path through a program.

for statement Iteration statement that provides iterative execution. Ordinarily used to step through a container or to repeat a calculation a given number of times.

goto statement Statement that causes an unconditional transfer of control to a specified labeled statement elsewhere in the same function. `goto`s obfuscate the flow of control within a program and should be avoided.

if else statement Conditional execution of code following the `if` or the `else`, depending on the truth value of the condition.

if statement Conditional execution based on the value of the specified condition. If the condition is `true`, then the `if` body is executed. If not, control flows to the statement following the `if`.

labeled statement Statement preceded by a label. A label is an identifier followed by a colon. Label identifiers are independent of other uses of the same identifier.

null statement An empty statement. Indicated by a single semicolon.

raise Often used as a synonym for throw. C++ programmers speak of "throwing" or "raising" an exception interchangeably.

range for statement Statement that iterates through a sequence.

switch statement A conditional statement that starts by evaluating the expression that follows the `switch` keyword. Control passes to the labeled statement with a `case` label that matches the value of the expression. If there is no matching label, execution either continues at the `default` label, if there is one, or falls out of the `switch` if there is no `default` label.

terminate Library function that is called if an exception is not caught. `terminate` aborts the program.

throw expression Expression that interrupts the current execution path. Each `throw` throws an object and transfers control to the nearest enclosing `catch` clause that can handle the type of exception that is thrown.

try block Block enclosed by the keyword `try` and one or more `catch` clauses. If the code inside a `try` block raises an exception and one of the `catch` clauses matches the type of the exception, then the exception is handled by that `catch`. Otherwise, the exception is handled by an enclosing `try` block or the program terminates.

while statement Iteration statement that executes its target statement as long as a specified condition is `true`. The statement is executed zero or more times, depending on the truth value of the condition.