The goto Statement:  
Legitimate Uses

The goto statement is, deservedly, much maligned. However, there remain at least two legitimate uses in contemporary programming languages like C++.

Exiting Nested Loops

Chapter 3 introduced the break and continue statements, which are used in conjunction with loops. The general usage looks like one of the following:

```c
for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        break;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        continue;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        goto next;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        goto done;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        goto next;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        goto next;
    ...
    ...
} for (int i = 0; i < counter; i++) {
    ...
    ...
    ...
    if (condition)
        goto next;
    ...
    ...
}
```

When a break statement is executed, all statements between the break and the end of the loop are skipped. Execution resumes with the first statement following the end of the loop. When a continue statement is executed, all statements between the continue and the end of the loop are skipped. However, execution begins at the top of the loop: if it is a for-loop, the increment expression is executed, then (for all loops) the test is evaluated and if it is true, the next iteration of the loop takes place.

However, the break and continue statements only work for one level of loop. If two or more loops are nested, break and continue only work for the loop in which they are called (i.e., if called from a nested loop, they cannot effect an outer loop). The goto statement is typically used when breaking or continuing out of nested loops:
Implementing State Machines

A state machine is an abstraction used in many disciplines to describe the behavior of both hardware and software. They are composed of a finite set of states and transitions. A state represents the current activity of the hardware or software. A transition is a legal way that the machine can change from one state to another. Transitions are triggered by events or by reading an input (which is really just an event). State machines are especially useful in cases where the current behavior depends on the past sequence of events or inputs.

The following state machine, represented as a diagram, describes a program that detects the two styles of comments permitted in a C++ program and separates them from the rest of the program.

Each state is represented by a “roundtangle” (a rectangle with radiused or rounded corners) and each transition is represented by an arrow from a source to a destination state. The solid disk denotes the start state; the “bulls eye” denotes a legal end state. The statements in square brackets are called guard conditions. A transition may only be taken if the optional guard condition is true. The C++ program implementing this state machine is posted on the CS 1220 web page under chapter 3.

You’ll learn more about state machines in CS 2750.