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.

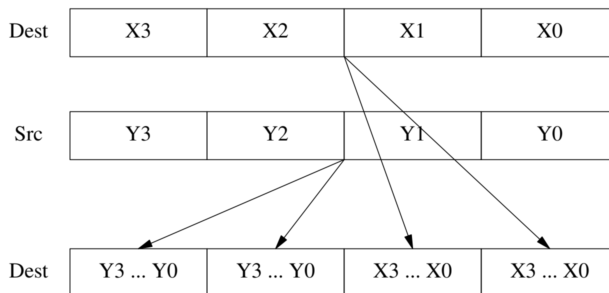

Figure 940.1: The SHUFPS (shuffle packed single-precision floating-point values) instruction, supported by the Intel Pentium processor,[19] places any two of the four packed floating-point values from the destination operand into the two low-order doublewords of the destination operand, and places any two of the four packed floating-point values from the source operand into the two high-order doublewords of the destination operand. By using the same register for the source and destination operands, the SHUFPS instruction can shuffle four single-precision floating-point values into any order.

6.5 Expressions

expressions

An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. 940

Commentary

This defines the term *expression*. The operators available for use in C expressions reflect the kinds of operations commonly supported by processor instruction sets. Processors often contain instructions for performing operations that have no equivalent in C. For instance, the Intel SSE extensions[19] to the Intel x86 instruction set support the SHUFPS instruction (see Figure 940.1). There is no equivalent operator in C.

NaN

Expressions in C differ from expressions encountered in most algebra books in that in C the representable range is finite and contains an additional representable quantity, NaN (a few advanced books[26] cover the issues of infinity and NaN). Some of the expression transformations that deliver the same result in mathematics can deliver different results in C.

sequence points

Although the evaluation of an expression is often thought about by developers as if it was a single sequence of operations, the only sequencing requirements are those imposed by sequence points.

expression optimal evaluation

Most expressions need to be evaluated during program execution. The generation of machine code to evaluate expressions is a very simple problem. The complexities seen in industrial-strength translators are caused by the desire to generate machine code that minimizes some attribute (usually either execution time, size of generated code, or electrical power consumed). It is known that the selection of an optimal sequence of instructions, for an expression, is an NP-complete problem[2] (even for a processor having a single register[8]). Algorithms (whose running time is linear in the size of the input) that minimize the number of executed instructions, or accesses to storage, are known for those expressions that do not contain common subexpressions (for processors without indirection,[32] stack-based processors of finite depth,[9] and general register-based processors[1]).

common subexpression

usual arithmetic conversions

Unless stated otherwise, the result type of an operator is either that of the promoted operand or the common type derived from the usual arithmetic conversions.

C++

The C++ Standard (5p1) does not explicitly specify the possibility that an expression can designate an object or a function.

Other Languages

This definition could be said to apply to most programming languages, which often contain a common core of similar operators operating on the same operand types. Languages vary in their support for additional operators and operand types that the core operators may operate on. A few languages (e.g., Algol 68, gcc

supports compound statements in an expression) support the use of statements within an expression, or to be more exact, statements can return a value. Functional languages are side effect free. Evaluation of an expression returns a value and has no other effect.

Common Implementations

Evaluation of expressions is what causes translators to generate a large proportion of the machine code output to a program image. Optimization technology has improved over the years. The first optimizers operated at the level of a single expression, or statement. As researchers discovered new algorithms, and faster processors with more main memory became available, the emphasis moved to basic blocks. Here, the generation of machine code for expressions takes the context in which they occur into account. Optimizing register allocation, detecting and making use of common subexpressions, and on modern processors performing instruction scheduling to try to avoid pipeline stalls.[11] The continuing demand for more optimization has led to further research and commercialization of optimizers working at the so-called *super basic block* level, the level of a complete function definition, and most recently at the complete program level.

basic block
register storage-class
common subexpression
basic block
function definition syntax
translation unit syntax

Performance is often an issue in programs that operate on floating-point data. A number of transformations are sometimes used to produce expressions that deliver their results more quickly. These transformations can result in changes of behavior, including a greater range on the error bounds. Possible transformations are discussed under the respective operators. The Diab Data compiler[13] supports the `-Xieee754-pedantic` option to control whether these (i.e., convert divide to multiple) optimizations are attempted.

expression processor evaluation

Expression evaluation often requires that intermediate results be temporarily stored. The almost universal technique used is for processors to contain temporary storage locations— registers. (Stack architectures[23] may be simpler and cheaper to implement, but do not usually offer the cost/performance advantages of a register-based architecture, although they are occasionally seen in modern processors.[17,30]) The use of registers is discussed in more detail elsewhere.

register storage-class

The fact that operators are defined to operate on one or two values, returning a single value as a result, is not a hindrance to extending them to operate in parallel on vectors of operands as some extensions have done. In the quest for performance a number of processors are starting to offer vector operations. Operations to manipulate elements of a vector exist in these processors, but have yet to become sufficiently widely used to be explicitly discussed by the C Standard.

processor SIMD

There is no requirement in C that the operators in an expression be executed one at a time, although this is how the majority of processors have traditionally worked. The quest for performance has led processor vendors to try to perform more than one operation at the same time. Some vendors have chosen not to require translator support, the selection of the operations to execute in parallel being chosen dynamically by the processor (i.e., the latest members of the Intel x86 processor family[20]). Other vendors have introduced processors that operate on more than one data value at the same time, using the same instruction (known as *SIMD*, Single Instruction Multiple Data— pronounced sim-d). For such processors operations on arrays in a loop need not occur an element at a time; they can be performed 8, 16, 32, or however many parallel data operations are supported, elements at a time. Such processors require that translators recognize those situations where it is possible to perform the same operation on more than one value at the same time. SIMD processors used to be the preserve of up-market users, who could afford a Cray or one of the myriad of companies set up to sell bespoke systems. They are starting to become available as single-chip coprocessors for special embedded applications.[36]

A description of how arithmetic operations are performed on integer and floating-point operands at the hardware level is given in Hennessy.[15]

Coding Guidelines

During almost all of human history, natural language has been used purely in a spoken form. The need to generate and comprehend sequences of words in realtime, using a less-than-perfect processor (the human brain), restricted the complexity of reliable communication. Realtime communication also provides an opportunity for feedback between speaker and listener, allowing content to be tailored for individual consumption.

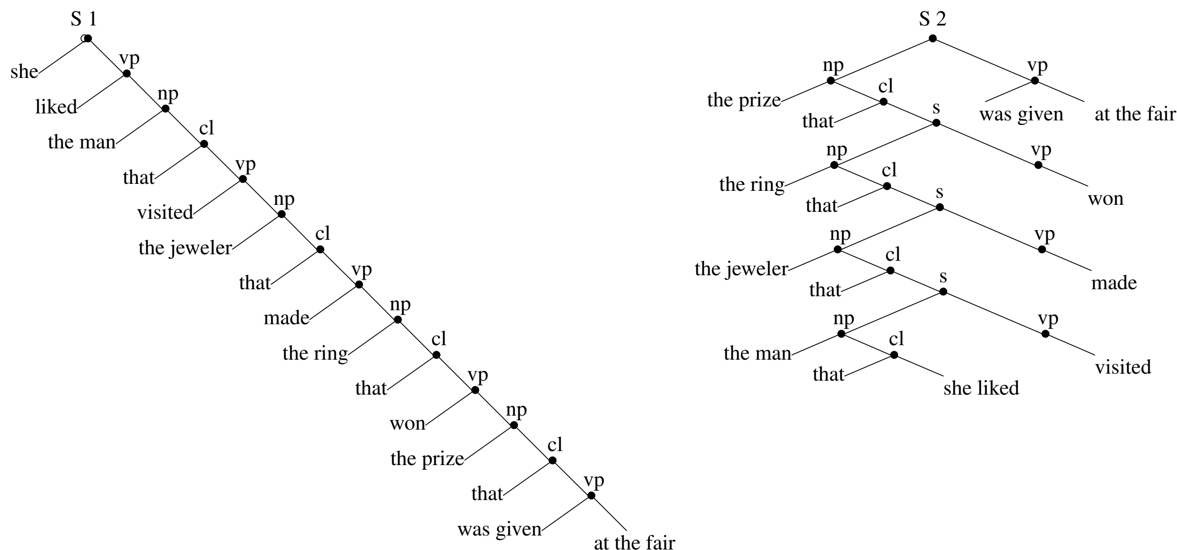

Figure 940.2: Parse tree of a sentence with no embedding (S 1) and a sentence with four degrees of embedding (S 2). Adapted from Miller and Isard.[27]

It is often said that C source code is difficult to read. This is a special case of a more general observation. Most material created by people who are untrained in communicating ideas in written form is difficult to read. The written form of communication is not constrained by the need to be created in realtime, does not offer the opportunity for immediate feedback, and may have a readership that is not known to the author. An additional contribution to the *unreadability* of source code is that it is often written by people who consider themselves to be communicating with a computer rather than communicating with another person. Writers of C statements have the time needed to write complex constructs and the specifications they are given to work from are often complex. Duplicating this complexity often requires less effort than creating a simplified representation (i.e., copying can require less effort than creating).

A study by Miller and Isard[27] investigated subjects' ability to memorize sentences that varied in their degree of embedding. The following sentences are written with increasing amounts of embedding (the parse tree of two of them is shown in Figure 940.2).

1. She liked the man that visited the jeweler that made the ring that won the prize that was given at the fair.
2. The man that she liked visited the jeweler that made the ring that won the prize that was given at the fair.
3. The jeweler that the man that she liked visited made the ring that won the prize that was given at the fair.
4. The ring that the jeweler that the man that she liked visited made won the prize that was given at the fair.
5. The prize that the ring that the jeweler that the man that she liked visited made won was given at the fair.

The results showed that subjects' ability to correctly recall wording decreased as the amount of embedding increased, although their performance did improve with practice.

Other studies of reader's performance with processing natural languages have found the following:

- People have significant comprehension difficulties when the degree of embedding in a sentence exceeds two.[5]
- Readers' ability to comprehend syntactically complex sentences is correlated with their working memory capacity, as measured by the reading span test.[22]
- Readers parse sentences left-to-right.[29] An example of this characteristic is provided by so called *garden path* sentences, in which one or more words encountered at the end of a sentence changes the parse of words read earlier:

```
The horse raced past the barn fell.
The patient persuaded the doctor that he was having trouble with to leave.
While Ron was sewing the sock fell on the floor.
Joe put the candy in the jar into my mouth.
The old train their dogs.
```

  In computer languages, the extent to which an identifier, operand, or subexpression encountered later in a full expression might change the tentative meaning assigned to what appears before it is not known.

reading span

How to readers represent expressions in memory? Two particular representations of interest here are the spoken and visible forms. Developers sometimes hold the sound of the spoken form of an expression in short-term memory; they also fix their eyes on the expression. The expression becomes the focus of attention. (This visible form of an expression, the number of characters it occupies on a line and possibly other lines, represents another form of information storage.)

Complicated expressions might be visually broken up into chunks that can be comprehended on an individually basis. The comprehension of these individual chunks then being combined to comprehend the complete expression (particularly for expressions having a boolean role). These chunks may be based on the visible form of the expression, the logic of the application domain, or likely reader cognitive limits. This issue is discussed in more detail elsewhere.

boolean role

memory chunking

The possible impact of the duration of the spoken form of an identifier appearing in an expression on reader memory resources is discussed elsewhere.

identifier primary spelling issues

Expressions that do not generate side effects are discussed elsewhere. The issue of spacing between tokens is discussed elsewhere. Many developers have a mental model of the relative performance of operators and sometimes use algebraic identities to rewrite an expression into a form that uses what they believe to be the faster operators. In some cases some identities learned in school do not always apply to C operators (e.g., if the operands have a floating-point type).

dead code

words white space between

The majority of expressions contain a small number of operators and operands (see Figure ??, Figure ??, Figure ??, and Figure ??). The following discussion applies, in general, to the less common, longer (large number of characters in its visible representation), more complex expressions.

Readers of the source sometimes have problems comprehending complex expressions. The root cause of these problems may be incorrect knowledge of C or human cognitive limitations. The approach taken in these coding guideline subsections is to recommend, where possible, a usage that attempts to nullify the effects of incorrect developer knowledge. This relies on making use of information on common developer mistakes and misconceptions. Obviously a minimum amount of developer competence is required, but every effort is made to minimize this requirement. Documenting common developer misconceptions and then recommending appropriate training to improve developers' knowledge in these areas is not considered to be a more productive approach. For instance, a guideline recommending that developers memorise the 13 different binary operator precedence levels does not protect against the reader who has not committed them to memory, while a guideline recommending the use of parenthesis does protect against subsequent readers who have incorrect knowledge of operator precedence levels.

943 operator precedence

943.1 expression shall be parenthesized

An expression might only be written once, but it is likely to be read many times. The developer who wrote the expression receives feedback on its behavior through program output, during testing, which is affected by its evaluation. There is an opportunity to revise the expression based on this feedback (assumptions may

still be held about the expression— order of evaluation— because the translator used happens to meet them). There is very little feedback to developers when they read an expression in the source; incorrect assumptions are likely to be carried forward, undetected, in their attempts to comprehend a function or program.

The complexity of an expression required to calculate a particular value is dictated by the application, not the developer. However, the author of the source does have some control over how the individual operations are broken down and how the written form is presented visually.

Many of these issues are discussed under the respective operators in the following C sentences. The discussion here considers those issues that relate to an expression as a whole. While there are a number of different techniques that can be used to aid the comprehension of a long or semantically complex expression, your author does not have sufficient information to make any reliable cost-effective recommendations about which to apply in most cases. Possible techniques for reducing the cost of developer comprehension of an expression include:

- A comment that briefly explains the expression, removing the need for a reader to deduce this information by analyzing the expression.
- A complex expression might be split into smaller chunks, potentially reducing the maximum cognitive load needed to comprehend it (this might be achieved by splitting an assignment statement into several assignment statements, or information hiding using a macro or function).
- The operators and operands could be laid out in a way that visually highlights the structure of the semantics of what the expression calculates.

The last two suggestions will only apply if there are semantically meaningful subexpressions into which the full expression can be split.

expression visual layout

Visual layout

An expression containing many operands may need to be split over more than one line (the term *long* expression is often used, referring to the number of characters in its visible form). Are there any benefits in splitting an expression at any particular point, or in visually organizing the lines in any particular manner? There are a number of different circumstances under which an expression may need to be split over several lines, including:

- The line containing the expression may be indented by a large amount. In this case even short, simple expressions may need to be split over more than one line. The issue that needs to be addressed in this case is the large indentation; this is discussed elsewhere.

statement visual layout

- The operands of the expression refer to identifiers that have many characters in their spelling. The issue that needs to be addressed in this case is the spelling of the identifiers; this is discussed elsewhere.

visual skimming

- The expression contains a large number of operators. The rest of this subsection discusses this issue.

Expressions do not usually exist in visual isolation and are not always read in isolation. Readers may only look at parts of an expression during the process of scanning the source, or they may carefully read an expression. (The issue of how developers read source is discussed elsewhere.) Some of the issues involved in the two common forms of code reading include the following:

reading kinds of

- During a careful reading of an expression reducing the cost of comprehending it, rather than differentiating it from the surrounding code, is the priority. Whether a reader has the semantic knowledge needed to comprehend how the components of an expression are mapped to the application domain is considered to be outside the scope of these coding guideline subsections. Organizing the components of an expression into a form that optimizes the cognitive resources that are likely to be available to a reader is within the scope of these coding guideline subsections. Experience suggests that the cognitive resource most likely to be exceeded during expression comprehension is working memory capacity.

Organizing an expression so that the memory resources needed at any point during the comprehension of an expression do not exceed some maximum value (i.e., the capacity of a typical developer) may reduce comprehension costs (e.g., by not requiring the reader to concentrate on saving temporary information about the expression in longer-term memory). Studies have found that human memory performance is improved if information is split into meaningful *chunks*. Issues, such as how to split an expression into *chunks* and what constitutes a recognizable structure, are skills that developers learn and that are not yet amenable to automatic solution. The only measurable suggestion is based on the phonological loop component of working memory, which can hold approximately two seconds worth of sound. If the spoken form of a chunk takes longer than two seconds to say (by the person trying to comprehend it), it will not be able to fit completely within this form of memory. This provides an upper bound on one component of chunk size (the actual bound may be lower).

memory chunking

phonological loop

- When scanning the code, being able to quickly look at its components, rather than comprehending it in detail, is the priority; that is, differentiating it from the surrounding code, or at least ensuring that different lines are not misinterpreted as being separate expressions. The edges of the code (the first non-white-space characters at the start and end of lines) are often used as reference points when scanning the source. For instance, readers quickly scanning down the left edge of source code might assume that the first identifier on a line is either modified in some way or is a function call.

Table 940.1: Occurrence of a token as the last token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). Based on the visible form of the `.c` files.

Token	% Occurrence of Token	% Last Token on Line	Token	% Occurrence of Token	% Last Token on Line
;	92.2	36.0	**#else**	89.1	0.2
* ... *\	97.9	8.4	**int**	5.3	0.2
)	20.6	8.3	||	23.7	0.2
{	86.7	8.1	|	12.3	0.1
}	78.9	7.4	+	3.8	0.1
,	13.9	6.1	?:	7.3	0.0
:	74.3	1.7	?	7.1	0.0
header-name	97.7	1.5	**do**	21.3	0.0
\\	100.0	0.9	**#error**	25.1	0.0
#endif	81.9	0.8	:b	7.2	0.0
else	42.2	0.7	**double**	3.1	0.0
string-literal	8.0	0.4	^	3.1	0.0
void	18.2	0.4	**union**	6.2	0.0
&&	17.8	0.2			

One way of differentiating multiline expressions is for the start, and end, of the lines to differ from other lines containing expressions. One possible way of differentiating the two ends of a line is to use tokens that don't commonly appear in those locations. For instance, lines often end in a semicolon, not an arithmetic operator (see Table 940.1), and at the start of a line additional indentation for the second and subsequent lines containing the same expression will set it off from the surrounding code. Some developers prefer to split expressions just before binary operators. However, the appearance of an operator as the last non-white-space character is more likely to be noticed than the nonappearance of a semicolon (the human visual system is better at detecting the presence rather than the absence of a stimulus). Of course, the same argument can be given for an identifier or operator at the start of a line. These coding guidelines give great weight to existing practice. In this case this points to splitting expressions before/after binary operators; however, there is insufficient evidence of a worthwhile benefit for any guideline recommendation.

distinguishing features

Optimization

Many developers have a view of expressions that treats them as standalone entities. This viewpoint is often extended to translator behavior, which is then thought to optimize and generate machine code on an expression-by-expression basis. This developer though process leads on to the idea that performing as many operations as much as possible within a single expression evaluation results in translators generating more efficient machine code. This thought process is not cost effective because the difference in efficiency of expressions written in this way is rarely sufficient to warrant the cost, to the current author and subsequent readers, of having to comprehend them.

translator optimizations

Whether a complex expression results in more, or less, efficient machine code will depend on the optimization technology used by the translator. Although modern optimization technology works on units significantly larger than an expression, there are still translators in use that operate at the level of individual expressions.

Example

```
extern int g(void);
extern int a,
           b;

void f(void)
{
a + b;           /* A computation. */
a;               /* An object. */
g();             /* A function. */
a = b;           /* Generates side effect. */
a = b , a + g(); /* A combination of all of the above. */
}
```

Usage

partial redundancy elimination

A study by Bodík, Gupta, and Soffa[6] found that 13.9% of the expressions in SPEC95 were partially redundant, that is, their evaluation is not necessary under some conditions.

full expression

See Table **??** for information on occurrences of full expressions, and Table **??** for visual spacing between binary operators and their operands.

Table 940.2: Occurrence of a token as the first token on a physical line (as a percentage of all occurrences of that token and as a percentage of all lines). /* *new-line* */ denotes a comment containing one or more new-line characters, while /* ... */ denotes that form of comment on a single line. Based on the visible form of the `.c` files.

Token	% First Token on Line	% Occurrence of Token	Token	% First Token on Line	% Occurrence of Token
default	0.2	99.9	**volatile**	0.0	50.0
#	5.0	99.9	**int**	1.8	47.0
typedef	0.1	99.8	**unsigned**	0.7	46.8
static	2.1	99.8	**struct**	1.1	38.9
for	0.8	99.7	**const**	0.1	35.5
extern	0.2	99.6	**char**	0.5	30.5
switch	0.3	99.4	**void**	0.6	28.7
case	1.6	97.8	***v**	0.5	28.7
* new-line *\	13.7	97.7	**++v**	0.0	27.8
register	0.2	95.0	**signed**	0.0	27.2
return	3.3	94.5	**&&**	0.3	21.2
goto	0.4	94.1	identifier	31.1	20.8
if	6.9	93.6	||	0.2	18.4
break	1.2	91.8	**--v**	0.0	17.9
continue	0.2	91.3	**short**	0.0	16.0
}	8.3	88.3	**#error**	0.0	15.6
do	0.1	87.3	*string-literal*	0.6	12.4
while	0.4	85.2	**sizeof**	0.1	11.3
enum	0.1	73.7	**long**	0.1	10.1
\\	0.6	70.8	*integer-constant*	2.2	6.6
else	1.1	70.2	**?**	0.0	5.6
union	0.0	63.3	**&v**	0.1	5.2
* ... *\	5.4	62.6	**-v**	0.1	5.0
{	5.1	54.9	**?:**	0.0	5.0
float	0.0	54.0	|	0.0	4.2
double	0.0	53.6	*floating-constant*	0.0	4.1

value profiling

Recent research[10, 14, 24] has found that for a few expressions, a large percentage of their evaluations return the same value during program execution. Depending on the expression context and the probability of the same value occurring, various optimizations become worthwhile[28] (0.04% of possible expressions evaluating to the same value a sufficient percentage of the time in a context that creates a worthwhile optimization opportunity). Some impressive performance improvements (more than 10%) have been obtained for relatively small numbers of optimizations. Citron[12] studied how processors might detect previously executed instruction sequences and reuse the saved results (assuming the input values were the same).

Table 940.3: Breakdown of invariance by instruction types. These categories include integer loads (*ILd*), floating-point loads (*FLd*), load address calculations (*LdA*), stores (*St*), integer multiplication (*IMul*), floating-point multiplication (*FMul*), floating-point division (*FDiv*), all other integer arithmetic (*IArth*), all other floating-point arithmetic (*FArith*), compare (*Cmp*), shift (*Shft*), conditional moves (*CMov*), and all other floating-point operations (*FOps*). The first number shown is the percent invariance of the topmost value for a class type, while the number in parenthesis is the dynamic execution frequency of that type. Results are not shown for instruction types that do not write a register (e.g., branches). Adapted from Calder, Feller, and Eustace.[10]

Program	ILd	FLd	LdA	St	IMul	FMul	FDiv	IArth	FArth	Cmp	Shft	CMov	FOps
compress	44(27)	0(0)	88(2)	16(9)	15(0)	0(0)	0(0)	11(36)	0(0)	92(2)	14(9)	0(0)	0(0)
gcc	46(24)	83(0)	59(9)	48(11)	40(0)	30(0)	31(0)	46(28)	0(0)	87(3)	54(7)	51(1)	95(0)
go	36(30)	100(0)	71(13)	35(8)	18(0)	100(0)	0(0)	29(31)	0(0)	73(4)	42(0)	52(1)	100(0)
ijpeg	19(18)	73(0)	9(11)	20(5)	10(1)	68(0)	0(0)	15(37)	0(0)	96(2)	17(21)	15(0)	98(0)
li	40(30)	100(0)	27(8)	42(15)	30(0)	13(0)	0(0)	56(22)	0(0)	93(2)	79(3)	60(0)	100(0)
perl	70(24)	54(3)	81(7)	59(15)	2(0)	50(0)	19(0)	65(22)	34(0)	87(4)	69(6)	28(1)	51(1)
m88ksim	76(22)	59(0)	68(8)	79(11)	33(0)	53(0)	66(0)	64(28)	100(0)	91(5)	66(6)	65(0)	100(0)
vortex	61(29)	99(0)	46(6)	65(14)	9(0)	4(0)	0(0)	70(31)	0(0)	98(2)	40(3)	20(0)	100(0)

Studies of operand values during program execution (investigating ways of minimizing processor power consumption) have found that a significant percentage of these values use fewer representation bits than are available to them (i.e., they are small positive quantities). Brooks and Martonosi[7] found that 50% of operand values in SPECint95 required less than 16 bits.

Table 940.4: Number of objects defined (in a variety of small multimedia and scientific programs) to have types represented using a given number of bits (i.e., mostly 32-bit `int`) and number of objects having a maximum bit-width usage (i.e., number of bits required to represent any of the values stored in the object; rounded up to the nearest byte boundary). Adapted from Stephenson,[33] whose analysis was performed by static analysis of the source.

Bits	Objects Defined	Objects Requiring Specified Bits
1	0	203
8	7	134
16	27	108
32	686	275

---

object modified once between sequence points

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. DR287) **941**

Commentary

object 942 read and modified between sequence points

A violation of this requirement results in undefined behavior. If an object is modified more than once between sequence points, the standard does not specify which modification is the last one. The situation can be even more complicated when the same object is read and modified between the same two sequence points. This requirement does not specify exactly what is meant by *object*. For instance, the following full expression may be considered to modify the object `arr` more than once between the same sequences points.

```
1 int arr[10];
2
3 void f(void)
4 {
5 arr[1]=arr[2]++;
6 }
```

C++

5p4 *Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression.*

The C++ Standard avoids any ambiguity in the interpretation of *object* by specifying scalar type.

Other Languages

In most languages assignment is not usually considered to be an operator, and assignment is usually the only operator that can modify the value of an object; other operators that modify objects are not often available. In such languages function calls is often the only mechanism for causing more than one modification between two sequence points (assuming that such a concept is defined, which it is not in most languages).

Common Implementations

Most implementations attempt to generate the best machine code they can for a given expression, independently of how many times the same object is modified. Since the surrounding context often has a strong influence on the code generated for an expression, it is possible that the evaluation order for the same expression will depend on the context in which it occurs.

Coding Guidelines

As the example below shows, a guideline recommendation against modifying the same object more than once between two adjacent sequence points is not sufficient to guarantee consistent behavior. A guideline recommendation that is sufficient to guarantee such behavior is discussed elsewhere.

944.1 expression same result for all evaluation orders

Example

In following the first expression modifies `glob` more than once between sequence points:

```
1  extern int glob,
2             valu;
3
4  void f(void)
5  {
6  glob = valu + glob++;                    /* Undefined behavior. */
7  glob = (glob++, glob) + (glob++, glob); /* Undefined and unspecified behavior. */
8  }
```

Possible values for `glob`, immediately after the sequence point at the semicolon punctuator, include

- `valu + glob`
- `glob + 1`
- `((valu + glob) && 0xff00) | ((glob + 1) && 0x00ff)`

The third possibility assumes a 16-bit representation for **`int`**— a processor whose store operation updates storage a byte at a time and interleaves different store operations. In the second expression the evaluation of the left operand of the comma operator may be overlapped. For instance, a processor that has two arithmetic logic units may split the evaluation of an expression across both units to improve performance. In this case `glob` is modified more than once between sequence points. Also, the order of evaluation is unspecified.

944 expression order of evaluation

In the following:

```
1  struct T {
2            int mem_1;
3            char mem_2;
4           } *p_t;
5
6  extern void f(int, struct T);
7
8  void g(void)
9  {
10 int loc = (*p_t).mem_1++ + (*p_t).mem_2++;
11         f((*p_t).mem_1++,   *p_t)        ; /* Modify part of an object. */
12 }
```

there is an object, `*p_t`, containing various subobjects. It would be surprising if a modification of a subobject (e.g., `(*p_t).mem_1`) was considered to be the same as a modification of the entire object. If it was, then the two modifications in the initialization of expression for `loc` would result in undefined behavior. In the call to `f` the first argument modifies a subobject of the object `*p_t`, while the second argument accesses all of the object `*p_t` (and undefined behavior is to be expected, although not explicitly specified by the standard).

---

942 Furthermore, the prior value shall be read only to determine the value to be stored.[71)]

object read and modified between sequence points

Commentary

In expressions, such as `i++` and `i = i*2`, the value of the object `i` has to be read before its value can be operated on and a potentially modified value written back. The semantics of the respective operators ensure that this ordering between operations occurs.

In expressions, such as `j = i + i--`, the object `i` is read twice and modified once. The left operand of the binary plus operator performs a read of `i` that is not necessary to determine the value to be stored into it. The behavior is therefore undefined. There are also cases where the object being modified occurs on the left side of an assignment operator; for instance, `a[i++] = i` contains two reads from `i` to determine a value and a modification of `i`.

Coding Guidelines

expression 944.1 same result for all evaluation orders

The generalized case of this undefined behavior is covered by a guideline recommendation dealing with evaluation order.

---

expression grouping operator precedence

operator 943 precedence
operator 955 associativity

The grouping of operators and operands is indicated by the syntax.[72)] 943

Commentary

The two factors that control the grouping are precedence and associativity.

Other Languages

Most programming languages are defined in terms of some form of formal, or semiformal, BNF syntax notation. While a few languages allow operators to be overloaded, they usually keep their original precedence. In APL all operators have the same precedence and expressions are interpreted right-to-left (e.g., `1*2+3` is equivalent to `1*(2+3)`). The designers of Ada recognized[16] that developers do not have the same amount of experience handling the precedence of the logical operators as they do the arithmetic operators. An expression containing a sequence of the same logical binary operator need not be parenthesized, but a sequence of different logical binary operators must be parenthesized (parentheses are not required for unary **`not`**).

Common Implementations

Most implementations perform the syntax analysis using a table-driven parser. The tables for the parser are generated using some automatic tool (e.g., `yacc`, `bison`) that takes a LALR(1) grammar as input. The grammar, as specified in the standard, and summarized in Annex A, is not in LALR(1) form as specified. It is possible to transform it into this form, an operation that is often performed manually.

Coding Guidelines

Developers over learn various skills during the time they spend in formal education. These skills include the following:

- The order in which words are spoken is generally intended to reduce the comprehension effort needed by the listener. The written form of languages usually differs from the spoken form. In the case of English, it has been shown[29] that readers parse its written form left-to-right, the order in which the words are written. It has not been confirmed that readers of languages written right-to-left parse them in a right-to-left order.
- Many science and engineering courses require students to manipulate expressions containing operators that also occur in source code. Students learn, for instance, that in an expression containing a multiplication and addition operator, the multiplication is performed first. Substantial experience is gained over many years in reading and writing such expressions. Knowledge of the ordering relationships between assignment, subtraction, and division also needs to be used on a very frequent basis. Through constant practice, knowledge of the precedence relationships between these operators becomes second nature; developers often claim that they are natural (they are not, it is just constant practice that makes them appear so).

reading practice

Your author knows of no research studying how developers read expressions. The assumption made in these coding guidelines subsections is that developers' extensive experience reading prose is a significant factor affecting how they read source code. Given the significant differences in the syntactic structure of natural languages (see Figure 943.1) the possibility of an optimal visual expression organization, which is universal to all software developers, seems remote.

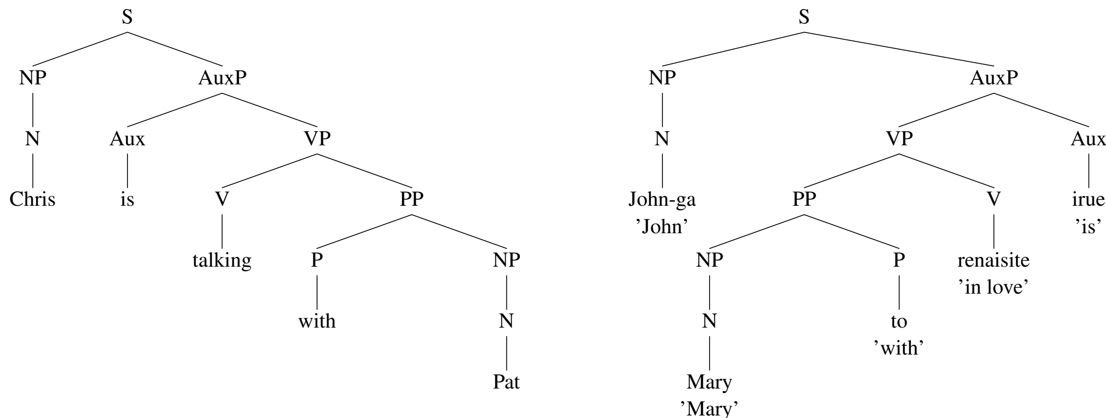

Figure 943.1: English ("Chris is talking with Pat") and Japanese ("John-ga Mary to renaisite irue") language phrase structure for sentences of similar complexity and structure. While the Japanese structure may seem back-to-front to English speakers, it appears perfectly natural to native speakers of Japanese. Adapted from Baker.[4]

While developers are likely to have had significant previous experience in reading and writing the commonly occurring C operators, few of developers attain the same level of practice with the other operators. Consequently, developer knowledge of the precedence levels of other operators is often faulty (which does not prevent them from being willing to jump to conclusions). A study by Jones[21] found that subjects (developers with an average of 14 years professional experience) failed to correctly parenthesize 33% of expressions containing two binary operators.

Factors that have been found to effect developer operator precedence decisions include the relative spacing between operators and the names of the operands.

operator relative spacing
operand name context

One solution to faulty developer knowledge of operator precedence levels is to require the parenthesizing of all subexpressions (rendering any precedence knowledge the developer may have, right or wrong, irrelevant). Such a requirement often brings howls of protest from developers. Completely unsubstantiated claims are made about the difficulties caused by the use of parentheses. (The typing cost is insignificant; the claimed unnaturalness is caused by developers who are not used to reading parenthesized expressions, and so on for other developer complaints.) Developers might correctly point out that the additional parentheses are redundant (they are in the sense that the precedence is defined by C syntax and the translator does not require them); however, they are not redundant for readers who do not know the correct precedence levels.

An alternative to requiring parentheses for any expression containing more than two operators is to provide a list of special where it is believed that developers are very unlikely to make mistakes (these cases have the advantage of being common). Listing special cases could either be viewed as the thin end of the edge that eventually drives out use of parentheses, or as an approach that gradually overcomes developer resistance to the use of parentheses.

When combined with binary operators, the correct order of evaluation of unary operators is simple to deduce and developers are unlikely to make mistakes in this case. However, the ordering relationship, when a unary operator is applied to the result of another unary operator, is easily confused when unary operators appear to both the left and right of the same operand. This is a case where the use of parentheses removes the possibility of reader mistakes.

In C both function calls and array indexing are classified as operators. There is likely to be considerable developer resistance to parenthesizing these operators because they are not usually thought of in these terms (they are not operators in many other languages); they are also unary operators and the pair of characters used is often considered as forming bracketed subexpressions.

In the following guideline recommendation the expression within

- the square brackets used as an array subscript operator are treated as equivalent to a pair of matching

parentheses, not as an operator; and

- the arguments in a function invocation are each treated as full expressions and are not considered to be part of the rest of the expression that contains the function invocation for the purposes of the deviations listed.

operator associativity 955

An issue related to precedence, but not encountered so often, is associativity, which deals with the evaluation order of operands when the operators have the same precedence. If the operands in an expression have different types, the evaluation order specifies the pairings of operand types that need to go through the usually arithmetic conversions.

usual arithmetic conversions

Cg 943.1 Each subexpression of a full expression containing more than one operator shall be parenthesized.

Dev 943.1 A full expression that only contains zero or more additive operators and a single assignment operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more multiplication, division, addition, and subtraction operators and a single assignment operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more additive operators and a single relational or equality operator need not be parenthesized.

Dev 943.1 A full expression that only contains zero or more multiplicative and additive operators and a single relational or equality operator need not be parenthesized.

Developers appear to be willing to accept the use of parentheses in so-called complex expressions. (An expression containing a large number of operators, or many different operators, is often considered complex; exactly how many operators is needed varies depending on who is asked.) Your author's unsubstantiated claim is that more time is spent discussing under what circumstances parentheses should be used than would be spent fully parenthesizing every expression developers ever write. Management needs to stand firm and minimize discussion on this issue.

Example

```
1  *p++;            /* Equivalent to *(p++); */
2
3  (char)!+-~*++p; /* Operators applied using an inside out order. */
4
5  ;m<-++pq++->m;  /* The token -> is not usually thought of as a unary operator. */
6
7   a = b = c;      /* Equivalent to  a = (b  = c); */
8   x + y + z;      /* Equivalent to (x +  y) + z ; */
```

---

expression order of evaluation

Except as specified later (for the function-call **()**, **&&**, **||**, **?:**, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified. 944

Commentary

The exceptional cases are all operators that involve a sequence point during their evaluation.

strictly conforming program output shall not

This specification, from the legalistic point of view, renders all expressions containing more than one operand as containing unspecified behavior. However, the definition of strictly conforming specifies that the output must not be dependent on any unspecified behavior. In the vast majority of cases all orders of evaluation of an expression deliver the same result.

Other Languages

Most languages do not define an order of evaluation for expressions. Snobol 4 defines a left-to-right order of evaluation for expressions. The Ada Standard specifies " . . . in some order that is not defined", with the intent[16] that there is some order and that this excludes parallel evaluation. Java specifies a left-to-right evaluation order. The left operand of a binary operator is fully evaluated before the right operand is evaluated.

Common Implementations

Many implementations build an expression tree while performing syntax analysis. At some point this expression tree is walked (often in preorder, sometimes in post-order) to generate a lower-level representation (sometimes a high-level machine code form, or even machine code for the executing host). An optimizer will invariably reorganize this tree (if not at the C level, then potentially though code motion of the intermediate or machine code form).

Even the case where a translator performs no optimizations and the expression tree has a one-to-one mapping from the source, it is not possible to reliably predict the order of evaluation. (There is more than one way to walk an expression tree matching higher-level constructs and map them to machine code.) As a general rule, increasing the number of optimizations performed increases the unpredictability of the order of expression evaluation.

Coding Guidelines

The order of evaluation might not affect the output from a program, but it can affect its timeliness. In:

```
1   printf("Hello ");
2   x = time_consuming_calculation() + print("World\n");
```

the order in which the two function calls on the right-hand side of the assignment are invoked will affect how much delay occurs between the output of the character sequences **Hello** and **World**.

In the expression `i = func(1) + func(2)`, the value assigned to `i` may, or may not, depend on the order in which the two invocations of `func` occur. Also the order of invocation may result in other objects having differing values. The sequence point that occurs prior to each function being invoked does not prevent these different behaviors from occurring. Sequence points are too narrow a perspective; it is necessary to consider the expression evaluation as a whole.

function call sequence point

Cg 944.1

The state of the C abstract machine, after the evaluation of a full expression, shall not depend on the order of evaluation of subexpressions or the order in which side effects take place.

Example

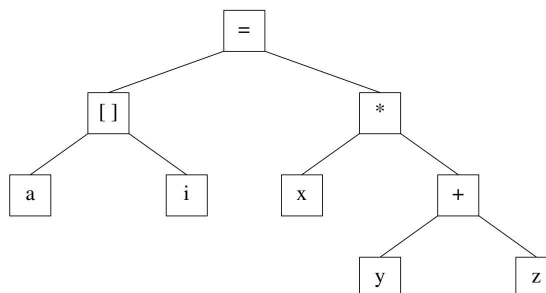

Figure 944.1: A simplified form of the kind of tree structure that is likely to be built by a translator for the expression `a[i]=x*(y+z)`.

```
1  #include <stdio.h>
2
3  extern volatile int glob;
4
5  void f(void)
6  {
7  int loc = glob + glob * glob;
8
9  /*
10  * In the following the only constraints on the order in
11  * which characters appear are that:
12  *    ) x must be output before y and
13  *    ) a must be output before b
14  */
15 loc = printf("x"),printf("y") + printf("a"),printf("b");
16 }
```

bitwise operators

Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. 945

Commentary

This defines the term *bitwise operators*.

C++

The C++ Standard does not define the term bitwise operators, although it does use the term bitwise in the description of the &, ^ and | operators.

Other Languages

PL/1 has a bit data type and supports bitwise operations on values having such types.

Coding Guidelines

Bitwise operations provide a means for manipulating an object's underlying representation. They also provide a mechanism for using a new data type, the *bit-set*. There is a guideline recommendation against making use of an object's underlying representation. The following discussion looks at possible deviations to this recommendation.

representation information using ??

Performance issues

The result of some sequences of bitwise operations are the same as some arithmetic operations. For instance, left-shifting and multiplication by powers of two. There is a general belief among developers that processors execute these bitwise instructions faster than the arithmetic instructions. The extent to which this belief is true varies between processors (it tends to be greater in markets where processor cost has been traded-off against performance). The extent to which a translator automatically performs these mappings will depend on whether it has sufficient information about operand values and the quality of the optimizations it performs. If performance is an issue, and the translator does not perform the desired optimizations, the benefit of using bitwise operations may outweigh any other factors that increase costs, including:

cognitive switch

- Subsequent reader comprehension effort— switching between thinking about bitwise and arithmetic operations will require at least a cognitive task switch.
- The risk that a change of representation in the types used will result in the bitwise mapping used failing to apply. This may cause faults to occur.
- Treating the same object as having different representations, in different parts of the visible source requires readers to use two different mental models of the object. Two models may require more cognitive effort to recall and manipulate than one, and interference may also occur in the reader's memory, potentially leading to mistakes being made.

Dev ??

A program may use bitwise operators to perform arithmetic operations provided a worthwhile cost/benefit has been shown to exist.

Bit-set

Some applications, or algorithms, call for the creation of a particular kind of set data type (in mathematics a set can hold many values, but only one of each value). The term commonly used to describe this particular kind of set is *bit-set*, which is essentially an array of boolean values. The technique used to implement this bit-set type is to interpret every bit of an integer type as representing a member of the set. (When the bit is set, the member is considered to be in the set; when it is not set, the member is not present.) The number of members that can be represented using this technique is limited by the number of bits available in an integer type. This technique essentially provides both storage and performance optimization. An alternative representation technique is a structure type containing a member for each member of the bit-set, and appropriate functions for testing and setting these members.

While the boolean role is defined in terms of operations that may be performed on a value having certain properties, it is possible to define a bit-set role in terms of the operations that may be performed on a value having certain properties.

boolean role

bit-set role

An object having an integer type, or value having an integer type has a bit-set role if it appears as the operand of a bitwise operator or the object is assigned a value having a bit-set role.

bitwise operator result bit-set role

For the purpose of these guideline recommendations the result of a bitwise operator has a bit-set role.

numeric role

An object having an integer type, or value having an integer type has a numeric role if it appears as the operand of an arithmetic operator or the object is assigned a value having a numeric role. Objects having a floating type always have a numeric role.

arithmetic operator result numeric role

For the purpose of these guideline recommendations the result of an arithmetic operator is defined to have a numeric role.

The sign bit, if any, in the value representation shall not be used in representing a bit-set. (This restriction is needed because, if an operand has a signed type, the integer promotions or the usual arithmetic conversions can result in an increase in the number of bits used in the value representation.)

integer promotions
usual arithmetic conversions

Dev ??

An object having a bit-set role that appears as the operand of a bitwise operator is not considered to be making use of representation information.

Example

Bitwise operations allow several conditions to be checked at the same time.

```
1  #define R_OK (0x01)
2  #define W_OK (0x02)
3  #define X_OK (0x04)
4
5  _Bool f(unsigned int permission)
6  {
7  return (permission & (R_OK | W_OK)) == 0;
8  }
```

---

946 These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

bitwise operations signed types

Commentary

The choice of behavior was largely influenced by what the commonly available processors did at the time the standard was originally written. In some cases there is a small set of predictable behaviors; for instance, left-shift can exhibit undefined behavior, while under the same conditions right-shift is implementation-defined.

left-shift undefined
right-shift negative value

Efficiency of execution has been given priority over specifying the exact behavior (which may be inefficient to implement on some processors).

Warren[35] provides an extensive discussion of calculations that can be performed and information obtained via bitwise operations on values represented in two's complement notation.

C++

These operators exhibit the same range of behaviors in C++. This is called out within the individual descriptions of each operator in the C++ Standard.

Other Languages

The issues involved are not specific to C. They are caused by the underlying processor representations of integers and how instructions that perform bitwise operations on these types are defined to operate. As such, other languages that support bitwise operations also tend to exhibit the same kinds of behaviors.

Coding Guidelines

The issues involved in using operators that rely on undefined and implementation-defined behavior are discussed under the respective operators.

---

exception condition

If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined. 947

Commentary

This defines the term *exceptional condition*. Note that the wording does not specify that an exception is raised, but that the condition is exceptional (i.e., unusual). These exceptional conditions can only occur for operations involving values that have signed integer types or real types.

There are only a few cases where results are not mathematically defined (e.g., divide by zero). The more common case is the mathematical result not being within the range of values supported by its type (a form of overflow). For operations on real types, whether values such as infinity or NaN are representable will depend on the representation used. In the case of IEC 60559 there is always a value that is capable of representing the result of any of its defined operations.

C90

The term *exception* was defined in the C90 Standard, not *exceptional condition*.

C++

5p5

> *If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression is a constant expression (5.19), in which case the program is ill-formed.*

The C++ language contains explicit exception-handling constructs (Clause 15, **`try`**/**`throw`** blocks). However, these are not related to the mechanisms being described in the C Standard. The term exceptional condition is not defined in the C sense.

Other Languages

Few languages define the behavior when the result of an expression evaluation is not representable in its type. However, Ada does define the behavior— it requires an exception to be raised for these cases.

Common Implementations

In most cases translators generate the appropriate host processor instruction to perform an operation. Whatever behavior these instructions exhibit, for results that are not representable in the operand type, is the implementation's undefined behavior. For instance, many processors trap if the denominator in a division operation is zero. It is rare for an implementation to attempt to detect that the result of an expression

evaluation overflows the range of values representable in its type. Part of the reason is efficiency and part because of developer expectations (an implementation is not expected to do it).

On many processors the instructions performing the arithmetic operations are defined to set a specified bit if the result overflows. However, the unit of representation is usually a register (some processors have instructions that operate on a subdivision of a register— a halfword or byte). For C types that exactly map to a processor register, detecting an overflow is usually a matter of generating an additional instruction after every arithmetic operation (branch on overflow flag set). Complications can arise for mixed signed/unsigned expressions if the processor also sets the overflow flag for operations involving unsigned types. (The Intel x86, IBM 370 set the carry flag in this case; SPARC has two add instructions, one that sets the carry flag and one that does not.) A few processors have versions of arithmetic instructions that are either defined to trap on overflow (often limited to add and subtract, e.g., MIPS) or provide a mechanism for toggling trap on overflow (IBM 370, HP–was DEC– VAX).

Example

In the following the multiplication by `LONG_MAX` will deliver a result that is not representable in a **`long`**.

```
1  extern int i;
2  extern long j;
3
4  void f(void)
5  {
6  i = 30000;
7  j = i * LONG_MAX;
8  }
```

---

948 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[73)]

effective type

Commentary

This defines the term *effective type*, which was introduced into C99 to deal with objects having allocated storage duration. In particular, to provide a documented basis for optimizers to attempt to work out which objects might be aliased, with a view to generating higher-quality machine code. Knowing that a referenced object is not aliased at a particular point in the program can result in significant performance improvements (e.g., it might be possible to deduce that its value can be held in a register throughout the execution of a critical loop rather than loaded from storage on every iteration).

Computing alias information can be very resource (processor time and storage needed) intensive. To reduce this overhead, translator vendors try to make simplifying assumptions. One assumption commonly made is that pointers to `type_A` are disjoint from pointers to `type_B`. The concept of effective type provides a mechanism for knowing the possible types that an object can be referenced through. If the same object is accessed using effective types that do not meet the requirements specified in the standard the behavior is undefined; one possible behavior is to do what an optimizing translator happens to do based on the assumption that accesses through different effective types do not occur.

960 object value accessed if type

Storing a value into an object that has a declared type, through an lvalue having a different type, does not change that object's effective type.

C90

The term *effective type* is new in C99.

C++

The term *effective type* is not defined in C++. A type needs to be specified when the C++ new operator is used. However, the C++ Standard includes the C library, so it is possible to allocate storage via a call to the `malloc` library function, which does not associate a type with the allocated storage.

Common Implementations

The RTC tool[25] performs type checking on accesses to objects during program execution. The type information associated with every storage location written to specifies the number of bytes in the type and one of unallocated, uninitialized, integer, real, or pointer. The type of a write to a storage location is checked against the declared type of that location, if any, and the type of a read from a location is checked against the type of the value last written to it.

Coding Guidelines

While an understanding of effective type might be needed to appreciate the details of how library functions such as `memcpy` and `memcmp` operate, developers rarely need to get involved in this level of detail.

If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. 949

Commentary

Only objects with allocated storage duration have no declared type. The type is assigned to such an object through a value being stored into it in name only; there is no requirement for this information to be represented during program execution (although implementations designed to aid program debugging sometimes do so). The type of an object with allocated storage duration is potentially changed every time a value is stored into it. A parallel can be drawn between such an object and another one having a union type.

Storing a value through an lvalue occurs when the left operand of an assignment operator is a dereferenced pointer value. The effective type is derived from the dereferenced pointer type in this case.

The character types are special in that they are the types often used to access the individual bytes in an object (e.g., to copy an object). This usage is sufficiently common that the Committee could not mandate that an object modified via an lvalue having a character type will only be accessed via a character type (it would also create complications for the specification of some of the library functions— e.g., `memcpy`.) An object having allocated storage duration can only have a character type as its effective type if it is accessed using such a type.

effective type 959 lvalue used for access

Other Languages

Many languages that support dynamic storage allocation require that a type be associated with that allocated storage. Some languages (e.g., awk) allocate storage implicitly without the need for any explicit operation by the developer.

Coding Guidelines

Objects with no declared type must have allocated storage duration and can only be referred to via pointers (this C sentence refers to the effective type of the objects, not the type of the pointers that refer to them). Objects having automatic and static storage duration have a fixed effective type— the one appearing in their declaration. The type of an object having allocated storage duration can change every time a new assignment is made to it.

Allocating storage for an object and treating it as having `type_a` in one part of a program and later on treating it as having `type_b` creates a temporal dependency (the two kinds of usage have to be disjoint) and a spatial dependency (the allocated storage needs to be large enough to be able to represent both types). Keeping track of these dependencies is a cost (developer cognitive resources needed to learn, keep track of, and take them into account) that is often significantly greater than the benefit (smaller, slightly faster-executing program image through not deallocating and reallocating storage). Explicitly deallocating storage when it is not needed and allocating it when it is needed is a minor overhead that creates none of these dependencies between different parts of a program.

Having the same allocated object referred to by pointers of different types creates a union type in all but name:

```
1  #include <stdlib.h>
2
3  float *p_f;
4  int *p_i;
5
6  void f(void)
7  {
8  void *p_v = malloc(sizeof(float)); /* Assume float & int are same size. */
9
10 /* Treat as union. */
11 p_f = p_v;
12 p_i = p_v;
13
14 p_v = malloc(sizeof(float)+sizeof(int));
15
16 /* Treat as struct. */
17 p_f = p_v;
18 p_i = (int *)((char *)p_v + sizeof(float));
19 }
```

Cg 949.1 Once an object having no declared type is given an effective type, it shall not be given another effective type that is incompatible with the one it already has.

Dev 949.1 Any object having no declared type may be accessed through an lvalue having a character type.

---

950 71) This paragraph renders undefined statement expressions such as

footnote 71

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

Commentary

The phrase *statement expressions* is used to make a distinction between the full expression contained within the statement and the syntactic construct `expression-statement`. Expressions can exhibit undefined behavior, but statements cannot (or at least are not defined by the standard to do so).

Other Languages

Even languages that don't contain the ++ operator can exhibit undefined behavior for one of these cases. If a ++ operator is not available, a function may be written by the developer to mimic it (e.g., `a[post_inc(i)] := i`). Many languages do not define the order in which the evaluation of the operands in an assignment takes place, while a few do.

---

951 72) The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first.

footnote 72

Commentary

Every operator is assigned a *precedence* relative to the other operators. When an operand syntactically appears between two operators, it binds to the operator with highest precedence. In C there are thirteen levels of precedence for the binary operators and three levels of precedence for the unary operators.

Requirements on the operands of operators, and their effects, appear in the constraints and semantics subclauses. These occur after the corresponding syntax subclause.

Other Languages

Many other language specification documents use a similar, precedence-based, section ordering. Ada has six levels of precedence, while operators in APL and Smalltalk all have the same precedence (operator/operand binding is decided by associativity).

Example

In the expression `a+b*c` multiply has a higher precedence and the operand `b` is operated on by it rather than the addition operator.

Thus, for example, the expressions allowed as the operands of the binary `+` operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. 952

Commentary

The subsections occur in the standard in precedence order, highest to lowest. For instance, in `a + b*c` the result of the multiplicative operator (discussed in clause 6.5.5) is an operand of the additive operator (discussed in clause 6.5.6). Also the ordering of subclauses within a clause follows the ordering of the nonterminals listed in that syntax clause.

The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.5.1), subscripting brackets `[]` (6.5.2.1), function-call parentheses `()` (6.5.2.2), and the conditional operator `?:` (6.5.15). 953

Commentary

cast-expression syntax
assignment-expression syntax
unary-expression syntax
conditional-expression syntax

A *cast-expression* is a separate subclause because there is a context where this unary operator is not permitted to occur syntactically as the last operator operating on the left operand of an assignment operator (although some implementations support this usage as an extension). In this context a *unary-expression* is required.

The parentheses `()`, subscripting brackets `[]`, and function-call parentheses `()` all provide a method of enclosing an expression within a bracketing construct that cuts it off from the syntactic effects of any surrounding operators. The conditional operator takes three operands, each of which are different syntactic expressions.

Other Languages

Many languages do not consider array subscripting and function-call parentheses as operators.

Within each major subclause, the operators have the same precedence. 954

Commentary

operator 955 associativity

However, the operators may have different associativity.

C++

This observation is true in the C++ Standard, but is not pointed out within that document.

Other Languages

Many language specification documents are similarly ordered.

---

operator associativity

Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein. 955

Commentary

operator 943 precedence

Every binary operator is specified to have an *associativity*, which is either to the left or to the right. In C the assignment operators and the conditional ternary operators associate to the right; all other binary operators associate to the left. Associativity controls how operators at the same precedence level bind to their operands. Operators with left-associativity bind to operands from left-to-right, Operators with right-associativity bind from right-to-left.

Most syntax productions for C operators follow the pattern $X_n \Rightarrow X_n\, op X_{n+1}$ where X_n is the production for the operator, op, having precedence n (i.e., they associated to the left); for instance, `i / j / k` is equivalent to `(i / j) / k` rather than `i / (j / k)`. The pattern for *conditional-expression* (and similarly for *assignment-expression*) is $X_n \Rightarrow X_{n+1} ? X_{n+1} : X_n$ (i.e., it associates to the right); for instance, `a ? b : c ? d : e` is equivalent to `a ? b : (c ? d : e)` rather than `(a ? b : c) ? d : e`.

Other Languages

Most algorithmic languages have similar associativity rules to C. However, operators in APL always associate right-to-left.

Coding Guidelines

Like precedence, possible developer misunderstandings about how operators associate can be solved using parentheses. Expressions, or parenthesized expressions that consist of a sequence of operators with the same precedence, might be thought to be beyond confusion. If the guideline recommendation specifying the use of parentheses is followed, associativity will not be a potential source of faults. However, some of the deviations for that guideline recommendation allow consideration for multiplicative operators to be omitted from the enforcement of the guideline. For the case of adjacent multiplicative operators, this deviation should not be applied.

943.1 expression shall be parenthesized

Cg 955.1 If the result of a multiplicative operator is the immediate operand of another multiplicative operator, then the two operators shall be separated by at least one parenthesis in the source.

If an expression consists solely of operations involving the binary plus operator, it might be thought that the only issue that need be considered, when ordering operands, is their values. However, there is a second issue that needs to be considered— their type. If the operand types are different, the final result can depend on the order in which they were written (which defines the order in which the usual arithmetic conversions are applied).

usual arithmetic conversions

Cg 955.2 If the result of an additive operator is the immediate operand of another additive operator, and the operands have different promoted types, then the two operators shall be separated by at least one parenthesis in the source.

Example

In the following the fact that `j` is added to `i` before `k` is added to the result is not of obvious interest until it is noticed that their types are all different.

```
1  extern float i;
2  extern short j;
3  extern unsigned long k;
4
5  void f(void)
6  {
7  int x, y;
8
9  x = i + j + k;
10
11 y = i / j / k; /* / associates to the left: (i / j) / k */
12
13 i /= j /= k;   /* /= associates to the right: i /= (j /= k) */
14 }
```

Associativity requires that `j` be added to `i`, after being promoted to type **`float`**. The result type of `i+j` (**`float`**) causes `k` to be converted to **`float`** before it is added. The sequence of implicit conversions would

have been different had the operators associated differently, or the use of parentheses created a different operand grouping. Dividing i by j, before dividing the result by k, gives a very different answer than dividing i by the result of dividing j by k.

footnote 73

73) Allocated objects have no declared type. 956

Commentary

The library functions that create such objects (malloc and calloc) are declared to return the type pointer to **void**.

C90

The C90 Standard did not point this fact out.

C++

The C++ operator **new** allocates storage for objects. Its usage also specifies the type of the allocated object. The C library is also included in the C++ Standard, providing access to the malloc and calloc library functions (which do not contain a mechanism for specifying the type of the object created).

Other Languages

Some languages require type information to be part of the allocation request used to create allocated objects. The allocated object is specified to have this type. Other languages provide library functions that return the requested amount of storage, like C.

footnote DR287

DR287) A floating-point status flag is not an object and can be set more than once within an expression. 957

Commentary

Processors invariably set various flags after each arithmetic operation, be it floating-point or integer. For instance, in w*x + y*z after each multiplication flags status flags denoting *result is zero* or *result overflows* may be set. Floating-point status flags differ from integer status flags in that Standard library functions are available for accessing and setting their value, which makes visible the order in which operations take place.

side effect floating-point state

object 941 modified once between sequence points

Implementations that support floating-point state are required to treat changes to it as a side-effect. But, by not treating floating-point status flags as an object, the undefined behavior that occurs when the same object is modified between sequence points does not occur.

This footnote was added by the response to DR #287.

Example

```
1  /*
2   * set/clear or clear/set one of the floating-point exception flags:
3   */
4  (feclearexcept)(FE_OVERFLOW) + (feraiseexcept)(FE_OVERFLOW);
```

If a value is copied into an object having no declared type using **memcpy** or **memmove**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. 958

Commentary

In the declarations of the library functions memcpy and memmove, the pointers used to denote both the object copied to and the object copied from have type pointer to **void**. There is insufficient information available in either of the declared parameter types to deduce an effective type. The only type information available is the effective type of the object that is copied. Another case where the object being copied would not have an

effective type, is when it is storage returned from a call to the `calloc` function which has not yet had a value of known effective type stored into it.

Here the effective type is being treated as a property of the object being copied from. Once set it can be carried around like a value. (From the source code analysis point of view, there is no requirement that this information be represented in an object during program execution.)

Use of character types to copy one object to another object is a common idiom. Some developers write their own object copy functions, or simply use an inline loop (often with the mistaken belief of improved efficiency or reduced complexity). The usage is sufficiently common that the standard needs to take account of it.

Other Languages

Many languages only allow object values to be copied through the use of an assignment statement. Few languages support pointer arithmetic (the mechanism needed to enable objects to be copied a byte at a time). While many language implementations provide a mechanism for calling functions written in C, which provides access to functions such as `memcpy`, they do not usually provide any additional specifications dealing with object types.

In some languages (e.g., awk, Perl) the type of a value is included in the information represented in an object (i.e., whether it is an integer, real, or string). This type information is assigned along with the value when objects are assigned.

Common Implementations

There are a few implementations that perform localized flow analysis, enabling them to make use of effective type information (even in the presence of calls to library functions). While performing full program analysis is possible in theory, for nontrivial programs the amount of storage and processor time required is far in excess of what is usually available to developers. There are also implementations that perform runtime checks based on type information associated with a given storage location.[25]

A few processors tag storage with the kinds of value held in it[34] (e.g., integer or floating-point). These tags usually represent broad classes of types such as pointers, integers, and reals. This functionality might be of use to an implementation that performs runtime checks on executing programs, but is not required by the C Standard.

Example

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  void *obj_copy(void *obj_p, size_t obj_size)
5  {
6  void *new_obj_p = malloc(obj_size);
7
8  /*
9   * It would take some fancy analysis to work out, statically,
10  * the effective type of the object being copied here.
11  */
12 memcpy(new_obj_p, obj_p, obj_size);
13
14 return new_obj_p;
15 }
```

959 For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

effective type lvalue used for access

Commentary

This is the effective type of last resort. The only type available is the one used to access the object. For instance, an object having allocated storage duration that has only had a value stored into it using lvalues of character type will not have an effective type. This wording does not specify that the type used for the access is the effective type for subsequent accesses, as it does in previous sentences.

Coding Guidelines

represen- ?? tation information using

The question that needs to be asked is why the object being accessed does not have an effective type. An access to the storage returned by the `calloc` function before another value is assigned to it, is one situation that can occur because of the way a particular algorithm works. Unless the access is via an lvalue having a character type, use is being made of representation information; this is discussed elsewhere.

---

object value accessed if type

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[74)] 960

Commentary

union member when written to
signed integer corresponding unsigned integer
positive signed integer type subrange of equivalent unsigned type
qualifiers representation and alignment
value copied using unsigned char

This list is sometimes known as the *aliasing rules* for C. Any access to the stored value of an object using a type that is not one of those listed next results in undefined behavior. To access the same object using one of the different types listed requires the use of a pointer type. Reading from a different member of a union type than the one last stored into is unspecified behavior.

The standard defines various cases where types have the same representation and alignment requirements, they all involve either signed/unsigned versions of the same integer type or qualified/unqualified versions of the same type. The intent is to allow objects of these types to interoperate. These cases are reflected in the rules listed in the following C sentences. There are also special access permissions given for the type **`unsigned char`**.

C90

In the C90 Standard the term used in the following types was *derived type*. The term *effective type* is new in the C99 Standard and is used throughout the same list.

Other Languages

Most typed languages do not allow an object to be accessed using a type that is different from its declared type. Accessing the stored value of an object through different types requires the ability to take the addresses of objects or to allocate untyped storage. Only a few languages offer such functionality.

Common Implementations

alignment

The only problem likely to be encountered with most implementations, in accessing the stored value of an object, is if the object being accessed is not suitably aligned for the type used to access it.

Coding Guidelines

represen- ?? tation information using

The guideline recommendation dealing with the use of representation information may be applicable here.

Example

The following is a simple example of the substitutions that these aliasing rules permit:

```
1  extern int glob;
2  extern double f_glob;
3
4  extern void g(int);
5
6  void f(int *p_1, float *p_2)
7  {
8  glob = 1;
9  *p_1 = 3;            /* May store value into object glob. */
10 g_1(glob);           /* Cannot replace the argument, glob, with 1. */
11
```

```
12  glob = *p_1;
13  *p_2 = f_glob * 8.6; /* Undefined behavior if store modifies glob. */
14  g_2(glob);           /* Translator can replace the argument, glob, with 2. */
15  }
```

Things become more complicated if an optimizer attempts to perform statement reordering. Moving the generated machine code that performs floating-point calculations to before the assignment to `glob` is likely to improve performance on pipelined processors. Alias analysis suggests that the objects pointed to by `p_1` and `p_2` must be different and that statement reordering is possible (because it will not affect the result). As the following invocation of `f` shows, this assumption may not be true.

alias analysis

```
1  union {
2         int i;
3         float f;
4        } u_g;
5
6  void h(void)
7  {
8  f(&u_g.i, &u_g.f);
9  }
```

---

961 — a type compatible with the effective type of the object,

object stored value accessed only by

Commentary

Accessing an object using a different, but compatible type(i.e., an enumerated type and its compatible integer type) is thus guaranteed to deliver the same result.

compatible type additional rules

C++

> — *the dynamic type of the object,*

3.10p15

> *the type of the most derived object (1.8) to which the lvalue denoted by an lvalue expression refers. [Example: if a pointer (8.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (clause 10), the dynamic type of the expression `*p` is "`D`." References (8.3.2) are treated similarly.] The dynamic type of an rvalue expression is its static type.*

1.3.3 dynamic type

The difference between an object's dynamic and static type only has meaning in C++.

Use of effective type means that C gives types to some objects that have no type in C++. C++ requires the types to be the same, while C only requires that the types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

compatible type if

Coding Guidelines

The two objects having compatible types might have been declared using one or more typedef names, which may depend on conditional preprocessing directives. Ensuring that such types remain compatible is a software engineering issue that is outside the scope of these coding guidelines.

The issue of making use of enumerated types and the implementation's choice of compatible integer type is discussed elsewhere.

enumeration type compatible with

Example

```
1  extern int f(int);
2
3  void DR_053(void)
4  {
5  int (*fp1)(int) = f;
6  int (**fpp)() = &fp1;
7
8  /*
9   * In the following call the value of fp1 is being accessed by an
10  * lvalue that is different from its declared type, but is compatible
11  * with its effective type: (int (*)()) vs. (int (*)(int)).
12  */
13 (**fpp)(3);
14 }
```

— a qualified version of a type compatible with the effective type of the object, 962

Commentary

qualifiers representation and alignment
pointer converting qualified/unqualified

Qualification does not alter the representation or alignment of a type (or of pointers to it), only the translation-time semantics. Adding qualifiers to the type used to access the value of an object will not alter that value. The **volatile** qualifier only indicates that the value of an object may change in ways unknown to the translator (therefore the quality of generated machine code may be degraded because a translator cannot make use of previous accesses to optimize the current access).

Other Languages

Languages containing a qualifier that performs a function similar to the C **const** qualifier (i.e., a read-only qualifier) usually allow objects having that type to access other objects of the same, but unqualified, type.

Example

```
1  extern int glob;
2
3  void f(const int *p_i)
4  {
5  /*
6   * Only ever read the value pointed to by p_i, but may
7   * directly, or indirectly, cause glob to be modified.
8   */
9  }
10
11 void g(void)
12 {
13 const int max = 33;
14
15 f(&max);
16 f((const int *)&glob);
17 }
```

— a type that is the signed or unsigned type corresponding to the effective type of the object, 963

Commentary

The signed/unsigned versions of the same type are specified as having the same representation and alignment requirements to support this kind of access. The standard places no restriction here on the values represented by the stored value being accessed. The intent of this list is to specify possible aliasing circumstances, not possible behaviors.

footnote 31
971 footnote 74

Other Languages

Few languages support an unsigned type. Those that do support such a type do not require implementations to support the inter-accessing of signed and unsigned types of the form available in C.

Coding Guidelines

The range of nonnegative values of a signed integer type is required to be a subrange of the corresponding unsigned integer type. However, it cannot be assumed that this explicit permission to access an object using either a signed or unsigned version of its effective type means that the behavior is always defined. The guideline recommendation on making use of representation information is applicable here.

If an argument needs to be passed to a function accepting a pointer to the oppositely signed type, an explicit cast will be needed. The issues involved in such casts are discussed elsewhere.

positive signed integer type subrange of equivalent unsigned type
?? representation information using
footnote 31

---

964 — a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

Commentary

This is the combination of the previous two cases.

---

965 — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

Commentary

A particular object may be an element of an array or a member of a structure or union type. Objects having one of these derived types can be accessed as a whole; for instance, using an assignment operator (the array object will need to be a member of a structure or union type). It is this access as a whole that in turn accesses the stored value(s) of the members.

Common Implementations

A great deal of research has been invested in analyzing the pattern of indexes into arrays within loops, with a view to parallelizing the execution of that loop. But, for array objects outside of loops, relatively little research effort has been invested in attempting to track the contents of particular array's elements. There are a few research translators that break structure and union objects down into their constituent members when performing flow analysis. This enables a much finer-grain analysis of the aliasing information.

data dependency
array element held in register

Example

```
1  #include <string.h>
2
3  extern long *p_l;
4
5  union U {
6           int i;
7           long l;
8          } uil;
9  struct S {
10           int i;
11           long l;
12          } sil_1,
13            sil_2;
```

```
14  long a_l[3];
15
16  void f(void)
17  {
18  *p_l = 33;
19
20  /*
21   * The following four statements may all cause the value
22   * pointed to by p_l to be modified.
23   */
24  memset(&uil, 0, sizeof(uil));
25  memset(&sil, 0, sizeof(sil_1));
26  memset(&a_l, 0, sizeof(a_l));
27  sil_2 = sil_1;
28
29  if (*p_l == 33) /* This test is not guaranteed to be true. */
30     a_l[0] = 9;
31  }
```

— a character type. 966

Commentary

Prior to the invention of the **void** type (during the early evolution of C[31]), pointer to character types were used as the generic method of passing values having different kinds of pointer types as arguments to function calls.

Although library functions have always been available for copying any number of bytes from one object to another (e.g., memcpy), many developers have preferred to perform inline copying (writing the loop at the point of copy) or to call their own functions. These preferences show no signs of dying out and the standard needs to continue to support the possibility of objects having character types being aliases for objects of other types.

C++

3.10p15 — *a* ***char*** *or* ***unsigned char*** *type.*

char range, representation and behavior

The C++ Standard does not explicitly specify support for the character type **signed char**. However, it does specify that the type **char** may have the same representation and range of values as **signed char** (or **unsigned char**).

It is common practice to access the subcomponents of an object using a **char** or **unsigned char** type. However, there is code that uses **signed char**, and it would be a brave vendor whose implementation did not assume that objects having type **signed char** were not a legitimate alias for accesses to any object.

Other Languages

While other languages may not condone the accessing of subcomponents of an object, their implementations sometimes provide mechanisms for making such accesses at the byte level.

Coding Guidelines

representation information using ??

value copied using unsigned char

Accessing objects that do not have a character type, using an lvalue expression that has a character type is making use of representation information, which is covered by a guideline recommendation. The special case of the type **unsigned char** is discussed elsewhere.

contracted

A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.75) 967

Commentary

This defines the term *contracted*.

Some processors have instructions that perform more than one operation before delivering a result. The most commonly seen instance of such a multiple operation instruction is the multiply/add pair; taking three operands and delivering the result of evaluating `x + y * z`. This so-called *fused* multiply/add instruction reflects the kinds of operations commonly seen in numerical computations— for instance, matrix multiple and FFT calculations. A fused instruction may execute more quickly than the equivalent two separate instructions and may return a result of greater accuracy (because there are no conversions or rounding performed on the intermediate result).

This wording in the standard explicitly states that the use of such fused instructions is permitted (subject to the use of the `FP_CONTRACT` pragma) by the C Standard, even if it means that the final result of an expression is different from what it would have been had several independent instructions been used.

C90

This explicit permission is new in C99.

C++

The C++ Standard, like C90, is silent on this subject.

Other Languages

Very few languages get involved in the instruction level processor details when specifying the behavior of programs. Fortran does not explicitly mention contraction but some implementations make use of it.

Common Implementations

Some implementations made use of fused multiply/add instructions in their implementation of C90.

Coding Guidelines

An expression that is contracted by an implementation may be thought to deliver the double advantage of faster execution and greater accuracy. However, in some cases the accuracy of the complete calculation may be decreased. The issues associated with contracting an expression are discussed elsewhere.

974 contraction undermine predictability

---

968 The **`FP_CONTRACT`** pragma in **`<math.h>`** provides a way to disallow contracted expressions.

Commentary

The `FP_CONTRACT` pragma also provides a way to allow contracted expressions if they are supported by the implementation.

C90

Support for the `FP_CONTRACT` pragma is new in C99.

C++

Support for the `FP_CONTRACT` pragma is new in C99 and not specified in the C++ Standard.

---

969 Otherwise, whether and how expressions are contracted is implementation-defined.[76)]

contracted how implementation-defined

Commentary

Contraction requires support from both the processor and the translator. (Even although fused instructions may be available on a processor, a translator may not provide the functionality needed to make use of them.) For instance, in `a*b+c*d` there are a number of options open to an implementation that supports a contracted multiply/add. The instruction sequence used to evaluate this expression is likely to be affected by the values from previous operations currently available in registers.

C++

The C++ Standard does not give implementations any permission to contract expressions. This does not mean they cannot contract expressions, but it does mean that there is no special dispensation for potentially returning different results.

Common Implementations

The operator combination multiply/add is the most commonly supported by processors because of the frequency of occurrence of this pair in FFT and matrix operations (these invariably occur in signal processing applications). Other forms of contraction have been proposed for other specialist applications (e.g., cryptography[37]).

The floating-point units in the Intel i860[18] can operate in pipelined or scalar mode, with a variety of options on how the intermediate results are fed into the different units. Depending on the generated code it is possible for the evaluation of `a*b+z` to differ from `c*d+z`, even when the products `a*b` and `c*d` are equal (this issue is discussed in WG14 document N291).

Coding Guidelines

Even in those cases where a developer is aware that expression contraction may occur, there is no guarantee that it will be possible to estimate its impact. For complex expressions the implementation-defined behavior may be sufficiently complex that developers may have difficulty deducing which, if any, subexpression evaluations have been contracted. (One way of finding out the translator's behavior is to examine a listing of the generated machine code.) Once known, what use is this information, on contracted expressions, to a developer? Probably none. The developer needs to look at the issue from a less-detailed perspective.

The only rationale for supporting contracted expressions is improved runtime performance. In those situations where the possible improvement in performance offered by contraction is not required it only introduces uncertainty, a cost for no benefit. Because the default behavior is implementation-defined (no contraction unless requested might have been a better default choice by the C Committee), it is necessary for the developer to ensure that contraction is explicitly switched off, unless it is explicitly required to be on.

Rev 969.1

Unless there is a worthwhile cost/benefit in allowing translators to perform contraction, any source file that evaluates floating-point expressions shall contain the preprocessing directive:

```
#pragma STDC FP_CONTRACT off
```

near the start of the source file, before the translation of any floating-point expressions.

When using the `FP_CONTRACT` pragma developers might choose to minimize the region of source code over which it is in the "ON" state (i.e., having a matching pragma directive that switches it to the "OFF" state) or have it the "ON" state during the translation of an entire translation unit. Until more experience is gained with the use of `FP_CONTRACT` pragma it is not possible to evaluate whether any guideline recommendation is worthwhile.

Forward references: the **`FP_CONTRACT`** pragma (7.12.2), copying functions (7.21.2). 970

---

footnote 74 object aliased

74) The intent of this list is to specify those circumstances in which an object may or may not be aliased. 971

Commentary

An object may be aliased under other circumstances, but the standard does not require an implementation to support any other circumstances. Aliasing is discussed in more detail in the discussion of the **`restrict`** type qualifier.

alias analysis

Other Languages

The potential for aliasing is an issue in the design of most programming languages, although this term may not explicitly appear in the language definition. There is a family of languages having the major design aim of preventing any aliasing from occurring— functional languages.

Coding Guidelines

Although some coding guideline documents warn about the dangers of creating aliases (e.g., developers need to invest effort in locating, remembering, and taking them into account), their cost/benefit in relation to alternative techniques (e.g., moving the declaration of an object from block to file scope rather than passing

its address as an argument in what appears to be a function call) is often difficult to calculate (experience suggest that developers rarely create aliases unless they are required). Given the difficulty of calculating the cost/benefit of various alternative constructs these coding guidelines are silent on the issue of alias creation.

```
1  extern int glob;
2
3  int f(int *valu)
4  {
5  return ++(*valu) + glob; /* Can valu ever refer to glob? */
6  }
```

972 75) A contracted expression might also omit the raising of floating-point exceptions.

footnote 75

Commentary

For instance, an exception might be raised when the evaluation of an expression is not mathematically defined, or when an operand has a NaN value. To obtain the performance improvement implied by fused operations, a processor is likely to minimize the amount of checking it performs on any intermediate results. Also any difference in the value of the intermediate result (caused by different rounding behavior or greater intermediate accuracy) can affect the final result, which might have raised an exception had two independent instructions been used.

947 exception condition
NaN raising an exception

C++

The contraction of expressions is not explicitly discussed in the C++ Standard.

973 76) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators.

footnote 76

Commentary

The difference between machine instructions that combine multiple operators taking floating-point operands and those taking integer operands is that in the former case the final result may be different. While an infinite number of combined processor instructions are possible, only a few combinations occur frequently in commercial applications. Also, while many combinations occur frequently, there are rarely any worthwhile performance advantages to be had from fusing them into a single instruction.

C90

Such instructions were available in processors that existed before the creation of the C90 Standard and there were implementations that made use of them. However, this license was not explicitly specified in the C90 Standard.

C++

The C++ Standard contains no such explicit license.

Other Languages

Although this implementation technique is not explicitly discussed in the Fortran Standard, some of its implementations make use of it.

Common Implementations

Some processors do have special integer instructions for handling graphics processing. However, these are not usually sufficiently general purpose (i.e., they are algorithm-specific) to be used for the combined evaluation of C operators. Combined arithmetic operations on integer data types do not appear in any processor known to your author.

While processors may have fused instructions available, implementations vary in their support for such instructions. The reason for this is that many of the processors providing such instructions are designed with

particular kinds of applications in mind and often have an irregular architecture. Even if fused instructions are generated by a translator, its pattern of register usage may be far from optimal because of the difficulties of mapping sequences of operators in a coherent way.

A study by Arnold and Corporaal[3] looked for frequently occurring sequences of operations in DSP applications. The results found that three operand arithmetic and load/store with prior address calculation occurred reasonably frequently (i.e., may be worth creating a single instruction to perform them).

---

contraction undermine predictability

As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented. 974

Commentary

This sentence applies to implementations. What exactly is a *well-defined* use? For instance, the intent of fused operations is increased performance. How much additional information is needed? Developers need information on the circumstances under which a translator will make use of such instructions and the impact of any surrounding subexpression evaluations (i.e., their potential impact on the generation of exceptions, NaNs, and underflow/overflow). Requiring developers to read listings of generated machine code probably does not count as clearly documented.

Algorithms created by floating-point experts take account of the rounding that will take place during the evaluation of an expression. If the expected rounding does not occur, the results may be less accurate than expected. This difference in rounding behavior need not be restricted to the contracted subexpression; one part of an expression may be contracted and another related part not contracted, leading to an imbalance in the expected values. (The decision on whether to contract can depend on the surrounding context, the same expression being contracted differently in different contexts, undermining predictability.)

Many developers do not have the mathematical sophistication needed to perform an error analysis of an algorithm containing one or more expressions that have been contracted (or not). The behavior of an implementation for particular algorithms is likely to be found, by developers, by measuring how it handles various test cases.

Other Languages

This is an issue that applies to all language implementations that make use of fused instructions to contract expressions.

Coding Guidelines

A clear, fully documented implementation provides no more rationale for allowing translators to contract expressions than a poorly documented one is a rationale for disallowing them. The guideline recommendations on contraction to apply whatever the state of the implementation's documentation.

Example

If the intermediate result of some operator is (with four additional bits of internal accuracy, shown using binary representation, with an underscore after position 53):

```
1.00000000000000000000010111111111111111111111111111111_0111
```

it would be rounded to:

```
1.00000000000000000000010111111111111111111111111111111
```

However, if the original result were not rounded, but immediately used as the operand of some form of fused add instruction with the other operand having the value (in practice the decimal point would be implicitly shifted right):

```
0.0000000000000000000000000000000000000000000000000000_0100
```

the result would be:

```
1.0000000000000000000001100000000000000000000000000000
```

In the nonfused instruction case, where rounding occurred, the result of the add would be the value of the first operand.

References

1. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.
2. A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, 1976.
3. M. Arnold and H. Corporaal. Data transport reduction in move processors. In *Third Annual Conference of the Advance School for Computing and Imaging*, pages 68–75, June 1997.
4. M. C. Baker. *The Atoms of Language*. Basic Books, 2001.
5. M. S. Blaubergs and M. D. S. Braine. Short-term memory limitations on decoding self-embedded sentences. *Journal of Experimental Psychology*, 102(4):745–748, 1974.
6. 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.
7. D. Brooks and M. Martinosi. Value-based clock gating and operation packing: Dynamic strategies for improving processor power and performance. *ACM Transactions on Computer Systems*, 18(2):89–126, May 2000.
8. J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, 1976.
9. J. L. Bruno and T. Lassagne. The generation of optimal code for stack machines. *Journal of the ACM*, 22(3):382–396, 1975.
10. B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.
11. G. Chen. *Effective Instruction Scheduling With Limited Registers*. PhD thesis, Harvard University, Mar. 2001.
12. D. Citron. *Instruction Memoization: Exploiting Previously Performed Calculations to Enhance Performance*. PhD thesis, Hebrew University of Jerusalem, Israel, 2000.
13. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.
14. F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 270–280. IEEE, Dec. 1997.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.
16. J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991.
17. Imsys AB. *the Cjip Technical Reference Manual*. Imsys AB, Sweden, v0.23 edition, 2001.
18. Intel. *i860 64-bit microprocessor programmer's reference manual*. Intel, Inc, 1989.
19. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
20. Intel. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Intel, Inc, Feb. 2001.
21. D. M. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, Aug. 2006.
22. J. King and M. A. Just. Individual differences in syntactic processing: The role of working memory. *Journal of Memory and Language*, 30:580–602, 1991.
23. P. J. Koopman Jr. *Stack Computers the new wave*. Mountain View Press, 1989.
24. M. H. Lipasti. *Value locality and speculative execution*. PhD thesis, Carnegie Mellon University, Apr. 1997.
25. A. Loginov, S. H. Yong, S. Horowitz, and T. Reps. Debugging via run-time type checking. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering 4th International Conference (FASE 2001)*, pages 217–232. Springer-Verlag, Apr. 2001.
26. A. I. Markushevich. *Theory of Functions of a Complex Variable*. American Mathematical Society, second edition, 1998.
27. G. A. Miller and S. Isard. Free recall of self-embedded English sentences. *Information and Control*, 7:292–303, 1964.
28. R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. In *Proceedings 7th International Static Analysis Symposium (SAS 2000)*, volume 1824 of *LNCS*, pages 340–359. Springer, 2000.
29. C. Phillips. *Order and Structure*. PhD thesis, M.I.T., Aug. 1996.
30. PTSC. *IGNITE Intellectual Property Reference Manual*. Patriot Scientific Corporation, San Diego, CA, 1.0 edition, Mar. 2002.
31. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.
32. R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.
33. M. W. Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Thesis (m.s.), M.I.T, Cambridge, MA, USA, May 2000.
34. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.
35. J. H. S. Warren. *Hacker's Delight*. Addison–Wesley, 4th edition, 2003.
36. J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, Mar. 1996.
37. L. Wu, C. Weaver, and T. Austin. CryptoManiac: A fast flexible architecture for secure communication. In *28th Annual International Symposium on Computer Architecture (ISCA-2001)*, pages 110–119, June 2001.