The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.4.1 The if statement

Constraints

if statement controlling expression scalar type

The controlling expression of an **if** statement shall have scalar type. 1743

Commentary

Although the type **_Bool** was introduced in C99 the Committee decided that there was too much existing code to change the specification for the controlling expression type.

C++

6.4p4 *The value of a condition that is an expression is the value of the expression, implicitly converted to* ***bool*** *for statements other than* ***switch****; if that conversion is ill-formed, the program is ill-formed.*

If only constructs that are available in C are used the set of possible expressions is the same.

Other Languages

In many languages the controlling expression is required to have a boolean type. Languages whose design has been influenced by C often allow the controlling expression to have scalar type.

Coding Guidelines

selection statement syntax

The broader contexts in which readers need to comprehend controlling expression are discussed elsewhere. This subsection concentrates on the form of the controlling expression.

The value of a controlling expression is used to make one of two choices. Values used in this way are generally considered to have a boolean role. Some languages require the controlling expression to have a boolean type and their translators enforce this requirement. Some coding guideline documents contain recommendations that effectively try to duplicate this boolean type requirement found in other languages. Recommendations based on type not only faces technical problems in their wording and implementation (caused by the implicit promotions and conversions performed in C), but also fail to address the real issues of developer comprehension and performance.

In the context of an **if** statement do readers of the source distinguish between expressions that have two possible values (i.e., boolean roles), and expressions that may have more than two values being used in a context where an implicit test against zero is performed? Is the consideration of boolean roles a cultural baggage carried over to C by developers who have previously used them in other languages? Do readers who have only ever programmed in C make use of boolean roles, or do they think in terms of *a test against zero*? In the absence of studies of developer mental representations of algorithmic and source code constructs, it is not possible to reliably answer these questions. Instead the following discussion looks at the main issues involved in making use of boolean roles and making use of the implicit *a test against zero* special case.

A boolean role is not about the type of an expression (prior to the introduction of the type **_Bool** in C99, a character type was often used as a stand-in), but about the possible values an expression may have and how they are used. The following discussion applies whether a controlling expression has an integer, floating, or pointer type.

In some cases the top-level operator of a controlling expression returns a result that is either zero or one (e.g., the relational and equality operators). The visibility, in the source, of such an operator signals its boolean role to readers. However, in other cases (see Table **??**) developers write controlling expressions that do not contain explicit comparisons (the value of a controlling expression is implicitly compared against zero). What are the costs and benefits of omitting an explicit comparison? The following code fragment contains examples of various ways of writing a controlling expression:

```
1  if (flag)           /* 1 */
2     /* ... */
```

```
3
4   if (int_value)       /* 2 */
5      /* ... */
6
7   if (flag == TRUE)    /* 3 */
8      /* ... */
9
10  if (int_value != 0)  /* 4 */
11     /* ... */
```

Does the presence of an explicit visual (rather than an implicit, in the developers mind) comparison reduce either the cognitive effort needed to comprehend the **if** statement or the likelihood of readers making mistakes? Given sufficient practice readers can learn to automatically process `if (x)` as if it had been written as `if (x != 0)`. The amount of practice needed to attain an automatic level of performance is unknown. Another unknown is the extent to which the token sequence `!= 0` acts as a visual memory aid.

When the visible form of the controlling expression is denoted by a single object (which may be an ordinary identifier, or the member of a structure, or some other construct where a value is obtained from an object) that name may provide information on the values it represents. To obtain this information readers might make use of the following:

- *Software development conventions.* In the software development community (and other communities) the term *flag* is generally understood to refer to something that can be in one of two states. For instance, the identifier `mimbo_flag` is likely to be interpreted as having two possible values relating to a *mimbo*, rather than referring to the national flag of Mimbo. Some naming conventions contain a greater degree of uncertainty than others. For instance, identifiers whose names contain the character sequence *status* sometimes represent more than two values.
- *Natural language knowledge.* Speakers of English regard some prepositions as being capable of representing two states. For instance, a cat *is* or *is not* black. This natural language usage is often adopted when selecting identifier names. For instance, `is_flobber` is likely to be interpreted as representing one of two states (being a, or having the attribute of, *flobber* or not).
- *Real world knowledge.* A program sometimes needs to take account of information from the *real world*. For instance, height above the ground is an important piece of information in an airplane flight simulator, with zero height having a special status.
- *Application knowledge.* The design of a program invariably makes use of knowledge about the application domain it operates within. For instance, the term `ziffer` may be used within the application domain that a program is intended to operate. Readers of the source will need the appropriate application knowledge to interpret the role of this identifier.
- *Program implementation conventions.* The design of a program involves creating and using various conventions. For instance, a program dealing with book printing may perform special processing for books that don't contain any pages (e.g., `num_pages` being zero is a special case).
- *Conventions and knowledge from different may be mixed together.* For instance, the identifier name `current_color` suggests that it represents color information. This kind of information is not usually thought about in terms of numeric values and there are certainly more than two colors. However, assigning values to symbolic qualities is a common software development convention, as is assigning a special interpretation to certain values (e.g., using zero to represent no known color, a program implementation convention).

The likelihood of a reader assuming that an identifier name has a boolean role will depend on the cultural beliefs and conventions they share with the author of the source. There is also the possibility that rather than using the identifier name to deduce a boolean role, readers may use the context in which it occurs to infer a boolean role. This is an example of trust based usage. Requiring that values always be compared (against true/false or zero/nonzero) leads to infinite regression, as in the sequence:

boolean role
trust based usage

```
1  if (flag)
2  if (flag == TRUE)
3  if ((flag == TRUE) == TRUE)
4  and so on...
```

At some point readers have to make a *final* comparison in their own mind. The inability to calculate (i.e., automatically enforceable) the form a controlling expression should take to minimize readers cognitive effort prevents any guideline recommendations being made here.

Semantics

---

if statement operand compare against 0

In both forms, the first substatement is executed if the expression compares unequal to 0. 1744

Commentary

null pointer constant

Depending on the type of the other operand this 0 may be converted to an integer type of greater rank, a floating-point `0.0`, or a null pointer constant.

C++

The C++ Standard expresses this behavior in terms of true and false (6.4.1p1). The effect is the same.

Other Languages

In languages that support a boolean type this test is usually expressed in terms of **`true`** and **`false`**.

Common Implementations

logical negation result is && operand compare against 0

The machine code generation issues are similar to those that apply to the logical operators. The degree to which this comparison can be optimized away depends on the form of the controlling expression and the processor instruction set. If the controlling expressions top-level operator is one that always returns a value of zero or one (e.g., an equality or relational operator), it is possible to generate machine code that performs a branch rather than returning a value that is then compared. Some processors have a single instruction that performs a comparison and branch, while others have separate instructions (the comparison instruction setting processor condition flags that are then tested by a conditional branch instruction). On some processors simply loading a value into a register also results in a comparison against zero being made, with the appropriate processor condition flags being set. The use of conditional instructions is discussed elsewhere.

conditional instructions

basic block

The machine code for the first substatement is often placed immediately after the code to evaluate the controlling expression. However, optimizers may reorder blocks of code in an attempt to maximize instruction cache utilization.

---

else

In the **`else`** form, the second substatement is executed if the expression compares equal to 0. 1745

Commentary

equality operators exactly one relation is true

Implementations are required to ensure that exactly one of the equality comparisons is true.

Coding Guidelines

Some coding guideline documents recommend that the **`else`** form always be present, even if it contains no executable statements. Such a recommendation has the benefit of ensuring that there are never any mismatching **`if`**/**`else`** pairs. However, then the same effect can be achieved by requiring nested **`if`** statements to be enclosed in braces (this issue is discussed elsewhere). The cost of adding empty **`else`** forms increases the amount of source code that may need to be read and in some cases decrease in the amount of non-null source that appears on a display device. Such a guideline recommendation does not appear worthwhile.

using braces block

Usage

In the visible form of the `.c` files 21.5% of **`if`** statements have an **`else`** form. (Counting all forms of *if* supported by the preprocessor, with **`#elif`** counting as both an *if* and an *else*, there is an **`#else`** form in 25.0% of cases.)

1746 If the first substatement is reached via a label, the second substatement is not executed.

Commentary

The flow of control of a sequence of statements is not influenced by how they were initially reached, in the flow of control. The label may be reached as a result of executing a **switch** statement, or a **goto** statement. The issue of jumping into nested blocks or the body of iteration statements is discussed elsewhere.

switch statement
causes jump
goto
causes unconditional jump
jump statement
causes jump to
iteration statement
executed repeatedly

C++

The C++ Standard does not explicitly specify the behavior for this case.

Other Languages

This statement applies to all programming languages that support jumps into more deeply nested blocks.

1747 An **else** is associated with the lexically nearest preceding **if** that is allowed by the syntax.

else
binds to nearest if

Commentary

As it appears in the standard the syntax for **if** statements is ambiguous on how an **else** should be associated in a nested **if** statement. This semantic rule resolves this ambiguity.

selection statement
syntax

Other Languages

Languages that support nesting of conditional statements need a method of resolving which construct an **else** binds to. The rules used include the following:

- Not supporting in the language syntax unbracketed nesting (i.e., requiring braces or **begin**/**end** pairs) within the *then* arm. For instance, Algol 60 permits the usage `IF q1 THEN a1 ELSE IF q2 THEN a2 ELSE a3`, but the following is a syntax violation `IF q1 THEN IF q2 THEN a1 ELSE a2 ELSE a3`.
- Using a matching token to pair with the **if**. The keyword **fi** is a common choice (used by Ada, Algol 68, while the C preprocessor uses **endif**). In this case the bracketing formed by the **if**/**fi** prevents any ambiguity occurring.
- Like C— using the *nearest preceding* rule.

Coding Guidelines

If the guideline recommendation on using braces is followed there will only ever be one lexically preceding **if** that an **else** can be associated with. Some coding guideline documents recommend that an **if** statement always have an associated **else** form, even if it only contains the null statement.

?? if statement
block not an if statement

null statement

References