The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

5.1.2.3 Program execution

program execution
abstract machine C

The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant. 184

Commentary

The properties of this abstract machine are never fully defined in the standard. The term can be thought of as a conceptual model used to discuss C, not as a formal definition for a full-blown specification. Several research projects have produced formal specifications of substantial parts of Standard C (using operational semantics based on evolving algebras,[12] structural operational semantics,[23] and denotational semantics[24]). These have investigated the language only (no preprocessor and a subset of the library).

The semantic descriptions also ignore other possible ways of modifying values in storage; for instance, by irradiating the processor or storage chips, forming the computing platform, with nuclear or electromagnetic radiation.[11]

Other Languages

The definition of most computer languages is written in a form of stylized English. The Modula-2 Standard[16] was defined using a formal definition language[17] (which in turn is defined in terms of Zermelo-Fraenkel set theory; which in turn is based on nine axioms, {although only four of these have been proved to be consistent and independent}) and English. Neither language was given priority over the other. Any difference in meaning between the two formalisms has to be decided by a discussing the intended behavior, not by giving one set of wording preference over the other. The original definition of Lisp was written in terms of a subset of itself and the Prolog Standard[15] provides (in an informative annex) a formal semantics written in a subset of Prolog. A Model Implementation (an implementation that exactly implements all of the requirements of a language standard, irrespective of performance issues) was created for Pascal.[29]

Common Implementations

Most vendors regard optimizations as being very important and sometimes invest more effort on performing them than doing anything else.

A Model Implementation was produced for C.[20] This implementation aimed to mimic the behavior of the abstract machine and diagnose all implementation-defined, unspecified and undefined behaviors (as well as the usual constraint and syntax violations). It generated code for an abstract machine whose instruction set had a one-to-one mapping to C operations, the idea being that such a one-to-one mapping helped simplify code generation and reduce possible bugs. The interpreter for this abstract machine did all the pointer checks and uninitialized object checks that caused undefined behavior.

At the time of this writing there is one formally verified compiler[3] for a substantial subset of C.

Coding Guidelines

Many developers spend a relatively large amount of time trying to optimize their code. The extent to which their efforts have any effect is often marginal. There are a lot of misconceptions about how to write efficient C. By far the biggest improvements in performance come from design and algorithmic optimization. Trying to second guess what machine code a translator will generate requires a great deal of knowledge on the target architecture and the code-generation techniques used by an implementation, skills that very few developers have. The only effective way to tune at the statement level is by looking at the generated machine code. Changes based on ideas about what the translator *should do* are often wrong.

While the benefits of using formal methods when writing software are often promoted by their followers, the associated costs[9] and lack of any evidence that the benefits can be realized in practice[13] are rarely mentioned.

side effect

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,11) which are changes in the state of the execution environment. 185

Commentary

An operation, in a C program, that does not generate a side effect is, at the very least, considered to be of no practical interest.

190 redundant code

The state of the execution environment includes information on the current flow of control (nested function calls and the current statement being executed). This means that the control expression in, for instance, an `if` statement has the side effect of selecting the flow of control. Accessing a volatile object is not guaranteed to change the state of the execution environment, but a translator must act as-if it does. The C library treats all I/O operations as occurring on files, and therefore generating a side effect. The C++ Standard is more explicit in stating, 1.9, " . . . calling a library I/O function, . . . ".

Other Languages

Some languages, usually functional, have been specifically designed to be side effect free. The advantage of being side effect free is that it significantly simplifies the task of mathematically proving various properties about programs.

Common Implementations

Calling a function, or one of the memory allocation functions, also changes the state of the execution environment. This state is part of the housekeeping performed by an implementation and does not normally cause an external effect. Such operations only become noticeable if there are insufficient resources to satisfy them.

Coding Guidelines

C is what is known as an imperative language. The design of C programs (and nearly every commercially written program, irrespective of language used) is generally based on using side effects to implement the desired functionality. Coding guideline documents that recommend against the use of side effects (which they do with surprising regularity) are rather missing the point. Developers need to ensure that side effects occur in a predictable order. The relevant issues are discussed elsewhere.

expression order of evaluation

Comprehending source code requires readers to deduce the changes it makes to the state of the abstract machine. This process requires that readers remember, and later recall, a sequence of abstract machine states. Some of the issues involved in organizing changes of machine state into a meaningful pattern are discussed elsewhere, as are issues of locally minimizing the number of changes of state that need to be remembered.

statement syntax
postfix operator constraint

---

186 Evaluation of an expression may produce side effects.

Commentary

An expression may also be evaluated for its result value. This value is used to decide the flow of control, or returned as the value of a function call.

expressions

Operations on objects having floating-point type may also set status flags, which is a change of state of the execution environment.

196 footnote 11

Common Implementations

Some optimizers invoke the as-if rule to delay the updating of storage after the evaluation of an expression. The storage may not be necessary if the value can be kept in a register and accessed their for all subsequent accesses (or at least until it is modified again).

Coding Guidelines

One of the most common reader activities when performing source code comprehension is the analysis of the consequences of side effects in the evaluation of expressions. An expression that does not produce side effects, when evaluated, is redundant. The issue of redundant code is discussed elsewhere.

190 redundant code

Example

```
1 extern int glob;
2
```

```
3   void f(void)
4   {
5   glob+4;          /* No side effect. */
6
7   if (glob+4 == 0) /* Evaluation selects the flow of control. */
8      glob--;       /* A side effect. */
9   }
```

sequence points

At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. 187

Commentary

This defines the term *sequence points*. Sequence points are points of stability. At these points there are no outstanding modifications to objects waiting to be completed. The ordering of sequence points during program execution is not always guaranteed. The freedom given to translators in evaluating binary operands leaves open the possibility of there being more than one possible ordering of sequence points. The following are some of the more important orderings implied by sequence points:

sequence points 187

object initializer evaluated when

statement executed in sequence

comma operator syntax

- Declaration evaluation order.
- Statement execution order.
- A left then right evaluation ordering of operands of some binary operators.

In some situations the specification given in the standard has been found to be open to more than one interpretation. Various C Committee members are working on a more formal specification of the semantics of expressions. Several notations have been proposed (e.g., using sets or trees); the intent is to select the one that developers will find the simplest to use. Whether the final document will be published as a TR or in some other form has not yet been decided. The working papers are available via the WG14 Web site.

WG14/N899

> *To define a formalism for the semantics of expressions in the C language (as defined in ISO/IEC 9899:1999) to enable users of the Standard to determine unambiguously what expressions do or do not conform to the language and their level of conformance.*

An example of the nontrivial issues that surround the analysis of expressions that contain more than one sequence point is provided by DR #087. In the following code, is the behavior for the expressions in lines A, B, and C defined?

```
1   static int glob;
2
3   int f(int p)
4   {
5   glob = p;
6   return 0;
7   }
8
9   void DR_087(void)
10  {
11  int loc;
12
13  loc = (glob = 1) + (glob = 2);                /* Line C */
14  loc = (10, glob = 1, 20) + (30, glob = 2, 40); /* Line A */
15  loc = (10, f (1), 20) + (30, f (2), 40);       /* Line B */
16  }
```

In *Line C* there are two assignments to glob between two sequence points. The behavior is undefined. In *Line A* the assignments to glob are bracketed by sequence points. It might be thought that there is no possibility that the two assignments could both occur between the same pair of sequence points. However, there is no requirement that an operator be performed immediately after one or more of its operands is evaluated. One of the evaluation orders an implementation could choose is:

```
evaluate 10
sequence point
evaluate 30
sequence point
evaluate glob=1
evaluate glob=2
sequence point
evaluate 20
sequence point
evaluate 40
perform addition
perform assignment
sequence point
```

Just like the case in *Line C* there are two assignments to glob between two sequence points. The behavior is undefined.

function call interleave

In *Line B* there are two calls to the function f; however, calls to functions cannot be interleaved (although this requirement appears in a number of committee papers dealing with sequence point and is part of the question in DR #087, it has never been stated in any normative document). This means that the two assignments to glob, which occur in the two calls to f, can never occur between the same pair of sequence points.

Other Languages

Most languages do not provide the operators, available in C, for modifying objects within an expression. However, most languages do allow function calls within an expression and these can modify objects with file scope or indirectly through parameters. Some language definitions specify that changing the value of a variable in an expression, while it is being evaluated (i.e., via function call), always results in undefined behavior (or some such term). Very few languages discuss the interaction of side effects and sequence points (if any such concept is defined).

Common Implementations

The as-if rule can be invoked to allow code motion across sequence points. However, implementations have to ensure that the behavior remains unchanged (in terms of external output or behavior, if not execution-time performance). The only way for a developer to find out if an implementation is using the as-if rule to reorder side effects around sequence points is to examine the generated code. Many implementations do provide an option to list the machine code generated.

Most C expressions are very simple, offering little opportunity for clever optimization in themselves. Obtaining high-quality code requires finding similarities within expressions occurring in sequences of statements, that is across sequence points. Value numbering is a commonly used technique.[5]

Coding Guidelines

The sequence point definition suggests, on first glance, a well-defined ordering of events. In most cases the ordering is predictable, but in a few cases the ordering of sequence points is not unique. Some of the constructs that are sequence points can themselves be evaluated in different orders. The two sequence points that cannot have multiple orderings are the **;** punctuator at the end of a statement and initialization of block scope definitions, and the evaluation of a full expression that is also a controlling expression. The possible orderings of these sequence points is fully specified by the standard and cannot vary between implementations.

All other sequence points can occur within a subexpression that is the operand of a larger expression. A sequence point may guarantee an evaluation order within a subexpression. However, there might not be any guarantee that the subexpression is always evaluated at the same point in the evaluation of its containing full expression. For instance in:

```
1 f(c++) + g(c -= 3);
```

the sequence point before a ( ) operator is invoked guarantees that either c will be incremented before f is called, or that c will be incremented before g is called. The standard does not specify which function should be called first, so there are two possible orderings of sequence points. In:

```
1 (x--, p = x+y) + (y++, q = x+y)
```

the sequence point on the comma operator guarantees that its left operand will be evaluated and all side effects will take place before the right operand is evaluated. But nothing is said about which operand of the plus operator is evaluated first.

Cg 187.1

The value of a full expression shall not depend on the order of evaluation of any sequence points it contains.

Cg 187.2

Any types declared in a declaration shall not depend on the order of evaluation of any expressions it contains.

object initializer evaluated when

The corresponding coding guideline issues for initializers are discussed elsewhere.

Example

```
1 extern int glob;
2 extern int g(int);
3
4 void f(void)
5 {
6 int loc = g(glob = 3) + g(glob = 4);
7 }
```

(A summary of the sequence points is given in annex C.) 188

---

expression evaluation abstract machine

In the abstract machine, all expressions are evaluated as specified by the semantics. 189

Commentary

The abstract machine has no concern for the size of the program being executed, or its execution-time performance.

C++

The C++ Standard specifies no such requirement.

Common Implementations

When translating programs for use with a symbolic debugger, most implementations tend to closely follow the requirements of the abstract machine. This ensures that the values of objects, in memory, closely follow their evaluations in the source code. This simplifies things considerably for developers, who might be having enough trouble following the behavior of a program, and don't need the additional confusion of objects receiving values at different times to those specified in the source.

expression grouping

This seemingly innocuous requirement prohibits implementations from performing expression rewriting.

Coding Guidelines

Trying to second-guess how an implementation might process a given construct is generally a fruitless exercise. Assuming that an implementation behaves like the abstract machine is a useful starting point that does not often need to be improved on.

Example

In the following an implementation may spot that the subexpression x*y occurs in two places and the values of x and y are not modified between these two points. The extent to which this information can be used will depend on the number of registers available for holding temporary values, the relative cost of evaluating x*y, and the priority given to this optimization opportunity relative to other opportunities within the function f.

```
1  extern int x, y;
2
3  void f(void)
4  {
5  int a,
6      b;
7
8  a = x * y;
9  /*
10  * Code that does not change the values of x and y.
11  */
12 b = x * y;
13 }
```

---

190 An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

expression need not evaluate part of

Commentary

This statement forms part of the as-if rule.

as-if rule

In the following the value is used, but it is also known at translation time. To answer the question of whether there are any needed side effects requires varying degrees of source analysis sophistication:

```
1  extern int g(void);
2  extern int glob;
3
4  void f(void)
5  {
6  int loc;
7
8  loc = 0 * g();        /* Result always zero, but could have needed side effect. */
9  loc = 0 * (glob + 4); /* Result always zero, no side effects. */
10 }
```

The value assigned to loc is known in both cases. In the first case the call to the function g must occur if it causes needed (the program output depends on them) side effects. In the second case (glob + 4) only need be evaluated if the object glob is declared with the volatile storage qualifier. The cases differ only in the easy with which a translator can deduce that a needed side effect occurs.

Those cases where there may be no needed side effect apply to binary operators. Operator/operand pairs where evaluations need not occur and consequently there may be no needed side effects include:

```
0 * (expr)
0 & (expr) always 0
0xffff | (expr) always 0xffff (or appropriate sized constant)
```

```
(expr) - (expr) always 0
(expr) / (expr) always 1
```

The expression `expr / expr` might cause a side effect because of the undefined behavior that occurs when `expr` is 0. An implementation is at liberty to deliver the result `1` (if the original code is simplified), raise an exception (if that is what the host processor does for this case), or carry out some other action.

There are also cases where a constant operand does not affect the result of an expression, but the other operand still needs to be evaluated. For instance: adding zero, shifting by zero bits positions, the **%** operator with a right-hand side of one, or multiplying by one. In some cases the cast operator can have no effect. In the function:

```
1 static long glob;
2
3 int f(void)
4 {
5 return (int)glob;
6 }
```

the cast to **int** would be redundant if **int** and **long** had the same representation.

Common Implementations

The extent to which implementations perform the analysis necessary to deduce that an expression, or part of it, is not used varies enormously. The simpler optimizing translators tend to spend their time producing the best code from the source code on a statement-by-statement basis. These tend to treat all parts of an expression as being needed. More sophisticated optimizers analyze blocks of statements, trying locate common subexpressions, CSEs, and performing some level of flow analysis. This level of optimization is willing to believe that certain kinds of subexpressions may be redundant.

A translator that generates very high performance code is of no use if the final behavior is incorrect. The savings to be made from removing some redundant subexpressions are sometimes not worth the risk of getting it wrong. The translator that optimizes:

```
1 #define A_OFFSET(x) (1+(x))
2
3 extern int p, q;
4 static volatile int r;
5
6 void f(void)
7 {
8 p=A_OFFSET(q) - r / r;
9 }
```

into `p=q` would either be foolhardy or perform a great deal of analysis of the program and its environment.

A translator can optimize those cases where the operands always have known values. However, studies have shown[25] that a significant number of these redundant operations still occur during program execution (because the calculated values of operands happen to be zero or one).

partial redundancy elimination

In some cases code is only redundant under certain conditions. For instance, its evaluation may only be necessary along a subset of the control flow paths that pass through it. So called *partial redundancy elimination* algorithms involve code restructuring to remove this redundancy. Ensuring that this code motion and duplication does not result in excessive code bloat requires information on the execution-time characteristics of the program.[4]

There have been proposals[21] for detecting, in hardware, common subexpression evaluations during program execution and reusing previously computed (and stored in a special table) results.

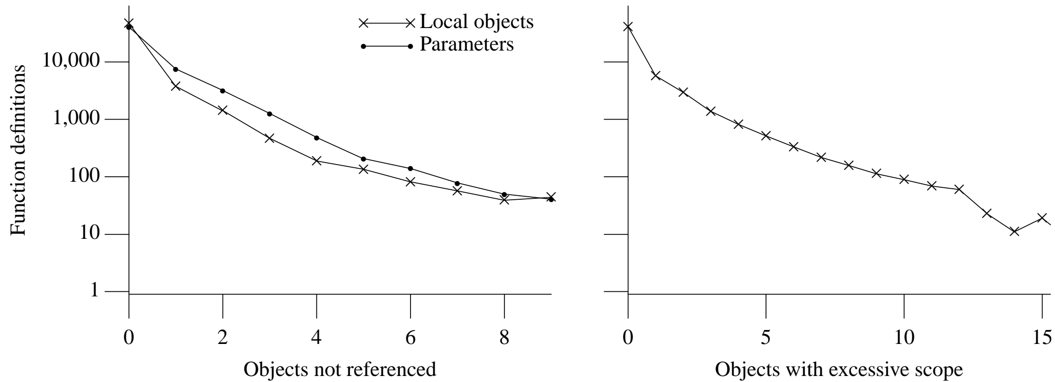

Figure 190.1: Number of parameters or locally defined objects that are not referenced within a function definition (left graph); number of objects declared in a scope that is wider than that needed for any of the references to them within a function definition (right graph). Based on the translated form of this book's benchmark programs.

Coding Guidelines

There are cases where, technically, part of an expression has no effect on the final result but where from the human point of view the operation needs to exist in the source code— including the following:

- Source code that is translated and forms part of the program image, but is never executed, is known as *dead code*. There is no commonly used equivalent term denoting objects that are defined but never referenced (see Figure 190.1). dead code
- Source code that is translated and forms part of the program image, whose execution does not affect the external appearance or output of a program, is known as *redundant code*. Declaring objects to have a scope wider than is necessary (e.g., an object declared in the outermost block that is only referenced within one nested block— Figure 190.1) is not generally considered to be giving them a redundant scope. redundant code
- Source code that appears within conditional inclusion directives (e.g., **`#if`**/**`#endif`**) is not classified as dead code. It may not be translated to form part of a program image in some cases, but in other cases it may be.

The issue of duplicate code is discussed elsewhere. Dead, or redundant, code can increase costs in a number of ways, including the following: duplicate code

- *Additional maintenance effort.* It is source code that developers do not need to read. They may even make changes to it in the mistaken belief that it affects program output.
- *Consume host processor resources.* For instance, in the case of unused storage, execution performance can be affected through its effect on cache behavior. cache

This kind of code can exist for a number of reasons, including:

- Defensive programming:
  - providing a default label even when it can be shown that all possible switch values are covered by labels
  - guideline recommendations that require a return at the end of a function, even though flow analysis shows it is never reached

- Conditional compilation based on translation-time constants in the conditional expressions of **if** statements, instead of using **#if/#endif**.
- Coding error.
- Using symbolic representations. For instance, a symbolic name, representing a numeric value, appearing as an operand, or the typedef name used in a cast operation that happens to denote the same C type as operand.

symbolic name
cast-expression syntax

- Design oversight:
  - an **if** statement that can never be true, or is always false, based on characteristics of the application
- Cut-and-paste of existing code rather than creating a new function.
- Specification changes:
  - conditions that could previously have been true become impossible to satisfy
- At the highest level, code may only be executed when certain hardware is available (e.g., joystick).
- A developer is aware of the situation but does not consider it worth investing effort to remove the code.

Locating all dead code within a program is technically a very difficult problem. The most commonly seen levels of analysis operate at the function and statement level. Automatic tools provide a partial solution to dealing with the complexities of detecting dead code. Such tools can operate at differing levels of sophistication:

- analyze each expression in isolation:
  - detects expressions that do not contain any side effects
- simple flow analysis within a function:
  - detects redundant statements
- symbolic flow analysis within a function, called intraprocedural analysis:
  - detects **if/while/for** never/always executed
- flow analysis, including information across function calls, called interprocedural analysis:
  - the additional information allows a more thorough job to be done
- complete program analysis:
  - Many linkers do not include function definitions that are never referenced in the program image. The extent to which objects that are not references are included in a program image varies.

linkers
program image

In practice, given current tool limitations, and theoretical problems associated with incomplete information (i.e., the set of values input to a program), it is unlikely that all dead code be detected.

Locating redundant code requires additional effort since it is necessary to show that particular side effects, which are performed, have no effect on the external behavior. An example of a case that is often flagged in an object defined and assigned a value within a function, but which is never referenced.

A study by Xie and Engler[30] attempted to find a correlation between redundant code and faults that existed in that code.

The extent to which the cost of including dead and redundant code in a program outweighs the cost of removing it is a management decision. It is not unusual to find that 30%, or more, of the functions in an application, which has evolved over many years, to be uncalled (i.e., they are dead code)[27] (see Table ??). It is also common to find objects that are defined and never referenced, and **#include**s that are unnecessary (see Figure ??). Dead code within functions that are called does occur, but not in significant quantities. In the case of duplicate code 5.9% of the lines of gcc have been found to be duplicate,[8] a study[2] of a 400 K C source line commercial application found an average duplication rate of 12.7%.

controlling expression
if statement

Example

```
1  #include <stdio.h>
2
3  extern int get_status(void);
4  static unsigned int glob_2;
5
6  void f(void)  /* Never called. */
7  {
8  glob_1++;
9  }
10
11 int main(void)
12 {
13 int local_1 = 11;
14
15 glob_1 = get_status();
16
17 if (0)
18    local_1--; /* Never executed. */
19 if (local_1 == 12)
20    glob_1--;  /* Never executed. */
21
22 if (glob_1 == 0)
23    glob_1 = 1;
24 if (glob_1 > 0) /* Always true, but needs logical deduction. */
25    printf("Hello world\n");
26 else
27    printf("Illogical world\n"); /* Never executed. */
28 }
```

---

191 When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on.

signal interrupt abstract machine processing

Commentary

C has no restrictions on when a signal can be raised. If it occurs via a call to the raise function there will have been a sequence point before the call. If the signal is raised as a result of some external mechanism, for instance a timer interrupt or the instruction being executed raising some signal, then the developer has no control over when it occurs.

Assuming an average of 3 machine code instructions for every source code statement gives a 66% chance (ignoring the issue of sequence points within the statement) of a C statement being partially executed when a signal occurs.

C++

1.9p9

When the processing of the abstract machine is interrupted by receipt of a signal, the value of any objects with type other than `volatile sig_atomic_t` are unspecified, and the value of any object not of `volatile sig_atomic_t` that is modified by the handler becomes undefined.

This additional wording closely follows that given in the description of the `signal` function in the library clause of the C Standard.

Common Implementations

processor pipeline

To improve performance modern processors hold a pipeline of instructions at various stages of completion. On some such processors, a signal is not processed until the pipeline has completed executing all of the instructions it contained when the signal was first received. For such processors it might not even be possible to associate a signal with a given, previous sequence point. Roberts[26] and Gehani[10] describe various implementation and application issues associated with signal handling.

Coding Guidelines

Because a signal handler does not have any knowledge of where a signal is likely to be raised, it cannot assume anything about the values of objects it references. Through knowledge of the source code and the host environment, a developer may be able to narrow down the locations where certain signals might be expected to be raised. Because it is not possible to say anything with certainty and the difficultly of performing automatic checking a guideline review recommendation is made.

Rev 191.1

When designing and implementing programs that operate in the presence of signals no assumptions shall be made about the values of objects modified by statements in the vicinity of the flow of control where a caught signal could be raised.

Example

In the following:

```
1  #include <signal.h>
2  #include <stdio.h>
3
4  extern volatile sig_atomic_t flag;
5  extern double value,
6                A, B;
7
8  void sig_handler(int sig_number)
9  {
10 if (flag == 1)
11    printf("Need not be after A / B\n");
12 }
13
14 void f(void)
15 {
16 signal(SIGFPE, sig_handler);
17
18 flag = 0;
19 value = A / B;
20 flag = 1;
21 }
```

The instructions to assign `1` to `f` might already be in the processor pipeline when the floating-point divide raises a signal. The function `sig_handler` cannot assume anything about the value of `flag` even though it has `sig_atomic_t` type.

The only solution to the pipeline problem is to insert sufficient statements between the floating-point operation and the assignment to flag to ensure that the assignment instructions are not in the pipeline when

the signal is raised. Even this is not a guaranteed solution because the translator might deduce (incorrectly in this case) that the machine code for these intervening statements could be moved to some other point in the flow of execution.

---

192 Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

modified objects received correct value

Commentary

It is not even possible to assume that objects will have one of two possible values (their previous or new value). In the two assignment:

```
1   A = 0x00ff;
2   /* Some code. */
3   A = 0xff00;
```

if a signal occurs during the evaluation of the second expression statement, the possible values the object `A` can take include `0x0000`, `0x00ff`, `0xffff`, or `0xff00`. These alternatives can occur if the underlying processor transfers multibyte values to storage one byte at a time.

The requirements on the Library typedef `sig_atomic_t` ensure that objects defined with this type will be accessed as an atomic operation. In the preceding example, for a processor with the byte at a time characteristics, the implementation would probably choose to use a character type in the definition of `sig_atomic_t`.

C++

The C++ Standard does not make this observation.

Common Implementations

It is rare for processors to support the handling of an interrupt after the partial execution of an instruction. This behavior is sometimes seen for instructions that move large amounts of data; for instance, string-handling instructions, where the processor may loop until some condition is met, can potentially take a very (relatively) long time to execute. Processors generally want to respond to an interrupt within a specified time limit.

Coding Guidelines

The myriad of possible problems that can occur as a result of an object not receiving its correct value, because of a signal being raised, are to0 diverse (and specialized) to be covered in a few guidelines.

---

193 The least requirements on a conforming implementation are:

implementation least requirements

Commentary

The *least requirements* listed provide the basis for ensuring that the output appears in the order required by the author of the code. However, the standard says nothing about how quickly it will appear after the statement performing the operation has finished executing. The only way of telling whether an implementation has ordered the operations that occurred during the execution of a program according to these requirements is to examine the output produced.

The workings of the abstract machine between the user issuing the command to execute a program image and the termination of that program is unknown to the outside world. For instance, a translator may recognize that a particular sequence of source code will output a given number of digits of the value of π. It may then translate the source into a program that prints Ascii characters from an internal table, to the desired precision, instead of executing the algorithm to calculate the digits contained in the program's source.

184 abstract machine C
205 semantics stringent correspondence

Some naive benchmark programs contain loops that have no effect on the output of the program (usually for some timing purpose). Users of such benchmarks continue to be confused by the timing results obtained from using an optimizing translator that deduces that it does not need to generate any executable code for these loops.

Other Languages

Most programming languages define the order in which statements are executed (in some languages that support a model of parallel execution, the order of some statement executions is indeterminate). Many of them have a more abstract view of program execution than C (which in many ways is more closely tied to host processor execution) and say little more than assignments update the value of a variable.

Coding Guidelines

Most order of execution coding problems occur because developers assumed an ordering that is not guaranteed by the standard; for instance, expressions containing multiple sequence points. The details of when this occurs is a developer education issue, not a coding guidelines issue (other coding guideline documents sometimes simply recommend against writing such expressions).

— At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred. 194

Commentary

This requirement ensures that if there is no more than one volatile access between consecutive sequence points, the order in which the accesses occur is the same as the order in which the sequence points are reached. Also this requirement deals with accesses, not values. Almost nothing can be said about the values of volatile objects.

sequence points 187

Except for the sequence point that occurs at the semicolon punctuator, usually little can be said about the order in which sequence points occur.

Common Implementations

Implementations usually treat expressions containing volatile objects with great caution. In some cases many optimizations are not performed for the full expression referencing such objects.

Coding Guidelines

sequence points 187

Experience suggests that developers sometimes have their own beliefs on access ordering relationships that goes beyond the minimum requirements of the C Standard. These beliefs only become a potential problem when the same object is modified more than once between two sequence points.

Example

In the following there is no guaranteed order of object access to `b` or `c` in the first statement. A comma operator provides the necessary sequence point in the second expression.

```
1  int a;
2  volatile int b, c;
3
4  void f(void)
5  {
6  int t;
7
8  a = b + c;
9  a = ((t = b), t + c);
10 }
```

while in:

```
1  int x,
2      y;
3  volatile int b, c;
4
5  void f(void)
6  {
7  x = b + c;
8  y = b + c;
9  }
```

the common subexpression b+c cannot be optimized in the assignment to y. Normally a translator would have the option of keeping the value of this calculation in a register, or loading it from x. However, because the operands have the volatile storage class, they must be accessed to obtain their values.

---

195 10) In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **main** will have ended in the former case, even where they would not have in the latter.

footnote 10

C90

This footnote did not appear in the C90 Standard and was added by the response to DR #085.

Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *pi;
5
6  void DR_085(void)
7  {
8  /*
9   * The lifetime of the object accessed by *pi
10  * has terminated.  Undefined behavior.
11  */
12 printf("Value is %d\n", *pi);
13 }
14
15 int main(void)
16 {
17 int i;
18
19 atexit(DR_085);
20 i = 42;
21 pi = &i;
22 return 0; /* Causes wording in DR #85 apply. */
23 }
```

---

196 11) The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes.

footnote 11

Commentary

The footnote is making the observation that modifying the values of these user-accessible status flags and control modes is to be considered a change of state of the execution environment. However, the response to DR #287 states that status flags are not objects (and modifying them between sequence points does not generate undefined behavior).

200 status flag floating-point
200 control mode floating-point
footnote DR287
object modified once between sequence points

> *When set, a status flag may contain additional system-dependent information, possibly inaccessible to some users. The operations of this standard may, as a side-effect, set some of the following flags: inexact result, underflow, overflow, divide by zero and invalid operation.*

IEC 60559

Inexact occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. This happens when the result of an operation overflows, or underflows, or is not exactly representable in the precision used. For instance, when FLT_RADIX == 2, the value of 1.0/10.0 is inexact, but is exact when FLT_RADIX == 10 (in the past, computers sold into the business market, where division by powers of 10 is common, were often designed to use a base 10-radix for this reason). One use for this flag is to support integer arithmetic in a floating-point unit.

subnormal numbers

Underflow: the result of an arithmetic operation, needs to be represented as a subnormal number or between zero and the smallest subnormal number. Underflow can also cause an exception to be raised.[7]

Overflow: the result of an arithmetic operation, is greater than the largest finite floating-point number. For, divide-by-zero, as the name suggests, an attempt has been made to divide by zero (except for the special case where the numerator is also zero). It also occurs if an operation on a finite value yields an exact infinite result. Invalid operation— this might include subtracting infinity from infinity, or dividing infinity by infinity, or zero by zero.

IEC 60559 *The following modes shall be implemented:*

1. *rounding, to control the direction of rounding errors,*
2. *and in certain implementations, rounding precision, to shorten the precision of the result. [This is not required for chips that do: single = single OP single. It is required for chips that do everything in more precision than single, such as Intel x87].*
3. *The implementation may, at his option, implement the following modes: traps disabled/enabled, to handle exceptions.*

Warning: The IEC 559 Standard copied the preceding words from the IEEE-754 and IEEE-854 standards incorrectly, changing the meaning. Make sure you have a copy of the IEC 60559 document, not IEC 559.

C90

The dependence on this floating-point format is new in C99. But, it is still not required.

C++

The C++ Standard does not make these observations about IEC 60559.

Common Implementations

Most processors implement the status flags listed here. The function `_control87` was supported by many MS-DOS based (with continuing compatibility support under Windows) implementations hosted on Intel x86 processors. This function enabled various Intel x87 maths coprocessor (now integrated with the CPU) control flags to be read and set.

Most implementations assume that the developer will not change the control modes (such usage is rare and handling it can introduce a large performance penalty) and that the processor will be running in round-to-nearest and default precision. If the user changes either mode and then calls a library function, defined in the headers **`<math.h>`** or **`<stdio.h>`**, incorrect results could be returned. The better vendors may supply two versions of the library: one that assumes the floating-point environment is the default one, and a slower one that allows for the developer altering the environment.

Coding Guidelines

The setting of the status flags and control modes defined in IEC 60559 represents information about the past and future execution of a program. Floating-point operations are a technically complex subject and the extent to which developers or source code alter or test this information will depend on many factors. Apart from the general exhortation to developers to be careful and to make sure they know what they are doing, there is little of practical use that can be recommended.

Floating-point operations implicitly set the status flags; 197

Commentary

type qualifier syntax

In a sense floating-point operations may have semantic similarities with accesses to objects declared with a **`volatile`** qualified type.

status flag 200 floating-point

Integer operations may also set status flags. The difference between the two sets of flags is that the floating-point flags are sticky (i.e., once set, they stay set until they are explicitly reset). This property of IEC 60559 status flags is now a recognized state of the C abstract machine.

C++

The C++ Standard does not say anything about status flags in the context of side effects. However, if a C++ implementation supports IEC 60559 (i.e., `is_iec559` is true, 18.2.1.2p52) then floating-point operations will implicitly set the status flags (as required by that standard).

Coding Guidelines

Checking status flags after every floating-point operation usually incurs a significant performance penalty. The status flags were designed to be sticky to enable checks to be made after a series of floating-point operations. However, the more operations performed before a check is made, the less detailed information is available about where the potential problem occurred. The extent to which it is cost effective to use the information provided by the status flags is outside the scope of these coding guidelines.

Example

```
1   #include <fenv.h>
2
3   extern float f_glob;
4
5   void f(void)
6   {
7   fexcept_t status_info;
8   float f_loc = f_glob * f_glob;
9
10  fegetexceptflag(&status_info, FE_ALL_EXCEPT);
11  /*
12   * Now check the information returned in status_info.
13   */
14  }
```

---

198 modes affect result values of floating-point operations.

Commentary

The modes affect such things as the rounding of arithmetic operations. To some extent developers have some control over them through the use of the `FENV_ACCESS` pragma.

FLT_ROUNDS

C++

The C++ Standard does not say anything about floating-point modes in the context of side effects.

---

199 Implementations that support such floating-point state are required to regard changes to it as side effects— see annex F for details.

side effect floating-point state

Commentary

In practice the status flags associated with floating-point state differ from those associated with other operations (e.g., the flags set as the result of integer operations, such as result is zero) in that they can affect the results of floating-point operations. A processor's ability to select how integer overflow is handled might, in theory, be considered a mode. However, in practice few processors provide such functionality.[6]

The `FENV_ACCESS` pragma provides a mechanism for developers to inform implementations that they are accessing the floating-point environment.

C++

The C++ Standard does not specify any such requirement.

Common Implementations

In a multiprocess environment any floating-point state flags have to be saved and restored on each context switch. This saving and restoring is usually handled by the operating system.

endian

Processors rarely have any status flags that affect operations on integer types (although some processors have the ability to dynamically change their endianness, which is usually only done at boot time). For this reason, the C Standard limits its discussion to floating-point state.

Coding Guidelines

An algorithm may depend on a particular floating-point state for its correct operation. Programs using such algorithms need to ensure that this state is set up as part of the initialization done prior to using the algorithm. They may also need to restore any previous state, if it was different from the one used for the algorithm.

The floating-point environment library **<fenv.h>** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases. 200

Commentary

status flag floating-point exception floating-point control mode floating-point

The following except from "7.6 Floating-point environment <fenv.h>" of the library section defines the terms *floating-point environment*, *floating-point status flag*, *floating-point exception*, and *floating-point control mode*.

7.6 Floating-point environment **<fenv.h>**

> *A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information. A floating-point control mode is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.*[DR287a)]

The WG14 document N753 says:

WG14/N753

> *It is assumed that the integer and floating-point environments each consist of a control word and a status word. The status word contains bits (sticky flags) to indicate the state of past operations. The status word need not be a hardware register, it may be in memory and maintained by system software.*
>
> *The floating-point environment consists of:*

```
status
   sticky flags
      invalid
      div-by-zero
      overflow
      underflow
      inexact
   optional
      current operation being performed
      exception(s) of current operation
      exception(s) still pending
      operand values
      destination's precision
      rounded result
control
   rounding
   precision (optional)
   trap enable/disable (optional)
      invalid
      div-by-zero
      overflow
```

```
            underflow
            inexact
```

While the Rationale says:

Rationale

The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that can affect a program's results. Each such option's deviation from this specification should be well documented.

Dynamic vs. static modes

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings, which imposes intellectual as well as time and space overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time, which inhibits optimization.

C99 addresses these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of static modes with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEC 60559 standard requires dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, standard C has no facility, other than pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling `FENV_ACCESS` pragma would need such special treatment.

Translation

An implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program.

The language and library provide facilities to cause floating-point operations to be done at execution time when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.

C90

Support for the header **<fenv.h>** is new in C99.

C++

Support for the header **<fenv.h>** is new in C99, and there is no equivalent library header specified in the C++ Standard.

Other Languages

There is an ISO Technical Report[18] that deals with floating-point exception handling in Fortran.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced. 201

Commentary

A strictly conforming program has a unique sequence of output data for a given sequence of input data. The output from a conforming program is not guaranteed to be unique (an unspecified behavior may cause different implementations to generate different).

interactive device intent 203

The standard says very little about when output appears. Program termination causes all open files to be closed. Although closing a file causes any buffered data to be written to it, nothing is said about when this flushing needs to be completed. The only thing that can be said about program termination is that there will be no more output from that program (during that execution).

The library file-handling functions deal with the issue from a single executing program's perspective. How another program, running on a host capable of supporting more than one program executing at the same time, might view the contents of a file being written to by more than one conforming C program at the same time is not dealt with by the C Standard.

C++

1.9p11

— At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

The C++ Standard is technically more accurate in recognizing that the output of a conforming program may vary, if it contains unspecified behavior.

Common Implementations

Many file systems' cache writes, to a file, into blocks. Once a block is full the data it contains is written to the file. Any cached data is also written to a file when it is closed. Many modern environments have multiple caches. Whether a buffer flushed from an individual program's I/O cache ever gets physically written, as a pattern of bits, on to an actual storage device might never be known. A temporary file may be opened, written to, closed and then removed; all associated data being held in one or more caches.

— The input and output dynamics of interactive devices shall take place as specified in 7.19.3. 202

Commentary

Clause 7.19.3 does not say what shall happen, only what is intended to happen.

Rationale

The class of interactive devices is intended to include at least asynchronous terminals, or paired display screens and keyboards. An implementation may extend the definition to include other input and output devices, or even network inter-program connections, provided they obey the Standard's characterization of interactivity.

Common Implementations

Having the host processor, in a multiuser environment, deal with every key press from all users can consume a large amount of resources. Some mainframe environments require an escape key to be pressed to indicate to the host that input is being sent. It can be more efficient to have the local input device store the characters until a response is required from the host (e.g., at the end of a newly typed line of characters).

Most, nonmainframe, implementations can support a line-oriented conversation.

interactive device intent

The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input. 203

Commentary

Such buffering is very desirable when attempting to have a realtime conversation with a person or computer at the sending/receiving end of the input/output. Not all hosts provide the ability to perform anything other than buffered I/O, hence this specification is an intent rather than a requirement.

Line-buffered output is intended to appear every time a new-line character is written to a stream. The individual characters need not appear as they are written by the program, for instance if a string is output, followed several source statements later by the output of a number. The output might appear a line at a time.

The closing of a stream is another event that is likely to cause output to appear fairly promptly.

Common Implementations

Having the host dispatch output characters one at a time for writing to a file is very inefficient. In a multiuser environment there are performance gains to be had in passing complete lines to the output device.

There can be a significant performance penalty associated with continually opening and closing streams.

Coding Guidelines

I/O handling is an important issue for applications and vendors often put a lot of effort into this functionality. It is one area where extensions are often used and where following guidelines can help minimize dependencies on a particular implementation. The extent to which an application depends on output appearing in a timely manner is a design issue that is outside the scope of these coding guidelines.

?? extensions cost/benefit

Example

Once consequence of buffered output handling is that the same program may behave differently depending on how it is executed:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3
4  int main(void)
5  {
6  printf("Started\n");
7  if (fork() == 0)
8     printf("Child\n");
9  }
```

If, when executed, `stdout` is an interactive device, the preceding program will produce:

```
Started
Child
```

If, when executed, `stdout` is not an interactive device, there is a high probability that the program will produce:

```
Started
Started
Child
```

The reason is that the `fork` system call (it's in POSIX, but not C) duplicates all of the parents' data structures, including its internal I/O buffers (defined by the C header **`<stdio.h>`**) for the child process. If buffered I/O is taking place (likely when writing to a noninteractive device), the string generated by the first `printf` will be held in one of these buffers before being copied to the O/S output buffers. When the **`<stdio.h>`** buffers are copied, their contents, including any pending output, is also copied. When these buffers are finally flushed, two copies of the string **Started** appear on the output stream (see section 8.2 of Zlotnick[31] for a detailed discussion and possible solutions).

---

204 What constitutes an interactive device is implementation-defined.

interactive device what constitutes

Commentary

Not only the type of device, but how it is used may affect whether it is treated as an interactive device. Serial devices (i.e., those that send and receive data as a stream of bytes) are often implemented as interactive devices. However, if access to a device is occurring as some background task because the program is executing in batch mode, the host may decide not to treat it as an interactive device.

Block devices (i.e., those that handle data in blocks, usually of 512 bytes or more) are not usually interactive devices. If more than one program is accessing data written to a block device, it may be necessary to ensure that all writes to that device occur when they are executed. But, support for such multiprogram behavior is outside the scope of the C Standard. One block device that is sometimes treated as an interactive device is a tape. Here the output is assembled into blocks for efficient writing and storage, but is written serially.

Other Languages

Most languages do not specify anything about low-level device details.

Common Implementations

The devices connected to `stdin`, `stdout`, and `stderr` on program startup are often interactive devices, if such are available.

Hard disks are block devices and are rarely interactive devices.

Coding Guidelines

Programs that require their output to appear in a timely fashion need to ensure that the devices they are using support such behavior. The C Standard provides intent to implementors, not guarantees to developers. Coding guidelines can say nothing, other than reminding developers that the intended behavior may vary between implementations.

---

semantics stringent correspondence

More stringent correspondences between abstract and actual semantics may be defined by each implementation. 205

Commentary

This is really a warning that implementations may have a low quality of implementation in this area. They might not perform any code optimizations, updating all objects at the point their values are modified; I/O could be performed on a character-by-character basis (input and output is defined in terms of `fgetc` and `fputc`).

Common Implementations

The commercial pressures on vendors is rarely to conform more strictly to standards (although a strong interest in conformance to standards is invariably claimed). Rather, it is to provide features that improve a program's ability to interact with the host environment. Many implementations provide several additional functions to ensure that characters are written to a device in a timely manner.

Coding Guidelines

More stringent correspondences would not change the behavior of a strictly conforming program. It may change the behavior of a conforming program to one of the set of possible behaviors that any implementation could produce.

A general principle of coding guidelines is to recommend usage that minimizes a program's exposure to the latitude available to an implementation in executing it. More stringent correspondences might be viewed as enabling programs to produce more reliable results. However, using a more stringent implementation only saves costs if the program has been written to that specification. Porting existing code to such an environment is simply that, another port.

---

abstract machine example

EXAMPLE 1 An implementation might define a one-to-one correspondence between abstract and actual 206

semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **volatile** would then be redundant.
Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the **signal** function would require explicit specification of **volatile** storage, as well as other implementation-defined restrictions.

Commentary

Use of **volatile** is usually treated as a very strong indicator that the designated object may change in unexpected ways. For it to be redundant the implementation would also have to specify the order of evaluations of an expression containing objects defined using this qualifier. If an optimizer were clever enough, it could even ignore these sequence points (which is leading edge optimizer technology). Commercially available optimizers tend to limit themselves to what they can analyze in a single translation unit.

Requiring that the program declare all objects accessed by an interrupt service routine with the **volatile** storage-class specifier is overly restrictive. Such routines can be invoked other than via a call to the `abort` or `raise` functions. The values of objects, as of the previous sequence point, can be relied on.

191 signal interrupt abstract machine processing

Common Implementations

On average every fifth statement is a function call. This is very frustrating for writers of optimizers (long sequences of C code without any function calls provide more opportunities to generate high-quality machine code). The introduction of support for the **inline** function specifier, in C99, offers one way around this problem for time-critical code.

function specifier syntax

Cross function call optimization is made easier if all of the source of the called function is available (i.e., it is in the same translation unit as the caller), something that does not often occur in practice. Generation of the highest quality code requires that code generation be delayed until link-time, when all of the information about a program is available.[22, 28]

The ability of an expression to raise a signal ruins the potential for optimization. For instance, in:

```
1  glob_flag = 1;
2
3  x=expr1_could_raise_signal;
4
5  glob_flag = 2;
6
7  x=expr2_could_raise_signal;
8
9  glob_flag = 3;
```

a function registered to handle the signal that may be raised in either expression might want to examine `glob_flag` to get some idea of which expression raised the signal.

Calls to the `raise` function can be detected and code generated to ensure that objects have the values required by the abstract machine.

---

207 EXAMPLE 2 In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the "integer promotions" require that the abstract machine promote the value of each variable to **int** size and then add the two **int**s and truncate the sum. Provided the addition of two **char**s can be done without overflow,

or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

Commentary

Silently wrapping to produce the correct result is the most commonly seen processor behavior. Because c1 and c2 have the same types in this example, there are no potential complications introduced by them having different signed types.

Common Implementations

Modern processor instructions invariably operate on the contents of registers. These having been zero filled or sign extended, if necessary, when the value was loaded from storage, irrespective of the size of the value loaded. The cast, for promoted values, is thus implicit in the load instruction.

Older processors (including the Intel x86 processor family) have the ability to load values into particular bytes of a register. It is possible for some instructions to operate on these subparts of a register. The original rationale of such a design is that instructions operating on smaller ranges of values could be made to execute faster than those operating on larger ranges. Technology has now advanced (the Intel x86 in particular) to the stage where there are no longer any performance advantages to such a design. The issue is now one of economics— wider processor buses incur higher costs.

Some implementations (e.g., Tasking[1]) support an option that causes operations on operands having character type to be performed using processor instructions that manipulate 8-bit values (i.e., the integer promotions are not performed).

Coding Guidelines

In expressions such as:

```
1  unsigned char c1, c2, c3;
2
3  void f(void)
4  {
5  if (c1 == c2 + c3)
6     c1=3;
7  }
```

there is a commonly encountered incorrect assumption made by developers that in this case the equality and arithmetic operations are all performed at character type, the result of the addition always being kept within range of the type **unsigned char**. Recommending that all developers' be trained on the default integer promotions and the usual arithmetic conversions is outside the scope of these coding guidelines. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults (perhaps resulting from poor training).

guidelines not faults

EXAMPLE 3 Similarly, in the fragment 208

```
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
```

the multiplication may be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if **d** were replaced by the constant **2.0**, which has type **double**).

Commentary

Such ascertaining is very hard, in practice, to do for floating-point arithmetic. It is also very tempting (instructions that operate on single-precision values are sometimes much faster than those that operate on double-precision values; not on Intel x87 where operations on objects having type **long double** are fastest).

FLT_EVAL_METHOD

The precision in which floating-point operations take place is not just a matter of performance. A numerical

algorithm may depend on a particular level of accuracy. More-than-expected accuracy can sometimes be just as damaging to the final result as less-than-expected accuracy.

---

209 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **d1** and **d2** are required to have been converted to **float**.

Commentary

register spilling

Spilling is a technical code-generation term for what happens when the generator runs out of free working registers. It has to temporarily copy the contents of a register to a holding area in storage, freeing that register to be used to hold another value. Compiler writers hate register spills (not being able to generate code that only requires the available registers is seen as a weakness in the quality of their product).

There may not need to be an explicit store-and-load operation to round a value to the precision of the storage type. A processor instruction may be available to perform such as a conversion, which has the same effect. Even if they are converted back to the type **double** prior to the assignment, the sequence of cast operations `(double)(float)` is not a no-operation. The conversion to type **float** may result in a loss of precision in the value converted.

C90

This example is new in C99.

Common Implementations

The floating-point unit for the Intel x86 processor family[14] uses 80 bits internally to represent floating-point values. All floating-point operations are usually performed using this representation. A precision-control word allows this behavior to be changed during program execution for the floating add, subtract, multiple, divide, and square root instructions. However, this control word only affects the precision of the significand; the range of possible exponent values is not reduced (15 bits are always used). Because the range of exponent values is not reduced, it is possible to represent smaller values than would be possible in the single- or double-precision type. The only way to ensure that all additional bits used in the 80-bit representation are removed is to store the value, to storage, and reload it. The specification of the Java virtual machine requires that floating-point operations be carried out to the accuracy of the type, only. Performing the store/load operations needed to meet this requirement causes a factor of 10 performance penalty, although techniques to reduce this to a factor of 2 to 4 have been proposed.[19]

Some implementations (e.g., Microsoft Visual C++, `gcc`) provide an option that results in all reads of objects (having a real type) to be loaded from the object (all assignments also cause the object value to be updated). By not optimizing the generated machine code to reuse any value that happens to be in one of the floating-point registers, the consistency of expression evaluations is improved. If this option is not enabled, there is the possibility that a reference to an object will have greater precision in some cases because the value used was one already present in an 80-bit register, with potentially greater accuracy because it had been returned as the result of an arithmetic operation, and not loaded from storage.

A register spill has the potential for altering the value of a floating-point number. For instance, if registers hold values to greater precision than the representation of a subexpression's C type, the instructions used to perform the spill may chose to save the exact register contents to temporary storage or may round the result to the C type. On some processors the overhead of storing the exact register contents is much higher than using

the ordinary floating store instructions; some implementations have been known to chose to use the faster instructions, effectively rounding an intermediate result in those cases where a register spill had to be made.

Example

In the following:

```
1  #include <stdio.h>
2  
3  double f3, f1, f2;
4  
5  void f(void)
6  {
7  f3=f1+f2; /* The assignment. */
8  
9  if (f3 == (f1+f2)) /* A simple comparison. */
10    printf("As expected\n");
11 else
12    printf("Did not expect to get here (but it can happen)\n");
13 
14 if (f3 == (double)(f1+f2)) /* Another comparison. */
15    printf("As expected\n");
16 else
17    printf("This never appears\n");
18 }
```

the assignment will scrape off any extra bits that may be held by the processor performing the addition. In the simple comparison it would be tempting (and permitted by the standard) for an implementation, in terms of quality of generated machine code, to perform the addition of `f1` and `f2`, hold the result in a register, and then compare against `f3`. However, if the result of the addition is not adjusted to contain the same number of representation bits as were stored into `f3`, there is the possibility that any extra bits returned by the addition operation will cause the *another comparison* to fail.

EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.8). 210

```
double x, y, z;
/* ... */
x = (x * y) * z;  // not equivalent to x *= y * z;
z = (x - y) + y ; // not equivalent to z = x;
z = x + x * y;    // not equivalent to z = x * (1.0 + y);
y = x / 5.0;      // not equivalent to y = x * 0.2;
```

Commentary

The normal arithmetic identities that hold for unsigned integer types (and sometimes for signed integer types) rarely hold for floating-point types.

C90

This example is new in C99.

Other Languages

Fortran is known for its sophistication in handling floating-point calculations.

Common Implementations

The average application does not use floating-point types. Applications involving intensive floating-point calculations tend to be niche markets, and there are niche vendors who specialize in translating programs that perform lots of floating-point operations.

Coding Guidelines

The average translator is not very sophisticated with respect to optimizing expressions containing floating-point values. If possible, such optimizations need to be disabled unless purchasing a product from a vendor known to specialize in numerical computation.

---

211 EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment

EXAMPLE expression grouping

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum **(a + 32760)** is next added to **b**, and that result is then added to **5** which results in the value assigned to **a**. On a machine in which overflows produce an explicit trap and in which the range of values representable by an **int** is [-32768, +32767], the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for **a** and **b** were, respectively, -32754 and -15, the sum **a + b** would produce a trap while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for **a** and **b** might have been, respectively, 4 and -8 or -17 and 12. However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

Commentary

expression grouping
operator precedence

This example illustrates how a left-to-right parse of the input token stream associates the operands. The binding of operands to operators having the same precedence is specified by the language syntax. The brackets do not imply any ordering on the accessing of the objects `a` and `b`. An implementation may choose to load `b` into a register before evaluating `(a+32760)`. Such behavior will not be noticeable unless both operands have a volatile-qualified type and their values change during execution.

volatile qualified attempt modify

C90

The C90 Standard used the term *exception* rather than *trap*.

Common Implementations

Host processors that trap on overflow of signed integer operations are rare and many translators would freely rewrite the expression. Issues of instruction performance and implementation simplicity have won the day, in most cases.

---

212 EXAMPLE 7 The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of **p** can occur at any time between the previous sequence point and the next sequence point (the **;**), and the call to **getchar** can occur at any point prior to the need of its returned value.

Commentary

A C translator also has complete freedom, subject to sequence point requirements, to select the order in which the various parts of an expression are evaluated. Perhaps even reusing a result from a previously calculated subexpression.

Common Implementations

With optimizations switched off, many translators will generate code to evaluate the expression in either left-to-right, or right-to-left order.

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), the **signal** function (7.14), files (7.19.3). 213

References

1. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
2. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–378. IEEE Computer Society Press, 1998.
3. S. Blazey, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: International Symposium on Formal Methods*, pages 460–475. Springer-Verlag, 2006.
4. R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
5. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software–Practice and Experience*, 26(6):701–724, 1997.
6. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.
7. A. Cuyt, P. Kuterna, B. Verdonk, and D. Verschaeren. Underflow revisited. *Calcolo*, 39(3):169–179, 2002.
8. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicate code. In H. Yang and L. White, editors, *Proceedings International Conference on Software Maintenance ICSM'99*, pages 109–119. IEEE Computer Society Press, Sept. 1999.
9. K. Finney and N. Fenton. Evaluating the effectiveness of Z: the claims made about CICS and where do we go from here. Technical Report DeVa TR No. 05, City University, London, 1996.
10. N. H. Gehani. Exceptional C or C with exceptions. *Software–Practice and Experience*, 22(10):827–848, 1992.
11. S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165. IEEE Press, May 2003.
12. Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Boerger, H. K. Buning, G. Jager, S. Martini, and M. M. Richter, editors, *Computer Science Logic: Selected Papers from CSL'92*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.
13. P. Gutmann. Verification techniques. In P. Gutmann, editor, *Design and Verification of a Cryptographic Security Architecture*, chapter 4. Springer-Verlag, 2003.
14. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
15. ISO. *ISO/IEC 13211-1:1995 Information technology —Programming languages, their environments and system software interfaces —Programming language Prolog —Part 1: General Core*. ISO, 1995.
16. ISO. *ISO/IEC 10514-1:1996 Information technology —Programming languages —Part 1. Modula-2, Base language*. ISO, 1996.
17. ISO. *ISO/IEC 13817-1:1996 Information technology —Programming languages, their environments and system software interfaces —Vienna Development Method —Specification language Part 1. Base language*. ISO, 1996.
18. ISO. *ISO TR 15580:1998 Information technology —Programming languages —Fortran —Floating-point exception handling*. ISO, 1998.
19. Java Grande Forum Numerics Working Group. Improving java for numerical computation. math.nist.gov/javanumerics/reports/jgfnwg-01.htm, Oct. 1998.
20. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.
21. C. Molina, A. González, and J. Tubella. Dynamic removal of redundant computations. In *Proceedings of the* 13th *International Conference on Supercomputing*, pages 474–481. ACM Press, 1999.
22. R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, The Department of Computer Science, University of Arizona, Wednesday, Dec. 9 1998.
23. M. Norrish. *C formalized in HOL*. PhD thesis, Cambridge University, 1998.
24. N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Greece, Feb. 1998.
25. S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, Inc, Sept. 1992.
26. E. S. Roberts. Implementing exceptions in C. Technical Report 40, Digital Systems Research Center, Mar. 1989.
27. P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari. Reverse engineering 4.7 million lines of code. *Software–Practice and Experience*, 30(2):129–150, Feb. 2000.
28. D. W. Wall. Global register allocation at link time. Technical Report 86/3, Western Research Lab, Oct. 1986.
29. J. Welsh and A. Hay. *A Model Implementation of Standard Pascal*. Prentice-Hall, Inc, 1986.
30. Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, Nov. 2002.
31. F. Zlotnick. *The POSIX.1 Standard: A Programmer's Guide*. The Benjamin/Cummings Publishing Company, 1991.