The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.6.1 The goto statement

Constraints

goto statement

The identifier in a **goto** statement shall name a label located somewhere in the enclosing function. 1787

Commentary

The situation where a label having a corresponding name does not occur within the enclosing function is likely to be some kind of fault. Having translators issue a diagnostic is probably the most useful behavior for the standard to specify.

Other Languages

Some languages that support nested function definitions (e.g., Pascal) only require that the label name be visible (i.e., it is possible to jump to a label in a different function). Other languages (e.g., Algol 68) give labels block scope, which restricts its visibility. Perl has a form of **goto** that causes the function named by the label to be called. However, when that function returns control is returned to the function that called the function that performed the **goto** (i.e., the behavior is as-if control returns to the **goto** which then immediately performs a return).

goto past variably modified type

A **goto** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier. 1788

Commentary

VLA lifetime starts/ends
storage layout

VLA lifetime starts/ends
switch past variably modified type

Storage allocation for objects having a variably modified type differs from storage allocation for objects having other types in that it requires internal implementation housekeeping code to be executed, when its definition is encountered during program execution, for it to occur (storage for objects having other types can be reserved at translation time). Similarly, storage deallocation requires housekeeping code to be executed when the lifetime of an object having a variably modified type ends (implementation techniques that have no dependency between execution of an objects definition and the termination of its lifetime discussed elsewhere); storage deallocation for objects having other types does not require any execution time actions.

This constraint, along with an equivalent one for the **switch** statement, provides a guarantee to implementations that if the storage deallocation code is executed, then the storage allocation code will have been previous executed (i.e., there is no way to bypass it in a conforming program). Jumping from outside of the scope of an identifier denoting such an object to inside the scope of that identifier bypasses execution of its definition. Bypassing the definition might be thought to be harmless if the object is never referenced. However, storage for the object has to be deallocated when its lifetime terminates, which is a (implicit) reference to the object.

This constraint does not require the identifier to be visible at the jumped to label, only that the label be within the scope of the identifier. Also, there is no constraint prohibiting the block containing the **goto** statement from containing an object defined to have a variably modified type and the destination of the jump to be outside of that block.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard. However, the C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that being visible rather than *in scope* more accurately reflects the intent):

6.7p3 *A program that jumps*[77] *from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).*

A C function that performs a jump into a block that declares an object with an initializer will cause a C++ translator to issue a diagnostic.

```
void f(void)
{
goto lab;    /* strictly conforming */
             // ill-formed
int loc = 1;

lab: ;
}
```

Semantics

1789 A **goto** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

goto causes unconditional jump

Commentary

Unless prefixed by a label, statements appearing after a **goto** statement, in the same block, are dead code. dead code

Other Languages

Some languages (e.g., Pascal) support jumping to statements contained within other functions.

Common Implementations

There are a number of practical algorithms for analyzing program control flow that depend on the graph of the control flow being reducible[2,4] (i.e., not irreducible; an irreducible graph might be thought of one that contains a cycle with multiple entry points). To create an irreducible flow graph in C requires the use of the **goto** statement, for instance:

```
     if (a > b)
        goto L_1;
L_2: a++;
L_1: if (a < 3)
        goto L_2;
```

Although algorithms are available for transforming an irreducible flow graph into a reducible one,[5] many optimizers and static analyzers don't perform such transformations (because of the potentially significant increase in code size created by the node splitting algorithms,[3] and because it is believed that function definitions rarely have a control flow that is irreducible[1]). For this reason the quality of the machine code generated for the few functions that do contain an irreducible control flow graph can be much lower than that for other functions (because irreducibility can prevent some of the information needed for optimization being deduced).

1790 EXAMPLE 1 It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

goto EXAMPLE

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **continue** statements, for example.)

```
/* ... */
goto first_time;
for (;;) {
        // determine next operation
```

```
                /* ... */
                if (need to reinitialize) {
                        // reinitialize-only code
                        /* ... */
                first_time:
                        // general initialization code
                        /* ... */
                        continue;
                }
                // handle other operations
                /* ... */
        }
```

Commentary

While this might seem like a contrived example, the same could probably be said for all examples of the use of the **goto** statement.

EXAMPLE goto variable length

EXAMPLE 2 1791

A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
        goto lab3;                      // invalid: going INTO scope of VLA.
        {
                double a[n];
                a[j] = 4.4;
        lab3:
                a[j] = 3.3;
                goto lab4;              // valid: going WITHIN scope of VLA.
                a[j] = 5.5;
        lab4:
                a[j] = 6.6;
        }
        goto lab4;                      // invalid: going INTO scope of VLA.
```

Commentary

goto 1788 past variably modified type

It is a constraint violation for a **goto** statement to jump past any declarations of objects with variably modified types.

References

1. F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture*. Morgan Kaufmann Publishers, 2002.
3. L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the* 30^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 106–114, Jan. 2003.
4. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
5. S. Unger. *Transforming Irreducible Regions of Control Flow into Reducible Regions by Optimized Node Splitting*. PhD thesis, Humboldt Universität zu Berlin, 1998.