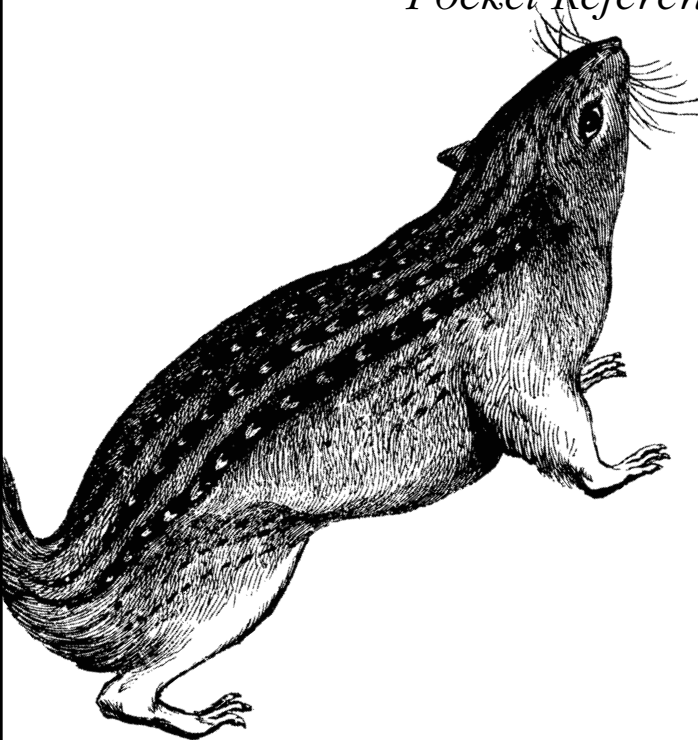


C++ Syntax and Fundamentals

C++

Pocket Reference



O'REILLY®

Kyle Loudon

C++

Pocket Reference

C++
Pocket Reference

Kyle Loudon

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Statements

There are many types of statements in C++. Some simply evaluate expressions, while others change the order in which statements are executed in the future.

Expression Statements

An *expression statement* is an expression followed by a single semicolon (;). Expression statements cause an expression to be evaluated. Side effects, such as an assignment to a variable, are completed before the next statement is executed. For example:

```
a = 10;
```

Null Statements

A *null statement* is written as a semicolon (;). Null statements are useful when the syntax of C++ requires a statement but you don't need anything performed. For example:

```
void spin(int n)
{
    for (int i = 0; i < n; i++)
        ;
}
```

This loop simply counts to a specified value, as might be required to insert a delay in a real-time system (assuming the compiler doesn't optimize the loop away altogether).

Compound Statements

A *compound statement* is a group of zero or more statements beginning with a left brace ({) and ending with a right brace (}). For example:

```
while (true)
{
    // Start of a compound statement.
    ...
}
```

```

    if (!done)
    {
        // Another compound statement.
    }
    else
    {
        // Another compound statement.
    }
}

```

Compound statements are often called *blocks*. A block defines a region that has its own local scope.

Iteration Statements

Iteration statements cause a statement or block to be executed repeatedly. There are three types of iteration statements in C++: *while*, *do*, and *for*.

while

A *while* loop repeats a statement or block as long as an expression (which can be a declaration) evaluated at the top of the loop is true. For example:

```

char          ch = 'y';

while (ch == 'y')
{
    // Do something to be repeated.
    ...

    cout << "Do it again (y or n)? ";
    cin  >> ch;
}

```

This repeats a block as long as *ch* is 'y'. The body of the loop is never executed if the expression at the top of the loop is false when the loop is first encountered.

do

A *do* loop repeats a statement or block as long as an expression evaluated at the bottom of the loop is true. For example:

```

char          ch;

do
{
    // Do something to be repeated.
    ...

    cout << "Do it again (y or n)? ";
    cin  >> ch;
} while (ch == 'y');

```

This repeats a block as long as `ch` is `'y'`; however, the body of the loop is executed at least once because the condition for looping is evaluated at the end of each iteration.

for

A `for` loop is similar to a `while` loop, but additional mechanisms are provided for initializing the loop and making adjustments after each of its iterations. For example:

```

// Prevent warnings in Visual C++.
#pragma warning(disable:4786)

typedef map<int, string> IntStringMap;

IntStringMap    m;
char            s[4];

for (int i = 0; i < 10; i++)
{
    s[0] = 'a' + i; s[1] = 'b' + i;
    s[2] = 'c' + i; s[3] = '\\0';

    m.insert(IntStringMap::value_type(i,
        string(s)));
}

```

The key to `for` loops is understanding the statements that go within the parentheses following the `for` keyword. The statement before the first semicolon is the statement executed to initialize the loop. Before each iteration, including the first, the expression between the two semicolons is evaluated. If the expression is true, the loop body is executed; otherwise, the

loop terminates. After each iteration, the rightmost expression is evaluated, and the cycle is repeated. `for` loops can contain more complicated expressions as well. For example:

```
void upperString(char *t, const char *s)
{
    for (; *s != '\0'; *(t++) = toupper(*(s++)))
        ;

    *(t++) = '\0';
}
```

This function uses a `for` loop to translate string `s` to uppercase and copy it to `t`. The function assumes that storage has already been allocated for `t`. A null statement is used for initialization since `s` and `t` are already initialized when the function starts. A null statement is also used for the loop body.

NOTE

A name declared in a `for` initialization statement is visible until the end of the `for` loop.

Selection Statements

Selection statements execute a different statement or block based on the result produced by an expression. There are two types of selection statements in C++: `if` and `switch`.

if

An `if` statement evaluates an expression (which may be a declaration) and uses the result to determine which of up to two statements or blocks to execute next. For example:

```
if (i > 0 && i < 100)
{
    // Do something when within range.
}
else
{
    // Do something when not in range.
}
```

If the expression is true, the statement or block immediately after the terminating parentheses of the `if` section is executed; otherwise, the statement or block after `else` is executed. If the `else` clause is omitted, nothing is performed when the expression is not true. When `if` statements are nested within one another, `else` clauses associate with the nearest `if`.

switch

A `switch` statement selects one of several sections of code to execute based on the value of a controlling expression. For example:

```
switch (type)
{
    case keyDown:
        // Do something for a key down.
        ...
        break;

    case keyUp:
        // Do something for a key up.
        ...
        break;

    ...
    default:
        // Handle anything not handled.
        ...
}
```

Each section is identified by the keyword `case` followed by an expression that must evaluate to a distinct, constant, integral value at compile time. At runtime, execution branches to the section identified by the value matching that of the controlling expression, and continues from that point onward. A `break` statement (see “Jump Statements”) is used at the end of each case to prevent the code associated with all subsequent cases from being executed. An optional `default` case can be provided to handle the situation when the value of the controlling expression does not match any of the cases.

Jump Statements

Jump statements jump unconditionally to a different statement. There are four types of jump statements in C++: `break`, `continue`, `goto`, and `return`.

break

A `break` statement is used to jump out of an innermost loop or `switch` statement. For example:

```
for (;;)
{
    if (done)
        break;

    // When we're done, set done to true
    // so we break on the next iteration.
    ...
}
```

This is a `for` loop that has no condition specified for stopping it. A `break` statement is used to break out of the loop when `done` is true.

continue

A `continue` statement is used to jump to the start of an innermost enclosing loop. For example:

```
while (!done)
{
    // If we need to skip the end portion
    // of the function, set skip to true.
    ...

    if (skip)
        continue;

    // This gets skipped when continuing.
    ...
}
```

This is a `while` loop that contains statements to be skipped when `skip` has been set to true. A `continue` statement is used

to jump to the start of the loop in order to skip these statements.

goto

A goto statement jumps to a label that you specify. For example:

```
if (GetLastError() != ERROR_SUCCESS)
    goto handleError;

// This code gets skipped on an error.
...

handleError:
// Do something to handle error state.
```

Because goto statements lead to unstructured code, they are seldom used.

return

A return statement jumps out of a function, and if needed sets a return value. For example:

```
double convertToSM(double nm)
{
    return nm * NMPPerSM;
}
```

A return statement can be called from anywhere within a function. The type of the value returned must be, or be convertible to, the return type of the function. Functions that return void do not require return statements. For example:

```
void sayHello()
{
    cout << "Hello" << endl;
}
```

A return is performed automatically when this function completes. However, a return statement with no value can be used to jump out of the function before the end of the function block is reached.