The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.1 Primary expressions

primary-expression syntax

975

```
primary-expression:
                identifier
                constant
                string-literal
                ( expression )
```

Commentary

A primary expression may be thought of as the basic unit from which a value can be read, or into which one can be stored. There is no simpler kind of expression (parentheses are a way of packaging up a complex expression).

C++

The C++ Standard (5.1p1) includes additional syntax that supports functionality not available in C.

Other Languages

Many languages treat a wider range of constructs as being primary expressions (because they do not define array selection, `[]`, structure and union member accesses, and function calls as operators).

Common Implementations

compound expression

`gcc` supports what is calls *compound expressions* as a primary expression.

Semantics

identifier is primary expression if

976

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[77)]

Commentary

An identifier that has not been declared cannot be a *primary-expression*. So references to undeclared identifiers are a violation of syntax (a less through reading of the standard had led many people to believe that such usage was implicitly undefined behavior; the footnote was added in C99 to highlight this point) and must be diagnosed. Whether the identifier denotes a value or refers to an object depends on the operator, if any, of which it is an operand.

lvalue converted to value

C++

The C++ definition of identifier (5.1p7) includes support for functionality not available in C. The C++ Standard uses the term *identifier functions*, not the term *function designator*. It also defines such identifier functions as being lvalues (5.2.2p10) but only if their return type is a reference (a type not available in C).

Other Languages

Some languages implicitly declare identifiers that are not explicitly declared. For instance, Fortran implicitly declares identifiers whose names start with one of the letters `I` through `N` as integers, and all other identifiers as reals.

Common Implementations

Using current, high-volume, commodity technology, accessing an object's storage can be one of the slowest operations. Since 1986 CPU performance has increased by a factor of 1.55 per annum, while DRAM (one of the most common kinds of storage used) performance has only increased at 7% per annum[7] (see Figure **??**). Many modern processors clock at a rate that is orders of magnitude faster than the random access memory chips they are interfaced to. This can result in a processor issuing a load instruction and having to wait 100 or more clock cycles for the value to be available for subsequent instructions to use. The following are a number of techniques are used to reduce this time penalty:

register storage-class

- Translators try to keep the values of frequently used objects in registers.

- Hardware vendors add caches to their processors. In some cases there can be an on-chip cache and an off-chip cache, the former being smaller but faster (and more expensive) than the latter. cache
- Processors can be designed to be capable of executing other instructions, which do not require the value being loaded, while the value is obtained from storage. This can involve either the processor itself deciding which instructions can be executed or its designers can expose the underlying operations and allow translators to generate code that can be executed while a value is loaded. For instance, the MIPS processor has a delay slot immediately after every load instruction; this can be filled with either a NOP instruction or one that performs an operation that does not access the register into which the value is being loaded. It is then up to translator implementors to find the sequence of instructions that minimizes execution time.[8]

Studies have found that a relatively small number of load instructions, so called *delinquent loads*, account for most of the cache misses (and therefore generate the majority of memory stalls). A study by Panait, Sasturkar, and Wong[12] applied various heuristics to the assembler generated from the SPEC benchmarks to locate those 10% of load instructions that accounted for over 90% of all data cache misses. When basic bock profiling was used they were able to locate the 1.3% of loads responsible for 82% of all data cache misses.

Many high-performance processors now support a 64-bit data bus, while many programs continue to use 32-bit scalar types for the majority of operations. This represents a 50% utilization of resources. One optimization is to load (store is less common) two adjacent 32-bit quantities in one 64-bit operation. Opportunities for such optimizations are often seen within loops performing calculations on arrays. One study[1] was able to significantly increase the efficiency of memory accesses and improve performance based on this optimization.

Predicting the value of an object represented using a 32-bit word might be thought to have a 1 in 4×10^9 chance of being correct. However, studies have found that values held in objects can be remarkably predictable.[4,9]

value locality
cache
processor pipeline

Given that high-performance processors contain a cache to hold the value of recently accessed storage locations, the predictability of the value loaded by a particular instruction might not be thought to be of use. However, high-performance processors also pipeline the execution of instructions. The first stages of the pipeline perform instruction decoding and pass the components of the decoded instruction on to later stages, which eventually causes a request for the value at the specified location to be loaded. The proposed (no processors have been built— the existing results are all derived from the behavior of simulations of existing processors modified to use some form of value prediction tables) performance improvement comes from speculatively executing[6] other instructions based on a value looked up (immediately after an instruction is decoded and before it passes through other stages of the pipeline) in some form of *load value locality* table (indexed by the address of the load instruction). If the value eventually returned by the execution of the load instruction is the same as the one looked up, the results of the speculative execution are used; otherwise, the results are thrown away and there is no performance gain. The size of any performance gain depends on the accuracy of the value predictors used and a variety of algorithms have been proposed.[2,11] It has also been proposed that some value prediction decisions be made at translation time.[3]

Coding Guidelines

Some coding guidelines documents require that all identifiers be declared before use. This requirement arises from the C90 specification that an implicit declaration be provided for references to identifiers, which had not been declared, denoting function designators. Such an implicit declaration is not required in C99 and a conforming implementation will issue a diagnostic for all references to undeclared identifiers. This issue is discussed elsewhere.

operator ()

Usage

A study by Yang and Gupta[13] found, for the SPEC95 programs, on average eight different values occupied 48% of all allocated storage locations throughout the execution of the programs. They called this behavior *frequent value locality*. The eight different values varied between programs and contained small values (zero

was often the most frequently occurring value) and very large values (often program-specific addresses of objects and string literals).

Table 976.1: Dynamic percentage of load instructions from different *classes*. The *Class* column is a three-letter acronym: the first letter represents the region of storage (Stack, Heap, or Global), the second denotes the kind of reference (Array, Member, or Scalar), and the third indicates the type of the reference (Pointer or Nonpointer). For instance, *HFP* is a load of pointer-typed member from a heap-allocated object. There are two kinds of loads generated as a result of internal translator housekeeping: *RA* is a load of the return address from a function-call, and any register values saved to memory prior to the call also need to be reloaded when the call returns, *CS* callee-saved registers The figures were obtained by instrumenting the source prior to translation. As such they provide a count of loads that would be made by the abstract machine (apart from *RA* and *CS*). The number of loads performed by the machine code generated by translators is likely to be optimized (evaluation of constructs moved out of loops and register contents reused) and resulting in fewer loads. Whether these optimizations will change the distribution of loads in different classes is not known. Adapted from Burtscher, Diwan and Hauswirth.[3]

Class	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex	bzip	gzip	mcf	Mean
SSN	–	1.28	3.50	0.42	4.40	12.10	6.23	7.26	0.12	0.15	0.15	2.97
SAN	–	0.63	1.01	16.61	–	0.45	2.58	–	12.73	0.01	–	2.84
SMN	–	0.67	–	3.62	–	0.30	–	2.60	–	–	–	0.60
SSP	–	0.37	–	0.17	1.40	–	–	0.33	–	0.02	–	0.19
SAP	–	0.25	–	0.17	–	–	–	–	–	–	–	0.04
SMP	–	0.29	–	0.25	0.01	0.24	2.15	0.05	–	–	–	0.25
HSN	–	0.88	–	14.75	3.51	–	8.07	7.32	0.27	0.01	0.20	2.92
HAN	–	7.39	–	48.55	–	–	4.30	5.39	31.83	–	2.75	8.35
HMN	–	16.37	–	0.76	8.80	6.11	8.42	0.85	–	3.54	27.35	6.02
HSP	–	0.33	–	–	1.82	–	20.01	7.64	–	–	–	2.48
HAP	–	9.42	–	1.33	0.56	–	3.02	4.97	–	–	0.88	1.68
HMP	–	1.82	–	0.11	24.44	0.57	6.29	0.16	–	0.01	17.47	4.24
GSN	43.46	11.10	14.23	0.45	12.76	17.49	16.81	27.79	43.71	43.75	3.12	19.56
GAN	19.27	6.51	52.03	3.00	–	21.86	–	0.03	3.63	26.24	–	11.05
GMN	–	0.81	–	0.41	–	10.96	–	0.16	–	–	2.79	1.26
GSP	–	0.68	–	0.04	–	–	–	–	–	–	0.48	0.10
GAP	–	2.17	–	–	–	0.86	–	0.60	0.41	–	4.72	0.73
GMP	–	0.77	–	0.20	–	0.07	–	–	–	–	0.26	0.11
RA	7.65	5.16	3.68	0.91	8.84	4.58	4.11	4.60	0.76	2.52	7.29	4.17
CS	29.62	33.10	25.55	8.27	33.46	24.40	18.01	30.24	6.54	23.75	32.55	22.12

function leaf/non-leaf

A common program design methodology specifies that all the work should be done in the leaf functions (a *leaf function* is one that doesn't call any other functions). The nonleaf functions simply forms a hierarchy that calls the appropriate functions at the next level. In their study of the characteristics of C and C++ programs (using SPECint92 for C), Calder, Grunwald, and Zorn[5] made this leaf/nonleaf distinction when reporting their findings (see Table 976.2).

instruction profile for different processors

The issue of dynamic instruction characteristics varying between processors and translators is discussed elsewhere. In the case of load instructions, Table 976.3 compares runtime percentages for two different processors.

Table 976.2: Occurrence of load instructions (as a percentage of all instructions executed on HP–was DEC– Alpha). The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* is for calls to nonleaf functions. Adapted from Calder, Grunwald, and Zorn.[5]

Program	Mean	Leaf	NonLeaf	Program	Mean	Leaf	Non-Leaf
burg	21.7	12.9	26.7	eqntott	12.8	11.8	20.2
ditroff	30.3	18.6	32.9	espresso	21.6	20.1	22.9
tex	30.7	19.6	31.3	gcc	23.9	16.7	24.6
xfig	23.5	15.6	25.8	li	28.1	44.1	26.3
xtex	23.2	16.1	28.2	sc	21.2	15.3	22.8
compress	26.4	0.1	26.5	Mean	23.9	17.3	26.2

Table 976.3: Comparison of percentage of load instructions executed on Alpha and MIPS. Adapted from Calder, Grunwald, and Zorn.[5]

Program	MIPS	Alpha	Program	MIPS	Alpha
compress	17.3	26.4	li	21.8	28.1
eqntott	14.6	12.8	sc	19.2	21.2
espresso	17.9	21.6	Program mean	18.2	22.3
gcc	18.7	23.9			

977 A constant is a primary expression.

Commentary

A constant is a single token. A constant expression is a sequence of one or more tokens.

constant expression syntax

Common Implementations

The numeric values of most constants that occur in source code tend to be small. Processor designers make use of this fact by creating instructions that contain a constant value within their encoding. In the case of RISC processors, these instructions are usually limited to loading constant values into a register (the constant zero occurs so often that many of them dedicate a, read-only, register to holding this value). Many CISC processors having instructions to perform arithmetic and logical operations, the constant value being treated as one of the operands. For instance, the Motorola 68000[10] had an optimized add instruction (`ADDQ`, add quick) that included three bits representing values between 1 and 8, in addition to the longer instructions containing 8-, 16-, and 32-bit constant values.

Coding Guidelines

Guidelines often need to distinguish between constants that are visible in the source code and those that are introduced through macro replacement. The reason for this difference in status is caused by how developers interact with source code; they look at the source code prior to translation phase 1, not as it appears after preprocessing. The issues involved in giving symbolic names to constants are discussed elsewhere.

macro replacement

symbolic name

Usage

Usage information on the distribution of all constant values occurring in the source is given elsewhere.

integer constant syntax

978 Its type depends on its form and value, as detailed in 6.4.4.

Commentary

Syntactically all constants have an arithmetic type (the null pointer constant either has the form of an octal constant, or a cast of such a constant, which is not a primary expression).

constant syntax

979 A string literal is a primary expression.

Commentary

The only context in which a string literal can occur in source code is as a primary expression.

Usage

Usage information on string literals is given elsewhere.

string literal syntax

980 It is an lvalue with type as detailed in 6.4.5.

Commentary

It is an lvalue because it has an object type and its type depends on the lexical form of the string literal.

lvalue string literal type

981 A parenthesized expression is a primary expression.

parenthesized expression

Commentary

Parentheses can be thought of as encapsulating the expression within them.

expression evaluation abstract machine base document

Implementations are required to honor the operator/operand pairings of an expression implied by the presence of parentheses. The base document differs from the standard in allowing implementations to rearrange expressions, even in the presence of parentheses.

Common Implementations

An optimizer may want to reorder the evaluation of operands in an expression to improve the performance or size of the generated code. For instance, in:

```
1  extern int i, j, k;
2
3  void f(void)
4  {
5  int x = i + k,
6      y = 21 + i + j + k;
7  /* ... */
8  }
```

it may be possible to improve the generated machine code by rewriting the subexpression `21 + i + j + k` as `i + k + j + 21`. Perhaps the result of the evaluation of `i + k` is available in a register, or is more quickly obtained via the object `x`.

However, such expression rewriting by a translator may not preserve the intended behavior of the expression evaluation. The expression may have been intentionally written this way because the developer knew that the evaluation order specified in the standard would guarantee that the intermediate and final results were always representable (`i + k` may be a large negative value, with `j` sometimes having a value that would cause this sum to overflow).

In the above case, rewriting as `((21 + i) + j) + k` would stop most optimizers from performing any reordering of the evaluation. However, if an optimizer can deduce that reordering the evaluation through parentheses would not affect the final result, it can invoke the as-if rule (on processors where signed integer arithmetic wraps and does not signal an overflow, reordering the evaluation of the expression would not cause a change of behavior). The only visible change in external behavior might be a change in program performance, or a smaller program image.

as-if rule

For floating-point types wrapping behavior cannot come to an optimizer's rescue (by enabling overflows to be ignored). However, some implementations may chose to consider overflow as a rare case, preferring the performance advantages in the common cases. Overflow is not the only issue that needs to be considered when operands have a floating-point type, as the following example shows:

```
1  extern double i, j, k;
2
3  void f(void)
4  {
5  double x = i + k,
6         y = i + j + k;
7  /* ... */
8  }
```

assuming the objects have the following values:

```
i = 1.0E20
j = -1.0E20
k = 6.0
```

then the value that is expected to be assigned to `y` is `6.0`. Rewriting the expression as `i + k + j` would result in the value `0.0` being assigned (because of limited accuracy, adding `6.0` to `1.0E20` gives the result `1.0E20`).

Coding Guidelines

?? expression shall be parenthesized

There is a guideline recommendation specifying that expressions shall be parenthesized.

Use of parentheses makes the intentions of the developer clear to all. Objections raised, by developers, against the use of unnecessary parentheses are often loud and numerous. Many developers consider that use of parentheses needs to be justified on a case-by-case basis. Others take the view that there are no excuses for not knowing the precedence of all C operators and that all developers should learn them by heart. These coding guidelines accept that many developers do not know the precedence of all C operators and that exhortations to learn them will have little practical effect. Parenthesizing all binary (and some unary) operators and their associated operands is considered to be the solution (to the problem of developers' incorrect deduction of the grouping of operands within an expression). In some instances a case can be made for not using parentheses, which are discussed in the Syntax sections of the relevant operators.

operator precedence

As the discussion in Common Implementations showed, the use of parentheses can sometimes reduce the opportunities available to an optimizer to generate more efficient machine code. These coding guidelines consider this to be a minor consideration in the vast majority of cases, and it is not given any weight in the formulation of any guideline recommendations.

Some coding guideline documents recommend against redundant parentheses; for instance, in `((x))` the second set of parentheses serves no purpose. Occurrences of redundant parentheses are rare and there is no evidence that they have any significant impact on source code comprehension. These coding guidelines make no such recommendation.

Usage

Usage information on parentheses usage is given elsewhere.

parenthesized expression nesting levels

---

982 Its type and value are identical to those of the unparenthesized expression.

Commentary

But, for the use of expression rewriting by an optimizer, the generated machine code will also be identical.

Common Implementations

Some early implementations considered that parentheses 'hid' their contents from subsequent operators. This created a difference in behavior between `sizeof("0123456")` and `sizeof(("0123456"))`— one returning the **`sizeof`** of an array operand, the other the size of a pointer operand. The C Standard makes no such distinction.

array converted to pointer

---

983 It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

Commentary

This means that `(a) = (f)(x)` and `a = f(x)` are equivalent constructs.

Other Languages

In many languages a function call is a primary expression, so it is not possible to parenthesize a function designator.

---

984 **Forward references:** declarations (6.7).

References

1. M. J. Alexander, M. W. Bailey, B. R. Childers, J. W. Davidson, and S. Jinturkar. Memory bandwidth optimizations for wide-bus machines. Technical Report CS-92-24, Department of Computer Science, University of Virginia, Aug. 4 1992.
2. M. Burtscher. *Improving Context-Based Load Value Prediction*. PhD thesis, University of Colorado, 2000.
3. M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, June 2002.
4. B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the* 30^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.
5. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
6. F. Gabbay. Speculative execution based on value prediction. Technical Report EE Department technical Report #1080, Technion - Israel Institute of Technology, Nov. 1996.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.
8. S. M. Kurlander, T. A. Proebsting, and C. N. Fischer. Efficient instruction scheduling for delayed-load architectures. *ACM Transactions on Programming Languages and Systems*, 17(5):740–776, 1995.
9. M. H. Lipasti. *Value locality and speculative execution*. PhD thesis, Carnegie Mellon University, Apr. 1997.
10. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
11. T. Nakra. *A framework for performing prediction in VLIW architectures*. PhD thesis, University of Pittsburgh, 2001.
12. V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 303–314, Mar. 2004.
13. J. Yang and R. Gupta. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems*, 2(3):1–27, 2002.