The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.4 Selection statements

selection statement syntax


```
selection-statement:
                if ( expression ) statement
                if ( expression ) statement else statement
                switch ( expression ) statement
```

Commentary

else binds to nearest if

This syntax is ambiguous in that it does not uniquely specify the parse for the token sequence `if (a) b; if (c) d; else e;`. This syntactic ambiguity is resolved by a semantic rule.

Other Languages

There is a great deal of variety in the syntactic forms of selection statements used by different languages. Some languages (e.g., those in the Pascal family) require that the controlling expression be followed by the keyword **then**, while others require the **if** statement to be terminated by some keyword (e.g., **end** in Ada, or **fi** in Algol 68). Some languages support an **elif** form (e.g., Ada, Algol 68, and the C preprocessor). Fortran has what it calls an *arithmetic if* statement:

preprocessor directives syntax

```
1         if (i) 10, 20, 30
2        C Jump to label 10 if i < 0
3        C Jump to label 20 if i == 0
4        C Jump to label 30 if i > 0
```

The Fortran equivalent of the **switch** statement is known as a *computed goto* (in the following code fragment the **goto** jumps to label 10 if `I==1`, 20 if `I==2`, and so on):

```
1         goto (10, 20, 30, 40), I
```

Algol 60 required the labels to be declared in a **switch** statement:

```
1   SWITCH x := label1, label2, label3;
2         ...
3   GOTO x[i];
```

The Algol 68 **case** statement essentially has Fortran semantics (it also supports a **default** form), but with a Pascal like syntax.

Most languages (including Java) unconditionally bracket, syntactically, the complete list of case labels and their associated statements that follow a **switch** header.

Java supports the occurrence of labels that do not label any statement, at the end of the sequence of statements in a **switch** statement (where they essentially act as a comment whose contents are syntactically and semantically checked).

Languages in the Pascal family use the keyword **case**, rather than **switch**, is used. The following example is a fragment of Ada:

```
1   case x is
2      when 2 => y:=1;
3      when 3 => y:=2;
4   end case;
```

A few languages supports a **switch** statement containing more than one dimension (e.g., CHILL supports an arbitrary number, while Cobol supports a maximum of two). The following example is a fragment of Cobol:

```
1         EVALUATE price ALSO quantity
2            WHEN ANY ALSO 0   PERFORM ABC
3            WHEN 0   ALSO ANY PERFORM PQR
4            WHEN 1   ALSO num PERFORM XYZ
5         END-EVALUATE
```

Common Implementations

The availability of the **switch** statement does not mean that developers always use it. Uh[51] gives the following example from the source of grep.

```
1  if ((c = *sp++) == 0)
2     goto cerror;
3  if (c == '<') { ... }
4  if (c == '>') { ... }
5  if (c == '[') { ... }
6  if (c == ']') { ... }
7  if (c >= '1' && c <= '9') { ... }
```

Uh added optimizations to an existing compiler to detect and optimize linear sequences of **if** statements that compared the same variable against different constant values. Performance improvements varied between 0.67% to 5.56%, depending on the processor (see Table **??**).

Yang, Uh, and Whalley[53] took a different approach to optimizing nested sequences of **if** statements. They used profile information to reorder the sequence in which the controlling expression tests were made, putting those that succeeded most often first. An average performance improvement of 4% was obtained (on a variety of Unix tools).

The implementation of the **if** statement (and some operators, e.g., logical-AND) usually uses a conditional branch machine instruction. Conditional branch instructions create a bottleneck for processors that have the ability to execute more than one instruction at the same time. Without knowing which sequence of instructions are going to be executed, after the branch instruction, the processor is unable to keep the execution pipeline full; execution stalls. Several methods have been used to help reduce the likelihood of stalls occurring, including:

logical-AND-expression syntax

- *Speculative execution of the two flows of control.* Speculative execution of two flows of control is implemented by keeping the results of both execution paths in shadow registers. Once the result of the branch test is known the values in the appropriate set of shadow registers are copied into the actual registers. This approach requires a lot of hardware resources (it has been implemented on the IBM 360/91 and the Sun SuperSPARC[54]).
- *Predicting the branch the instruction is likely to take*, so called *branch prediction*, and speculatively executing that flow of control. If the prediction turns out to be correct the processor has a full pipeline of instructions and can continue from the point it had speculatively executed to, otherwise the pipeline has to be flushed and filled with instructions from the start location of the other branch.
- *Conditional instructions.* An alternative to a conditional branch is to use instructions whose execution is conditional on the setting of various processor flags. These instructions are read from the instruction stream but are only executed if the condition specified, as part of the encoding of the instruction, is currently true. The current conditions being set by executing a compare instruction, much like that which might appear before a conditional branch instruction.

1739 conditional instructions

Branch prediction

branch prediction

Most modern high-performance processors perform some kind of branch prediction followed by speculative execution of the predicted destination instructions (unconditional branches are discussed elsewhere). Processor based branch prediction can take the following two forms:

jump statement syntax

- Encoding the branch instruction to include a bit specifying whether the branch is likely to be taken or not (e.g., the IBM PowerPC). This branch probability decision bit is filled in by the translator when generating machine code.
- Using the history of an executing programs previous branch decisions (taken/not taken) is used to predict the probably outcome of the next branch decision. Studies have found that there is often a

strong correlation between the branch decisions made by a particular instruction (i.e., the same decision is repeated), also the last branch decision (usually a completely different branch instruction) and the decision made by the following branch instruction may be correlated (some branch predictors maintain information on the history of the branch itself and the branch previous to it, in the dynamic execution flow). This branch history information is often held in a table (a few bits specifying the direction of the last few jump decisions), indexed by the branch instructions address (usually a few of the least significant address bits).

The following are two methods a translator can use to predict branch direction:

- *Static analysis of the translated source code* (i.e., the machine code making up the program image). Various heuristics have proven to be reliable indicators of branch direction. This technique is known as *program-based* branch prediction.
- *Gather data on the branch decisions made during program execution.* This profile of branch decisions is then used in a subsequent translation to build a program image whose branch instructions are encoded to specify the most frequent branch direction taken during the profiling executions. This technique is known as *profile-based* branch prediction.

Static analysis of branch direction

The quantity measured in branch prediction is the *miss rate* (predicted destination incorrect), expressed as a percentage. The *perfect* predictor selects the branch direction based on the most common direction actually chosen, for each branch, over a large number of executions of the program. A study by Ball and Larus[2] analyzed translator generated machine code to create a number of heuristics that could be used to predict branch behavior. The heuristics found to be worthwhile (see Table 1739.1) were:

- *Opcode heuristic.* A relational comparison against zero is assumed to fail if it tests for less than (or less than or equal) and assumed to succeed if it tests for greater than (or greater than or equal).
- *Loop heuristic.* The successor does not postdominate[1739.1] the branch and is either a loop head or a loop preheader (i.e., passes control unconditionally to a loop head which it dominates). If the heuristic applies, predict the successor with the property.
- *Call heuristic.* The successor block contains a call or unconditionally passes control to a block with a call that it dominates, and the successor block does not postdominate the branch (many conditional calls handle exceptional cases, which are rarely taken). If the heuristic applies, predict the successor without the property.
- *Return heuristic.* The successor block contains a return or unconditionally passes control to a block that contains a return. If the heuristic applies, predict the successor without the property.
- *Guard heuristic.* Register r is an operand of the branch instruction, register r is used in the successor block before it is defined, and the successor block does not postdominate the branch (this test can only be performed after code generation and assumes that a translator has performed some form of optimizing register assignment). If the heuristic applies, predict the successor with the property.
- *Store heuristic.* The successor block contains a store instruction and does not postdominate the branch. If the heuristic applies, predict the successor without the property,
- *Pointer heuristic.* A test for equality between two pointers (one of them possibly being the null pointer constant) is assumed to fail. A test for inequality between two pointers (one of them possibly being the null pointer constant) is assumed to succeed.

[1739.1] A node y, of some flow graph, is said to *postdominate* the node x if every path from x to the end of the graph includes y.

Table 1739.1: Dynamic breakdown of non-loop branches for programs in SPEC89. *% of All Branches* is the percentage of all branches that are non-loop branches. *Heuristics* are the results of using the heuristics for predicting the target successor of each non-loop branch, *Perfect* the results for the perfect predictor, *Random* the results for predicting each non-loop branch randomly. *Big* is the number of non-loop branches in the program contributing more than 5% of all dynamic non-loop branches (and in parenthesis as a percentage of non-loop branches). Based on Ball and Larus.[2]

Program	% of All Branches	Heuristics	Perfect	Random	Big (%)	Program	% of All Branches	Heuristics	Perfect	Random	Big (%)
gcc	73	37	11	50	0 (0)	poly	20	40	3	31	3 (54)
lcc	71	32	12	52	1 (13)	fpppp	86	42	9	41	0 (0)
qpt	70	26	9	52	0 (0)	costScale	71	29	21	49	6 (52)
compress	66	40	18	66	6 (69)	doduc	52	33	3	49	0 (0)
xlisp	62	28	7	50	0 (0)	tomcatv	38	2	0	50	2 (98)
addalg	52	43	30	43	7 (67)	dcg	21	15	4	46	4 (51)
ghostview	52	16	4	47	4 (53)	spice2g6	21	36	8	52	2 (27)
eqntott	49	50	25	50	2 (92)	sgefat	18	26	8	61	8 (73)
rn	48	34	1	51	3 (25)	dnasa7	10	32	4	55	4 (58)
grep	44	1	0	3	3 (96)	matrix300	4	33	0	66	3 (99)
congress	40	28	3	57	2 (10)	Mean		29	10	49	
espresso	37	26	13	42	3 (24)	Std.Dev.		12	8	13	
awk	29	14	3	57	4 (29)						

A more detailed analysis of this heuristic approach can be found in Deitrich.[14] An approach that looked at source code was investigated by Sokolova,[46] who reported some improvements. Calder[5] trained a neural network, using a corpus of programs, to infer the branching behavior of new programs (they called the approach *evidence-based* static prediction).

Conditional instructions

conditional instructions

The extent to which large sophisticated branch prediction circuitry can be incorporated into processors used in resource limited environments is limited by the significant amount of power it consumes (up to 10% of a processors total dynamic power dissipation,[34] which drains batteries and heats up the device). An alternative solution to the bottleneck created by conditional branches is to do away with them. Some processors (e.g., ARM[1] and Intel Itanium[9,31,50]) achieve this by making the execution of all instructions conditional. This is implemented by having a sequence of bits in the instruction encoding represent various conditions. During program execution fetched instructions are only executed if the condition encoded in their bit sequence is true. Conditional execution of instructions can be more efficient than using a conditional branch instruction when there are only a few *nulled* instructions (representing the unexecuted arm). Since the number of instructions in the arms of **if** statements is often small, use of such instructions can often be worthwhile ([9] gives empirical results).

The following discussion and example is based on one appearing in the Intel XScale Microarchitecture Programmers Reference Manual.[26]

```
1  int f(int x)
2  {
3  if (x > 10)    /* if-cond   */
4     return 0;   /* if-stmt   */
5  else
6     return 1;   /* else-stmt */
7  }
```

The unoptimized machine code generated, using branch instructions, for the **if** statement usually has the form:

```
cmp  r0, 10
ble  L1
mov  r0,  0
```

```
    b    L2
L1: mov  r0,  1
L2:
```

while that generated using conditional instructions has the form:

```
    cmp   r0, 10
    movgt r0,  0
    movle r0,  1
```

Intel recommend[26] using the conditional instruction form when:

$$N1_C \times \frac{P1}{100} + N2_C \times \frac{100 - P1}{100} \leq N1_B \times \frac{P1}{100} + N2_B \frac{100 - P1}{100} + \frac{P2}{100} \times 4 \quad (1739.1)$$

where:

$N1_B$: Number of cycles to execute the *if_stmt* assuming the use of branch instructions.

$N2_B$: Number of cycles to execute the *else_stmt* assuming the use of branch instructions.

$P1$: Percentage of times the *if_stmt* is likely to be executed.

$P2$: Percentage of times a branch misprediction penalty is likely to be incurred.

$N1_C$: Number of cycles to execute the *if-else* portion using conditional instructions assuming *if-cond* to be true.

$N2_C$: Number of cycles to execute the *if-else* portion using conditional instructions assuming the *if-cond* to be false.

The code generated above requires three cycles to execute the *else-stmt* and four cycles for the *if-stmt*, assuming best-case conditions and no branch misprediction penalties.

Coding Guidelines

1 Introduction

conditional statement

Source code comprehension that involves a controlling expression of an **if** statement can have many facets. Some of these include the following:

- *Comprehending the conditional expression itself.* In particular the conditions under which it is true and false. These issues are discussed in detail in this subclause.
- *Interpreting the conditional expression within the application domain.* A conditional expression is used to make one of two choices and these may be driven by applications requirements (e.g., `line_len < MAX_LINE_LEN` might be the implementation of a requirement on the maximum number of characters that can appear on an output line). These application mapping comprehension issues are outside the scope of these coding guideline subsections and are not discussed further here.
- *Control flow dependencies.* An **if** statement may be nested within other **if** statements. The conditions represented by the additional controlling expressions may need to be taken into account,
- *Data dependencies.* When reading statements whose execution is directly affected by the flow of control of the conditional expression (e.g., an assignment statement in one of the arms of an **if** statement). These statements may need to be processed in the context of the conditions interpretation with the application or may involve operands that appear in the condition. For instance, in:

```
1  if ((x >= y) && (x <= z))
2     {
3     p++;
4     q=z-y;
5     ...
```

the reason for p being incremented is likely to be connected to what the condition represents within the application domain, while some information about the assignment to q can be obtained directly from the condition expression (e.g., it is always positive). A data dependency can also exist between an **if** statement and another such statement executed before it.

```
1  if (x == 2)
2     y = 45;
3
4  if (y != 45)
5     {
6     /* x does not equal 2 here */
7     }
```

Some coding guideline documents recommend a value for the maximum nesting of **if** statements (sometimes giving a rationale based on the limits of human short-term memory). Human short term memory does place constraints on the complexity of code than can be easily comprehended. However, while the time taken to comprehend a sequence of nested **if** statements may not feel like very long, it is much longer than the period of time over which short-term memory operates (a guideline recommendation based purely on human short-term memory capacity would probably need to limit nesting to a single level).

memory developer Miller 7±2

Any guideline recommendation designed to deal with human cognitive capacity limits needs a reasonably accurate model of how the people involved are likely to process a given construct. Given that the controlling expressions of **if** statements can have significantly different complexities and couplings to other statements it is necessary to calculate how readers are likely to interpret and chunk the information present in them (i.e., simply counting nesting is much too simplistic). At the time of this writing it is not possible to perform this calculation. Like all guideline recommendation it is also necessary to consider what alternative constructs developers might use. In the case of **if** statements some of the issues include the following:

- Many *alternative* techniques don't remove the nesting, they simply rearrange the source. For instance, replacing the inner most nested **if** statements by a call to a function that contains them only reduces nesting depth because of the accounting practices used to measure depth (the measure is usually based on the contents of a single function; the contents of called functions are not examined).
- The implicit **if** statements implied by the use of the logical operators in a conditional expression. For instance, `if ((a < 1) && (b > 10))` is usually counted as a single **if** statement, while the equivalent form `if (a < 1) if (b > 10)` is counted as two (there have been few psychological studies comparing how people treat rephrasing of these two forms; one study[39] found no psychological equivalence).

The rest of the discussion in this coding guideline subsection deals with the reader comprehension aspects of the dependencies between a controlling expression and the rest of the source code that contains it (other dependency issues are discussed elsewhere; while there has been a lot of talk about the software engineering costs associated with dependencies there has been little experimental research on the topic). In particular it discusses studies of human performance in reasoning tasks. It is possible to draw many parallels between solving these tasks and comprehending the use of, and references to, conditional expressions. These coding guidelines make the assumption that the pattern of mistakes made by developers will be the same as those made by the subjects used in these studies and that the factors found to affect subjects performance will be the same as those that affect developers performance. This is a big assumption, because most studies of reasoning aim to uncover some aspect of *natural* human thinking and thus avoid using subjects who have had training in the formal (i.e., mathematical) use of logical. This contrasts with developers who will have had some training in the use of mathematical logic and a significant amount of reasoning experience through attempts to comprehend of code. The assumption stated above implies that the extent to which developers are more adept at deductive reasoning, compared to the general undergraduate population used in the studies (which may be different from the general population[47]), is purely due to experience (formal training on its

function definition syntax

own does not usually affect performance; a study[8] of students who had taken a course in logic showed only a 3% improvement in performance, compared to those who had no formal logic training).

The three main topics discussed are:

memory
developer

1. use of working memory resources (for simplicity the use of long-term memory is not considered here),
2. the impact of the visual appearance of the source, and
3. the performance of humans when reasoning.

At the time of this writing the research results discussed raise questions, rather than providing answers. However, they do go a long way towards dispelling the notion that developer reasoning performance is independent of the logical form of a problem.

2 Remembering conditional information

How is the information denoted by a conditional expression represented in a readers mind? There are a number of possibilities, including:

- It is not represented at all. The source containing the controlling expression is reread on an as-needed basis.
- As some representation of the conditional expression appearing in the visible source. For instance, the spoken form of the expression (in which case the use of identifiers having shorter spoken forms would be an advantage).

```
1  if (character_count > 20)
2
3  if (char_count > 20) /* Contains fewer syllables, in spoken form, than above. */
```

- as what it represents internally, in the program, or what it represents within the application. For instance, the following two conditional expressions may both be represented in a readers mind using the same information.

```
1  if (character_count > MARGIN_WIDTH)
2
3  if (SKIPPED_PAST_MARGIN)
```

- some combination of the above. For instance, the conditional expression:

```
1  if ((ch >= MIN_PRINTABLE) && (ch <= MAX_PRINTABLE) &&
2      (ch != 'Q'))
```

might be represented, in a readers mind, as “a printable character that is not equal to the letter Q”.

3 Visual appearance

The bodies of **if** statements are commonly indented, visually, to the right of the **if** keyword. The greater the nesting of **if** statements the greater the indentation. This indenting practice has a cost impact that is separate from the cost of comprehending any relationship between the controlling expressions; the cost factors include:

statement
visual layout

- Reducing the visible line length reduces the probability that individual statements will fit on a single line. The issue of the readability of statements split over more than one line is discussed elsewhere.
- Each indentation creates a visually discriminable location that needs to be temporarily remembered by readers. Your author could not find any studies relating to this problem (the study by Hake[23] asked subjects to judge discriminate the position of a pointer on a scale, relative to two end points, using 5, 10, 20 and 50 different locations).

Situations requiring many that levels of nested **if** statements be written also often involve relatively large numbers of statements. Separating out the individual contributions made by indentation and number of lines (the issue of function size is discussed elsewhere) to the total comprehension effort, and being able to calculate the cost/benefit of the various possible source code organizations, requires a significantly greater amount of expertise than is currently available. For this reason these coding guidelines do not discuss these issues further. The issue of the visual layout of conditional expressions is discussed elsewhere.

function definition syntax
expressions

4 Reasoning

It has been claimed that the ability to reason is what separates humans from the rest of the animal kingdom. However, studies[43] of reasoning using illiterate subjects from remote parts of the world obtained answers to verbal reasoning problems that were based on personal experience and social norms, rather than the western ideal of logic. The answers given by subjects, in the same location, who had received several years of schooling were much more likely to match those demanded by mathematical logic; the subjects had learned to *play the game*. Peng, Ames, and Knowles[36] discuss styles of reasoning used in various cultures.

Until relatively recently (Wason's famous four card selection task[52] was first published in 1966; he did not at first question whether logic was the correct normative theory and interpreted the results in terms of people being illogical or irrational) it was believed that mathematical logic formed the basis for human rational thought (a belief in line with the model of the human mind as a general purpose computer).

An example of how a reader's performance can be affected by the kind of question asked, about conditions, is provided by a study by Bell and Johnson-Laird.[3] They asked subjects to give yes/no responses to two kinds of questions, asking about what is possible or what is necessary. They predicted that subjects would find it easier to infer a 'yes' answer to a question about what is possible, compared to one about what is necessary, because only one instance needs to be found, whereas all instances need to be checked to answer 'yes' to a question about necessity (see Table 1739.2). For instance, in a game in which only two can play:

```
If Allan is in then Betsy is in.
If Carla is in then David is out.
```

answering 'yes' to the question "Can Betsy be in the game?" (a possibility) is easier than giving the same answer to "Must Betsy be in the game?" (a necessity).

However, subjects would find it easier to infer a 'no' answer to a question about what is necessary, compared to one about what is possible, because only one instance needs to be found, whereas they all instances need to be checked to answer 'no' to a question about possibility. For instance, in another two person game:

```
If Allan is out then Betsy is out.
If Carla is out then David is in.
```

answering 'no' to the question "Must Betsy be in the game?" (a necessity) is easier than giving the same answer to "Can Betsy be in the game?" (a possibility).

Table 1739.2: Percentage of correct responses given to the four kinds of questions. Adapted from Bell and Johnson-Laird.[3]

Kind of Question	Correct 'yes' Response	Correct 'no' Response
is possible	91%	65%
is necessary	71%	81%

The study of reading and representing natural language sentences has to deal with the fact that such sentences in such languages are often syntactically and semantically ambiguous. Conditional statements written in the C language have a well defined syntactic and semantic meaning (we ignore the ill-formed cases here). However, the existence of a well defined meaning does not imply that all readers will find and use it.

5 Deductive reasoning

deductive reasoning

Heuristics and Biases

conjunction fallacy

pragmatic interpretation

A number of reasons for people's failure to give answers that matched those required by one of the mathematical logics (e.g., propositional or predicate calculus) have been proposed. These include the use of heuristics as a means of overcoming cognitive limits, interpreting the wording of questions in a pragmatic way based on how natural language are used rather than as logical formula, and interpreting the questions in a social context.[24] The following are some of the factors that have been found to affect peoples performance in solving deductive reasoning problems:

- *age of reasoner*— Performance in reasoning tasks declines with age.[19] Contributing factors for this decline include a reduction in working memory capacity with age[21] and a slowing down of cognitive processing speed.[42]
- *Belief bias*— people have been found to be more willing to accept a conclusion, derived from given premises, that they believe to be true than one that they believe to be false. A study by Evans, Barston, and Pollard[18] gave subjects two premises and a conclusion and asked them to state whether the conclusion was true or false (based on the premises given). The conclusions were rated as either believable or unbelievable (this status was checked by asking a separate group of subjects to rate the believability of the conclusions on a seven point scale). The results showed (see Table 1739.3) that belief did affect performance, particularly when the conclusion was invalid.

Table 1739.3: Percentage of subjects accepting that the stated conclusion could be logically deduced from the given premises. Based on Evans, Barston, and Pollard.[18]

Status-context	Example	Conclusion Accepted
Valid-believable		
	No Police dogs are vicious Some highly trained dogs are vicious Therefore, some highly trained dogs are not police dogs	88%
Valid-unbelievable		
	No nutritional things are inexpensive Some vitamin tablets are inexpensive Therefore, some vitamin tablets are not nutritional things	56%
Invalid-believable		
	No addictive things are inexpensive Some cigarettes are inexpensive Therefore, some addictive things are not cigarettes	72%
Invalid-unbelievable		
	No millionaires are hard workers Some rich people are hard workers Therefore, some millionaires are not rich people	13%

- *Form of premise*— a study by Dickstein[15] measured subjects performance on the 64 possible two premise syllogisms (a premise being one of the propositions: *All S are P*, *No S are P*, *Some S are P*, and *Some S are not P*). For instance, the following syllogisms show the four possible permutations of three terms (the use of S and P is interchangeable):

```
All M are S     All S are M     All M are S     All S are M
No P are M      No P are M      No M are P      No M are P
```

  The results showed that performance was affected by the order in which the terms occurred (known as the *figure*) in the two premises of the syllogism. The order in which the premises are processed may affect the amount of working memory needed to reason about the syllogism, which in turn can affect human performance.[22]

- *Individual preferences*— different people have been found to prefer the use of different strategies when solving a deductive problem.

reasoning 1739 strategy choice

Whatever the reason for people giving the answers they do, studies have shown that there are patterns to the *mistakes* made and that these patterns are found in a large number of people. A variety of theories and models have been proposed to explain these patterns of behavior. The following is a brief description of some of the different theories that have been proposed (the *mental model* theory currently enjoys general acclaim, based on its ability to predict the behavior seen in a large number of studies and the number of researchers currently publishing papers based on some form of it):

- *Mental logic.*[4, 40] People perform logical reasoning by the application of formal (syntactic) rules. The larger the number of rules that are applied to infer a conclusion the more difficult the problem is.

- *Mental models.*[27, 28] Readers construct a mental model (or set of models) from the given premises (this is a two-stage process that involves comprehending a premise, followed by combining the information contained in each premise to form the model). This model is used to draw possible conclusions (i.e., people reason from the content of a problem, unlike mental logic where reasoning is based on the syntactic form) which are then subject to a process of validation. Validation involves searching for alternative models, or counter examples, that are consistent with the premises, but which refute the conclusion. A conclusion is considered to be valid if no counter example can be found. Errors in reasoning are the result of limitations in the processing ability of the mind, in particular:

  1. one model is better than many. That is, the fewer models needed for an inference, and the simpler they are, the easier the inference,
  2. reasoners sometimes fail to consider all models in multiple-model problems. This results in them drawing conclusions that are possible rather than necessary,
  3. reasoners focus on what is true and neglect what is false. This can result in illusionary inferences,
  4. premise contents and background knowledge can affect the interpretation of assertions made in a premise and the process of reasoning, and
  5. with experience, reasoners develop tailor-made strategies for particular sorts of problems. To refute invalid conclusions, they can search for counter-examples.

- *Darwinian algorithms.*[12] It is argued that being able to perform certain kinds of conditional reasoning offers an evolutionary advantage and that the human mind has innate rule structures for dealing with these kinds of problems. For instance, one of the basis for social exchange is the generic rule *If you take a benefit, you pay a cost.*

- *Information gain.*[33] Here reasoners are said to have the goal of gaining information or reducing uncertainty (an adaption to the environment in which people traditionally have use reasoning). Conclusions are informative to the extent that they are improbable, or surprising.

- *Pragmatic reasoning schemas.*[7] These schemas are packages of knowledge about specific domains, containing rules for thought and action. Solving a particular problem requires matching production rules that evoke the appropriate schema.

- *Dual process theories.*[45, 47] People use two systems of reasoning (see Table 1739.4). Stanovich[47] has shown a strong correlation between students performance on SAT tests (as well as intelligence tests) and the extent to which they are likely to use System 2 in reasoning tasks. The results of some studies[17] have suggested that people sometimes use both systems of reasoning when reasoning about a problem.

Table 1739.4: Properties of the two systems of thinking. Based on Stanovich.[47]

System 1	System 2
Unconscious	Conscious
Automatic	Controlled
Associative	Rule-based
Heuristic processing	Analytic processing
Undemanding of cognitive capacity	Demanding of cognitive capacity
Relatively fast	Relatively slow
Acquisition by biology, exposure, and personal experience	Acquisition by cultural and formal training
Highly contextualized	Decontextualized
Conversational and socialized	Asocial
Independent of general intelligence	Correlated with general intelligence

Deductive reasoning problems can take many forms and this discussion limits itself to syllogisms (the simplest form and the one used by most studies). A syllogism consists of two premises and a conclusion. The two premises are usually given and the conclusion has to be deduced from them. The following are some of the different forms of syllogism:

- *Categorical syllogisms* use relations of inclusion. For instance, from

```
All A are B
All B are C
```

  we can deduce that "All A are C".

- *Linear syllogisms* use relations of order. For instance, from

```
A is taller than B
B is taller than C
```

  we can deduce that "A is taller than C".

- *Conditional syllogisms* use relations of implication. For instance, from

```
if A, then B
A is true
```

  we can deduce that "B is true".

These coding guidelines are aimed at the C programming language, which does not directly support operators for the operations that occur in categorical syllogisms (e.g., *all* or *some*). While developers may implement functions that perform equivalent operations these coding guidelines do not attempt to address developer defined operations. For this reason categorical syllogisms are not discussed further.

5.1 Linear reasoning

linear reasoning
relational operators
syntax

The use of relational operators have an obvious interpretation in terms of linear syllogisms. A study by De Soto, London, and Handel[13] provides a good example. The task they used was based on what they called *social reasoning*, using the relations *better* and *worse*. Subjects were shown two premises, involving three people, and a possible conclusion (e.g., "Is Mantle worse than Moskowitz?"). They had 10 seconds to answer *yes*, *no*, or *don't know*. All four possible combinations of conclusions were used.

Table 1739.5: Eight sets of premises describing the same relative ordering between A, B, and C (peoples names were used in the study) in different ways, followed by the percentage of subjects giving the correct answer. Adapted from De Soto, London, and Handel.[13]

	Premises	Percentage Correct Response		Premises	Percentage Correct Response
1	A is better than B B is better than C	60.5	5	A is better than B C is worse than B	61.8
2	B is better than C A is better than B	52.8	6	C is worse than B A is better than B	57.0
3	B is worse than A C is worse than B	50.0	7	B is worse than A B is better than C	41.5
4	C is worse than B B is worse than A	42.5	8	B is better than C B is worse than A	38.3

Based on the results (see Table 1739.5) De Soto et al made two observations (which they called *paralogical principles*; cases 5 and 6 possess both, while cases 7 and 8 possess neither):

1. People learn orderings better in one direction than another. In this study people gave more correct answers when the direction was better-to-worse (case 1), than mixed direction (case 2, 3), and were least correct in the direction worse-to-better (case 4). This suggests that use of the word *better* should be preferred over *worse* (the British National Corpus[30] lists *better* as appearing 143 times per million words, while *worse* appears under 10 times per million words and is not listed in the top 124,000 most used words).
2. People end-anchor orderings. That is, they focus on the two extremes of the ordering. In this study people gave more correct answers when the premises stated an end term (better or worse) followed by the middle term, than a middle term followed by an end term.

A second experiment in the same study gave subjects printed statements about people. For instance, "Tom is better than Bill". The relations used were *better*, *worse*, *has lighter hair*, and *has darker hair*. The subjects had to write the peoples names in two of four possible boxes; two arranged horizontally and two arranged vertically.

The results showed 84% of subjects selecting a vertical direction for better/worse, with better at the top (which is consistent with the *up is good* usage found in English metaphors[29]). In the case of lighter/darker 66% of subjects used a horizontal direction, with no significant preference for left-to-right or right-to left.

A third experiment in the same study used the relations *to-the-left* and *to-the-right*, and *above* and *below*. The above/below results were very similar to those for better/worse. The left-right results showed that subjects learned a left-to-right ordering better than a right-to-left ordering.

The results of this study show the affect that operand order has on the accuracy of peoples responses. However, the interpretation placed on the operator also plays a significant role. It appears that without knowing what interpretation a reader is likely to give to the operands and operators in the following two (logically equivalent) conditional expressions, for instance, it is not possible to select the one that is most likely to minimize incorrect reasoning on the part of readers.

```
1  if ((x <= y) && (x => z))
2  if ((x >= z) && (x <= y))
```

Since the De Soto, et al. study additional factors have been discovered and a number of models have been proposed to explain the strategies used by people in solving linear reasoning problems. These include:

- *The spatial model*[13, 25]— people integrate information from each premise into a spatial array representing all known relationships.

- *The linguistic model*[10]— people represent each premise using linguistic propositions (the individual premises are not integrated).
- *The algorithmic model*[38]— people apply some algorithm to the structure of the linguistic representation of the premises. For instance, given "Reg is taller than Jason; Keith is shorter than Jason" and the question "Who is the shortest?", a so called *elimination strategy* was used by some subjects in the study. (The answer for the first premise is Jason, which eliminates Reg; the answer to the second premise is Keith which eliminates Jason, so Keith is the answer).
- *The mixed model*[48]— the information in the premise is first decoded into a linguistic form and then encoded into a spatial form.

reasoning strategy choice

The strategy used to solve a given problem has been found to vary between people. A study by Sternberg and Weil[49] found a significant interaction between a subjects' aptitude (as measured by verbal and spatial ability tests) and the strategy they used to solve linear reasoning problems. However, a person having high spatial ability, for instance, does not necessarily use a spatial strategy. A study by Roberts, Gilmore, and Wood[41] asked subjects to solve what appeared to be a spatial problem (requiring the use of a very inefficient spatial strategy to solve). Subjects with high spatial ability used non-spatial strategies, while those with low spatial ability used a spatial strategy. The conclusion made was that those with high spatial ability were able to see that the spatial strategy was inefficient to select as alternative strategy, while those with less spatial ability were unable to perform this evaluation.

A study by Mayer[32] asked subjects to learn a sequence of facts expressed as a relationship, such as $B > C$, $A > B$, and $A > C$ (the ordering $A > B > C > D > E > F$ was true for all facts the subjects were asked to learn). One group of subjects were told to think of the operands as boys names and the relation as representing *taller than*, while the other group were given no such instruction. Once the subjects had learned the facts presented to them, they were tested by having to answer questions about them. The results suggested that subjects who had been told to think about the relationship as representing *taller than* had integrated the separately presented facts into a single linear-encoding. While the other subjects did not integrate the separate facts and unencoded each fact using a single association.

The information encoding used by people can affect how well they later recall that information and also their performance when making comparisons about objects or symbols having some attribute that varies along a continuem. For instance, studies[37] have found that subjects performance improves the further apart the two objects being compared are perceived to be.

symbolic distance effect

5.2 Causal reasoning

A question often asked by developers, while reading source, is "what causes this event/situation to occur?" Causal questions such as this are also common in everyday life. However, there has been little mathematical research (statistics deals with correlation) on causality (Pearl[35] is an exception) and little psychological research on causal reasoning. It is often possible to express a problem in either a causal or conditional form. A study by Sloman, and Lagnado[44] gave subjects one of the following two reasoning problems and associated questions:

- *Causal argument* form:

```
A causes B
A causes C
B causes D
C causes D
D definitely occurred
```

  with the questions: "If B had not occurred, would D still have occurred?", or "If B had not occurred, would A have occurred?".

- *Conditional argument* form:

```
If A then B
If A then C
If B then D
If C then D
D is true
```

with the questions: "If B were false, would D still be true?", or "If B were false, would A be true?".

The results (see Table 1739.6) showed that subject performance depended on the form in which the problem was expression.

Table 1739.6: Percentage *yes* responses to various forms of questions (based on 238 responses). Based on Sloman, and Lagnado.[44]

Question	Causal	Conditional
D holds?	80%	57%
A holds?	79%	36%

There has been relatively little psychological research into causality and counter factual reasoning, although this is starting to change. This subsection is intended to show that human performance with this kind of reasoning may not be the same as that when using conditional reasoning.

5.3 Conditionals in English

In all natural languages, the conditional clause generally precedes the conclusion, in a conditional statement.[11] An example where the conditional follows the conclusion is "I will leave, if you pay me" given as the answer to the question "Under what circumstances will you leave?". In one study of English[20] the conditional preceded the conclusion in 77% of written material and 82% of spoken material.

Table 1739.7: Occurrence of the most common conditional sentence types in speech (266 conditionals from a 63,746 word corpus) and writing (948 conditionals from 357,249 word corpus). In the notation *if + x, y*: *x* is the condition (which might, for instance, be in the past tense) and *y* can be thought of as the *then part* (which might, for instance, use one of the words would/could/might, or be in the present tense). Adapted from Celce-Murcia.[6]

Structure	Speech	Writing
If + present, present	19.2	16.5
If + present, (will/be going to)	10.9	12.5
If + past, (would/might/could)	10.2	10.0
If + present, (should/must/can/may)	9.0	12.1
If + (were/were to), (would/could/might)	8.6	6.0
If + (had/have +en), (would/could/might) have	3.8	3.3
If + present, (would/could/might)	2.6	6.1

Table 1739.8: Occurrence of various kinds of **`if`** statement controlling expressions (as a percentage of all **`if`** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), *`integer-constant`* is an integer constant expression, and *expression* represents all other expressions. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	32.4	! function-call	3.8
object	15.5	object < *`integer-constant`*	2.2
object == object	8.9	object > *`integer-constant`*	1.8
! object	7.4	function-call == object	1.6
function-call	7.4	object > object	1.4
expression	5.7	object != *`integer-constant`*	1.3
object != object	4.2	function-call == *`integer-constant`*	1.2
object == *`integer-constant`*	4.0	object < object	1.1

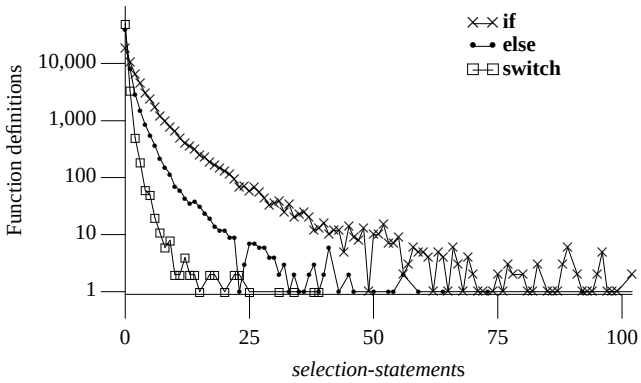

Figure 1739.1: Number of function definitions containing a given number of `selection-statement`s. Based on the translated form of this book's benchmark programs.

Table 1739.9: Occurrence of various kinds of **`switch`** statement controlling expressions (as a percentage of all **`switch`** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`), `integer-constant` is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%
object	75.3
function-call	14.2
expression	5.2
others	3.3
`*v` object	2.0

Table 1739.10: Occurrence of equality, relational, and logical operators in the conditional expression of an **`if`** statement (as a percentage of all such controlling expressions and as a percentage of all occurrences each operator in the source). Based on the visible form of the `.c` files. The percentage of controlling expressions may not sum to 100% because more than one of the operators occurs in the same expression.

Operator	% Controlling Expression	% Occurrence of Operator	Operator	% Controlling Expression	% Occurrence of Operator
==	31.7	88.6	>=	3.5	76.8
!=	14.1	79.7	no relational/equality	47.5	—
<	6.9	45.6	||	9.6	85.9
<=	1.9	68.6	&&	14.5	82.3
>	3.5	84.9	no logical operators	84.2	—

Semantics

controlling expression
if statement

A selection statement selects among a set of statements depending on the value of a controlling expression. 1740

Commentary

terms
defined where

The term *controlling expression* does not appear in italic type in the Standard. This C sentence might be considered to be its first definition.

C++

The C++ Standard omits to specify how the flows of control are selected:

6.4p1

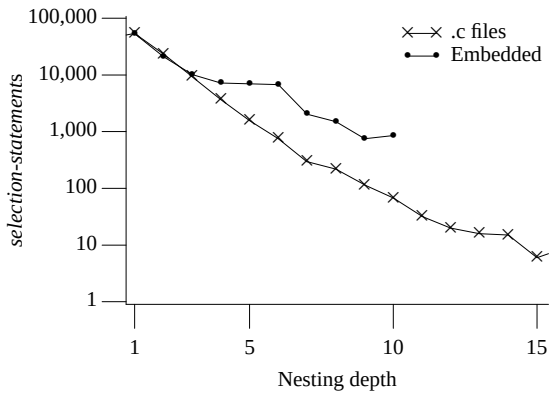

Figure 1739.2: Number of `selection-statement`s having a given maximum nesting level for embedded C[16] (whose data was multiplied by a constant to allow comparison; the data for nesting depth 5 was interpolated from the adjacent points). Based on the visible form of the `.c` files.

Selection statements choose one of several flows of control.

Coding Guidelines

The evaluation of a controlling expression is used to select the flow of control. The issue of side effects occurring during the evaluation is frequently discussed by developers and writers of coding guideline documents. Experience suggests that the following are the primary reasons for developers to write controlling expressions containing side effects:

- A belief that the resulting machine code is more efficient. In many cases this belief is false. For instance, most translators generate the same machine code generated for:

```
1  if (x = y)
```

  and:

```
1  x=y;
2  if (x != 0)
```

- Reducing the effort needed to organize the layout of the visible source, when writing it. For instance, an assignment inside a series of nested **`if`** statements would require the use of braces:

```
1  if (x = y)
2     /* ... */
```

  might have to be written as:

```
1  {
2  x=y;
3  if (x != 0)
4     /* ... */
5  }
```

Neither of these reasons could be said to contain an actual benefit. The cost associated with side effects in controlling expressions is the possibility that they will go unnoticed by a reader of the source (especially if scanning along the left edge looking for assignments).

reading
kinds of

The most common form of side effect in a controlling expression is assignment, in particular simple assignment. The case where the author of the code intended to type an equality operator, rather than a simple

guidelines not faults

controlling expression if statement 1740

assignment operator is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. However, it is possible that a reader of the visible source will mistake a simple assignment for an equality operator (the token **==** is much more likely than **=** in the context of a controlling expression) and reducing the likelihood of such a mistake occurring is also a cost reduction.

This discussion has referred to controlling expressions as if these costs and benefits apply to their use in all contexts (i.e., selection and iteration statements). The following example shows that writing code to avoid the occurrence of side effects in controlling expressions contained with iteration statements requires two, rather than one, assignments to be used.

```
1  extern int glob_1,
2             glob_2;
3
4  void f_1(void)
5  {
6  if (glob_1 = glob_2)
7     ;
8  while ((glob_1 = glob_2 + 1) != 3)
9     { /* ... */ }
10 }
11
12 void f_2(void)
13 {
14    {
15    glob_1 = glob_2;     /* Single statement. */
16    if (glob_1 != 0)
17       ;
18    }
19
20 glob_1 = glob_2 + 1;    /* Statement 1: always occurs. */
21 while (glob_1 != 3)
22    {
23    /* ... */
24    glob_1 = glob_2 + 1; /* Statement 2: occurs after every iteration. */
25    }
26 }
```

Duplicating the assignment to `glob_1` creates a maintenance dependency (any changes to one statement need to be reflected in the other). The increase in cost caused by this maintenance dependency is assumed to be greater than the cost reduction achieved from reducing the likelihood of a simple assignment operator being mistaken treated as an equality operator.

Cg 1740.1 The simple assignment operator shall not occur in the controlling expression of an **if** statement.

Experience has shown that there are a variety of other constructs, appearing in a controlling expression, that developer have difficulty comprehending, or simply miscomprehend when scanning the source. However, no other constructs are discussed here. The guideline recommendation dealing with the use of the assignment operator has the benefit of simplicity and frequency of occurrence. It was difficult enough analyzing the cost/benefit case for simple assignment and others are welcome to address more complicated cases.

Experience shows that many developers use the verbal form "if expression is not true then" when thinking about the condition under which an **else** form is executed. This use of *not* can lead to double negatives when reading some expressions. For instance, possible verbal forms of expressing the conditions under which the arms of an **if** statement are executed include:

```
1  if (!x)
2     a(); /* Executed if not x.              */
3  else
```

```
4      b(); /* Executed if not x is not true.   */
5           /* Executed if not x is equal to 0. */
6           /* Executed if x is not equal to 0. */
7
8   if (x != y)
9      c(); /* Executed if x is not equal to y.             */
10  else
11     d(); /* Executed if x is not equal to y is not true. */
```

The possible on linguistic impact of the `!` operator on expression comprehension is discussed elsewhere. ! operand type

Cg 1740.2

The top-level operator in the controlling expression of an **if** statement shall not be `!` or `!=` when that statement also contains an **else** arm.

If the value of the controlling expression is known a translation time, the selection statement may contain dead code and the controlling expression is redundant. These issues are discussed elsewhere. dead code
redundant code

Usage

In the translated form of this book's benchmark programs 1.3% of *selection-statement*s and 4% of *iteration-statement*s have a controlling expression that is a constant expression. Use of simple, non-iterative, flow analysis enables a further 0.6% of all controlling expressions to be evaluated to a constant expression at translation time.

1741 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. block
selection statement

Commentary

Rationale

```
enum {a, b};

int different(void)
{
    if (sizeof(enum {b, a}) != sizeof(int))
        return a;  // a == 1
    return b;  // which b?
}
```

In C89, the declaration `enum {b, a}` persists after the if-statement terminates; but in C99, the implied block that encloses the entire **if** statement limits the scope of that declaration; therefore the different function returns different values in C89 and C99. The Committee views such cases as unintended artifacts of allowing declarations as operands of cast and **sizeof** operators; and this change is not viewed as a serious problem.

See the following C sentence for a further discussion on the rationale. 1742 block
selection sub-statement

C90

See Commentary.

C++

The C++ behavior is the same as C90. See Commentary.

Coding Guidelines

Developers are more likely to be tripped up by the lifetime issues associated with compound literals than enumeration constants. For instance in:

```
1   if (f(p=&(struct S){1, 2}))
2      /* ... */
3   val=p->mem_1;
```

the lifetime of the storage whose address is assigned to `p` ends when the execution of the **`if`** statement terminates. Ensuring that developers are aware of this behavior is an educational issue. However, developers intentionally relying on the pointed-to storage continuing to exist (which it is likely to, at least until storage needs to be allocated to another object) is a potential guideline issue. However, until experience has been gained on how developers use compound literals it is not known whether this issue is simply an interesting theoretical idea of a real practical problem.

---

block selection sub-statement

Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement. 1742

Commentary

Rationale

A new feature of C99: A common coding practice is always to use compound statements for every selection and iteration statement because this guards against inadvertent problems when changes are made in the future. Because this can lead to surprising behavior in connection with certain uses of compound literals (§6.5.2.5), the concept of a block has been expanded in C99.

Given the following example involving three different compound literals:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    if (*(q = (int[2]){i, j}))
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    else
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);

    return *p;
}
```

it seemed surprising that just introducing compound statements also introduced undefined behavior:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    if (*(q = (int[2]){i, j})) {
        fn(p = (int[5]){9, 8, 7, 6, 5}, q);
    } else {
        fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
    }

    return *p;  // undefined--no guarantee *p designates an object
}
```

Therefore, the substatements associated with all selection and iteration statements are now defined to be blocks, even if they are not also compound statements. A compound statement remains a block, but is no longer the only kind of block. Furthermore, all selection and iteration statements themselves are also blocks, implying no guarantee that `*q` in the previous example designates an object, since the above example behaves as if written:

```
extern void fn(int*, int*);

int examp(int i, int j)
{
    int *p, *q;

    {
        if (*(q = (int[2]){i, j})) {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){9, 8, 7, 6, 5}, q);
        } else {
            // *q is guaranteed to designate an object
            fn(p = (int[5]){4, 3, 2, 1, 0}, q + 1);
        }
    }

    // *q is not guaranteed to designate an object

    return *p;  // *p is not guaranteed to designate an object
}
```

If compound literals are defined in selection or iteration statements, their lifetimes are limited to the implied enclosing block; therefore the definition of “block” has been moved to this section. This change is compatible with similar C++ rules.

C90

The following example illustrates the rather unusual combination of circumstances needed for the specification change, introduced in C99, to result in a change of behavior.

```
1  extern void f(int);
2  enum {a, b} glob;
3
4  int different(void)
5  {
6  if (glob == a)
7                        /* No braces.                                              */
8     f((enum {b, a})1); /* Declare identifiers with same name and compatible type. */
9
10 return b;  /* C90: numeric value 1 */
11            /* C99: numeric value 0 */
12 }
```

C++

The C++ behavior is the same as C90.

Coding Guidelines

Some coding guideline documents recommend that the block associated with both selection and iteration statements always be bracketed with braces (i.e., that it is always a compound statement). When the compound statement contains a single statement the use of braces is redundant and their presence decreases the amount of information visible on a display (the number of available lines is fixed and each brace usually occupies one line). However, experience has shown that in some cases the presence of these braces can:

using braces
block
compound statement
syntax

- Provide additional visual cues that can reduce the effort needed, by readers, to comprehend a sequence of statements. However, the presence of these redundant braces reduces the total amount of information immediately visible, to a reader, on a single display (i.e., the amount of source code that can be seen

cost/accuracy trade-off

without expending motor effort giving editor commands to change the display contents). The way in which these costs and benefits trade-off against each other is not known.

statement header

- Help prevent faults being introduced when code is modified (i.e., where a modification results in unintended changes to the syntactic bindings of blocks to statement headers). Experience shows that nested **if** statements are the most common construct whose modification results in unintended changes to the syntactic bindings of blocks.

In the following example the presence of braces provides both visual cues that the **else** does not bind to the outer **if** and additional evidence (its indentation provides counter evidence because it provides an interpretation that the intent is to bind to the outer **if**) that it is intended to bind to the inner **if**.

```
1  void f(int i)
2  {
3  if (i > 8)
4     if (i < 20)
5        i++;
6  else
7     i--;
8
9  if (i > 8)
10    {
11    if (i < 20)
12       i++;
13 else
14    i--;
15    }
16 }
```

Blocks occur in a number of statements, is there a worthwhile cost/benefit in guideline recommendation specifying that these blocks always be a compound statement?

The block associated with a **switch** statement is invariably a compound statement. A guideline recommendation that braces be used is very likely to be redundant in this case. Iteration statements are not as common as selection statements and much less likely to be nested (in other iteration statements) than selection statements (compare Figure 1739.2 and Figure **??**), and experience suggests developer comprehension of such constructs is significantly affected by the use of braces. Experience suggests that the nested **if** statement is the only construct where the benefit of the use of braces is usually greater than the cost.

Cg 1742.1

The statement forming the block associated with either arm of an **if** statement shall not be an **if** statement.

References

1. ARM. *ARM1026EJ-S: Technical Reference Manual*. ARM Limited, r0p1 edition, Dec. 2002.
2. T. Ball and J. R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, June 1993.
3. V. A. Bell and P. N. Johnson-Laird. A model theory of modal reasoning. *Cognitive Science*, 22(1):25–51, 1998.
4. M. D. S. Braine and D. P. O'Brian. A theory of *if*: A lexical entry, reasoning program, and pragmatic principles. *Psychological Review*, 98(2):182–203, 1991.
5. B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. G. Zorn. Corpus-based static branch prediction. *SIGPLAN Notices*, 30(6):79–92, June 1995.
6. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.
7. P. Cheng and K. J. Holyoak. Pragmatic reasoning schemas. *Cognitive Psychology*, 17:391–416, 1985.
8. P. Cheng, K. J. Holyoak, R. E. Nisbett, and L. M. Oliver. Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, 18:293–328, 1986.
9. Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 182–191. IEEE Computer Society, 2001.
10. H. H. Clark. Linguistic processes in deductive reasoning. *Psychological Review*, 76(4):387–404, 1969.
11. B. Comrie. Conditionals: A typology. In E. C. Traugott, A. T. Meulen, J. S. Reilly, and C. A. Furguson, editors, *On Conditionals*, chapter 4, pages 77–97. Cambridge University Press, 1986.
12. L. Cosmides. The logic of social exchange: Has natural selection shaped how humans reason? Studies with the Wason selection task. *Cognition*, 31:187–276, 1989.
13. C. B. De Soto, M. London, and S. Handel. Social reasoning and spatial paralogic. *Journal of Personality and Social Psychology*, 2(4):513–521, 1965.
14. B. L. Deitrich. *Static Program Analysis to Enhance Profile Independence in Instruction-Level Parallelism Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.
15. L. S. Dickstein. The effect of figure on syllogistic reasoning. *Memory & Cognition*, 6(1):76–83, 1978.
16. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.
17. J. S. B. T. Evans. Deciding before you think: Relevance and reasoning in the selection task. *British Journal of Psychology*, 87:223–240, 1996.
18. J. S. B. T. Evans, J. L. Barston, and P. Pollard. On the conflict between logic and belief in syllogistic reasoning. *Memory & Cognition*, 11(3):295–306, 1983.
19. J. E. Fisk and C. Sharp. Syllogistic reasoning and cognitive ageing. *The Quarterly Journal of Experimental Psychology*, 55A(4):1273–1293, 2002.
20. C. E. Ford and S. A. Thompson. Conditionals in discourse: A text-based study from English. In E. C. Traugott, A. T. Meulen, J. S. Reilly, and C. A. Furguson, editors, *On Conditionals*, chapter 18, pages 353–372. Cambridge University Press, 1986.
21. A. S. Gilinsky and B. B. Judd. Working memory and bias in reasoning across the life span. *Psychology and Ageing*, 9(3):356–371, 1994.
22. V. Girotto, A. Mazzocco, and A. Tasso. The effect of premise order on conditional reasoning: a test of the mental model theory. *Cognition*, 63:1–28, 1997.
23. H. W. Hake and W. R. Garner. The effect of presenting various numbers of discrete steps on scale reading accuracy. *Journal of Experimental Psychology*, 42(5):358–366, 1951.
24. D. J. Hilton. The social context of reasoning: Conversational inference and rational judgment. *Psychological Bulletin*, 118(2):248–271, 1995.
25. J. Huttenlocher. Constructing spatial images: A strategy in reasoning. *Psychological Review*, 75(6):550–560, 1968.
26. Intel. *Intel XScale Microarchitecture Programmers Reference Manual*. Intel, Inc, 2001.
27. P. N. Johnson-Laird. *Mental Models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983.
28. P. N. Johnson-Laird. Mental models and deduction. *TRENDS in Cognitive Science*, 5(10):434–442, 2001.
29. G. Lakoff and M. Johnson. *Metaphors We Live By*. The University of Chicago Press, 1980.
30. G. Leech, P. Rayson, and A. Wilson. *Word Frequencies in Written and Spoken English*. Pearson Education, 2001.
31. R. Leupers. Exploiting conditional instructions in code generation for embedded VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999*, 1999.
32. R. E. Mayer. Qualitatively different encoding strategies for linear reasoning premises: Evidence for single association and distance theories. *Journal of Experimental Psychology: Human Learning and Memory*, 5(1):1–10, 1979.
33. M. Oaksford and N. Chater. A rational analysis of the selection task as optimal data selection. *Psychological Review*, 101(4):608–631, 1994.
34. D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *Proceedings of Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 233–244, Feb. 2002.
35. J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.
36. K. Peng, D. R. Ames, and E. D. Knowles. Culture and human inference: Perspectives from three traditions. In D. Matsumoto, editor, *The Handbook of Culture and Psychology*, pages 243–263. Oxford University Press, Oct. 2001.
37. G. R. Potts. Storing and retrieving information about ordering relationships. *Journal of Experimental Psychology*, 103(3):431–439, 1974.
38. G. Quinton and B. J. Fellows. 'Perceptual' strategies in the solving of three-term series problems. *British Journal of Psychology*, 66:69–78, 1975.
39. J. Richardson and T. C. Ormerod. Rephrasing between disjunctives and conditionals: Mental models and the effects of thematic content. *The Quarterly Journal of Experimental Psychology*, 50A(2):358–385, 1997.
40. L. J. Rips. *The Psychology of Proof*. The MIT Press, 1994.

41. M. J. Roberts, D. J. Gilmore, and D. J. Wood. Individual differences and strategy selection in reasoning. *British Journal of Psychology*, 88:473–492, 1997.

42. T. A. Salthouse. The processing-speed theory of adult age differences in cognition. *Psychological Review*, 103(3):403–428, 1996.

43. S. Scribner. Modes of thinking and ways of speaking: culture and logic reconsidered. In P. N. Johnson-Laird and P. C. Wason, editors, *Thinking: Readings in Cognitive Science*, chapter 29, pages 483–500. Cambridge University Press, 1977.

44. S. Sloman and D. A. Lagnado. Counterfactual undoing in deterministic causal reasoning. In *Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society*, 2002.

45. S. A. Sloman. The empirical case for two systems of reasoning. *Psychological Bulletin*, 119(1):3–22, 1996.

46. S. Sokolova and D. R. Kaeli. Static branch prediction using high-level language. Technical Report Technical Report ECE-CEG-98-009, Northeastern University, 1998.

47. K. E. Stanovich. *Who Is Rational?* Lawrence Erlbaum Associates, 1999.

48. R. J. Sternberg. Representation and process in linear syllogistic reasoning. *Journal of Experimental Psychology: General*, 109(2):119–159, 1980.

49. R. J. Sternberg and E. M. Weil. An aptitude $\times$ strategy interaction in linear syllogistic reasoning. *Journal of Educational Psychology*, 72(2):226–239, 1980.

50. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, spru189f edition, Oct. 2000.

51. G.-R. Uh. *Effectively exploiting indirect jumps*. PhD thesis, Florida State University, 1997.

52. P. C. Wason. Reasoning. In B. M. Foss, editor, *New Horizons in Psychology 1*, chapter 6, pages 135–151. Penguin Books, 1966.

53. M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):667–697, 2002.

54. R. Yung. Evaluation of a commercial microprocessor. Technical Report SMLI TR-98-65, Sun Microsystems, 1998.