Communications of the Association for Information Systems

Volume 13 Article 34

March 2004

The Subtle and Counterintuitive Hazards of Non-Sequential Evaluation in Languages of the C Family

Adam Victor Reed
California State University. Los Angeles, areed2@calstatela.edu

Knox B. Wasley
California State University. Los Angeles, kwasley@calstatela.edu

Communications of the Association for Information Systems

THE SUBTLE AND COUNTERINTUITIVE HAZARDS OF NON-SEQUENTIAL EVALUATION IN LANGUAGES OF THE C FAMILY

Adam Reed
Knox B. Wasley
Department of Computer Information Systems
California State University, Los Angeles.
areed2@calstatela.edu

ABSTRACT

In the family of programming languages derived from the C programming language, the order of evaluation of expressions between sequence points is left undefined and may be counterintuitive. While most programmers assume that this lack of a defined order refers only to a lack of preference between left-to-right and right-to-left sequential evaluation, the standards actually permit, and language processors implement, a variety of non-sequential evaluation orders. In some cases, particularly those where expressions containing sequence points are combined without an intervening sequence point, computed results can be unanticipated, and can vary with different compilers for the same code. Yet these results fail to trigger any warning from compilers or syntax-checkers. Because languages of the C family dominate Information Systems curricula and new business application code, the risk that code in those languages may give incorrect results without any warning from automated tools presents significant hazards against the integrity of financial calculations. This article explains those hazards and outlines countermeasures.

KEYWORDS: ANSI C, C, C++, Perl, Java, programming languages, programming standards, software engineering, order of evaluation, sequence points, programming education, language processors, compilers, interpreters, virtual machines.

I. INTRODUCTION

During the 2001-2002 academic year, IS students at California State University, Los Angeles, took 3568 units of coursework in programming languages of the C family (C++, Perl and Java) and 1052 units of coursework in the only programming language we teach outside the C family, Visual Basic. With 77% of all programming coursework in languages whose syntax and semantics are derived from C, most of our students did not obtain experience in any programming language outside the C family by the time they graduated, and wrote all or nearly all of subsequent business application code in languages of the C family.

In that year, in one of our examinations in a C++ course several students wrote code that appeared to be correct and generated no warning from any compiler or syntax-checking tool available on our campus, yet gave apparently incorrect results when compiled and run. Clearly, programmers face the risk that similar, apparently correct code, may threaten the integrity of IS systems in ways that are difficult to detect and fix. In this article we share our observations on the subtle and counterintuitive hazards of non-sequential evaluation.

The C language was originally developed in 1971, on a PDP-11 computer with 24 K bytes of core memory [Ritchie 1984, pp 1588, 1592]. That is approximately one-millionth the typical memory size (24 GB) of a comparably priced server computer ($ 65,000., Ritchie 1984 p. 1587) in 2004. C was the first computer language to combine a concise and readable free-format syntax with the flexibility and expressive precision of assembly language. In 2004, C remains the language of choice, among programmers, for the development of operating systems, device drivers, and other software that still demands the combination of expressive precision with extreme computational efficiency.

The minimal storage requirements and extreme computational efficiency of the C language were achieved by relaxing some of the intuitive constraints implicit in how programmers read, write, and think about programs. Programmers read and write code as though programmatic expressions were evaluated in some well-defined order, such as left to right or right to left. This assumption, the assumption of sequentiality, was mostly discarded in the design of compilers and interpreters for C and its family of languages. ANSI C contains only three sequentiality requirements, two in the definition of pre-increment/decrement and post-increment/decrement operators, plus the requirement that all operations before a sequence point precede operations after a sequence point [ISO 1999, p. 13]. The order of operations between sequence points is otherwise left undefined. Precedence and grouping, even explicit grouping with parentheses, do not actually determine the order in which computations may be carried out [ISO 1999, p. 16]. The replacement of strict sequentiality implicit in human cognitive models of program operation, with discrete sequence points at fixed places in the code and possibly non-sequential evaluation in-between, is the source of the subtle and counterintuitive problems discussed in this paper.

Regardless of the order in which a programmer thinks about the operation of a program, a C language processor is just as likely to evaluate a statement with more than two expressions between sequence points in some asymmetric sequence, such as middle-right-left or right-left-middle, as in an intuitively expected order such as right to left or left to right. Even when expressions are organized in sub-expressions, the language processor is free to evaluate some parts of one sub-expression, then work on parts of other sub-expressions, and eventually come back, often repeatedly jumping from place to place in the code. Moreover, our experience shows that putting precautionary sequence points into the code is not a real solution. If a sub-expression containing an internal sequence point is used together with other sub-expressions, without an external sequence point between them, language processors may, and do, jump out of and into different sub-expressions (even sub-expressions with internal sequence points) in any order that may be convenient.

II. HISTORY

The first book on C by Kernighan and Ritchie [1978,] included a two-page section on "Precedence and Order of Evaluation" (pp. 48-50). The precedence rules discussed in this section were well understood by most programmers; many kept page 49, with its table of precedence and associativity of operators, permanently bookmarked in their copy of "K&R". The material on order of evaluation, on the other hand, was seldom understood in all of its implications. K&R wrote that "C, like most languages, does not specify in what order the operands of an operator are evaluated" - but "intermediate results can be stored in temporary variables to assure a particular sequence." "In any expression involving side effects," that is where some variable is changed as a by-product of evaluating an expression, "there can be subtle dependencies on the order in which the variables taking part in the expression are stored."

When assignments take place it “is left to the discretion of the compiler, since the best order strongly depends on machine architecture.”

While K&R's discussion can be seen, after consideration of findings presented later in this article, as a warning about C's general lack of a defined order of evaluation, it was actually read, by most programmers, as indicating a lack of preference between the two sequential orders of evaluation, left to right and right to left. The first author of this article (Adam Reed) participated in several dozen code reviews at Bell Labs, AT&T, and Lucent Technologies, at which the possibility of non-sequential evaluation was never considered by anyone, including programmers who learned C directly from its inventors. When questions about order of evaluation arose during code reviews, the participants considered only evaluation in the two intuitive orders, sequentially left to right and sequentially right to left; if the result was the same, the code was approved.

The availability of code verifiers, such as lint, also contributes to general ignorance about the lack of sequentiality constraints in the order of evaluation of C. The qualifier "most" (in K&R's note that "lint will detect most dependencies on order of evaluation") was often overlooked. After lint became a required step in the code development process, code that linted cleanly was seldom examined for defects that lint was thought to warn about. And if it was examined, code that evaluated to the same result in the two sequential orders was assumed to be correct. Code verification eventually became a standard, on-by-default feature of production compilers and other language processors. Programmers educated in the last two decades assume that they will be warned about all errors related to the syntax, and the relation between syntax and semantics, of the languages they are working with. As we will see, such assumptions with respect to languages of the C family can lead to defective programs and corrupt results.

This conflict, between intuitive and literal interpretations of the lack of specified order of evaluation in C, is most likely to surface when multiple assignments are made to the same variable in the course of computing an expression. The intuitive interpretation is that the assigned value is the value that will be used in the place where the evaluation occurs. That is, when the value of "(v=something)" is used, its value is the value returned by the "something". However, the language processor is free to do anything else, including actions that change the value of the just-evaluated variable, between assigning a value to a variable and using the variable in the expression. Therefore, by the time v is used, its value may have changed to something else.

This last point was first recognized by Bolsky [1985]. In a half-page section on "operand evaluation order" (p. 19), Bolsky writes:

> *"Caution - When you make an assignment to a variable in any kind of expression (including function calls), do not use that variable again in the same expression."*

This statement is perhaps , in retrospect, the first published warning about the counterintuitive consequences of the possibility of non-sequential evaluation in C. We did not find any other warnings about this possibility until the first ANSI C standard ISO/IEC 9899-1990 [ISO 1990].

Other counterintuitive aspects of the syntax and semantics of C, and of languages derived from C, are well known among programmers. The short-circuiting of Boolean expressions, for example, is counterintuitive to anyone used to thinking of Boolean expressions in the context of symbolic logic rather than the programmatic specification of computational procedures. For this reason, instructors usually make sure that this aspect of C and C-like languages is well understood by students. C standards [ISO, 1990, ISO, 1999] explicitly specify the short-circuiting of Boolean expressions. Some programmers consider C to be, in this respect, better than some other languages, which leave the choice to short-circuit (or not to) to compiler implementers. The Java Language Specification [Gosling et al 2000] provides alternative operators (“&” and “|” instead of “&&” and “||”) without short-circuiting, and in Java those operators are often taught as the default.

Beginning with *C Traps and Pitfalls* [Koenig 1988], there has been a persistent market for books on counterintuitive aspects of languages of the C family. Some issues, including the shortcicuiting of Boolean expressions, are covered well in these books. But none of those books mention pitfalls associated with C's undefined order of evaluation. Even the most recent (*C++ Gotchas: Avoiding Common Problems in Coding and Design*, [Dewhurst 2003]) conflates order of evaluation with precedence, though the ANSI C standard explicitly warns against assuming that precedence and grouping determine the order of evaluation (e.g. [ISO1999] p. 16.)

III. COUNTERINUTIVE ORDERS OF EVALUATION

Programmers, and automated syntax checking components of language processors, tend to assume that the ambiguity in the order of evaluation is adequately removed by inserting a sequence point after each assignment or other side-effect. Consider, for example, the following expression, written to compute the desired result and communicate the programmer’s intention in a situation that is frequently encountered in real financial applications:

```
( (actual(&returnOnInvestment),returnOnInvestment)
  - (expected(&returnOnInvestment),returnOnInvestment) )
```

or more generally,

```
( (f1(&v),v) - (f2(&v),v) )
```

which is a special case of

```
(a,b) - (c,d).
```

Intuitively one of two orders of actual evaluation is expected: [a,b,c,d] if expressions are evaluated left-to-right, or [c,d,a,b] if they are evaluated right-to-left.

However, the only constraints imposed by inserting sequence points are that a be evaluated before b, and c before d. These constraints are satisfied by any of the following orders:

(1)	[a,b,c,d]
(2)	[c,d,a,b]
(3)	[a,c,b,d]
(4)	[a,c,d,b]
(5)	[c,a,b,d]
(6)	[c,a,d,b]

Abbreviating (f1(&v),v) as v1 and (f2(&v),v) as v2, we see that when (f1(&v),v) - (f2(&v),v) is evaluated, the first two orders produce the intuitively expected result, (v1-v2). However, orders (3) and (4) give the result (v2-v2), while orders (5) and (6) give (v1-v1). If real language processors use the latter orders rather than the former, the results of intuitively coded computations will not be what the programmer expects.

Instead of the case of ((f1(&v),v) - (f2(&v),v)), frequently used in actual business applications, in the following example we used the case of ((f1(&v),v) + (f2(&v),v)), which more clearly illustrates the variety of results from identical or equivalent code in common C language processors.

IV. RESULTS WITH COMMON LANGUAGE PROCESSORS

Table 1 shows the variety of results obtained when the code for a simple example, ((v=2,v) + (v=1,v)), is processed with 12 common language processors for different languages of the C family:

Table 1. Results

Language and Code		**Language Processor**	**PLATFORM**	**(2+1)**

C	main(){
	    int v, sum;
	    sum = (v=2,v) + (v=1,v);
	    printf("The value of “,
	        “(2 + 1) is %i\n",sum);
	}
		GNU gcc, version 3.0.3	Solaris 8 UltraSparc	4

		Sun WorkShop 6 cc (update 2 C 5.3 2001/05/15)	Solaris 8 UltraSparc	3
		Microsoft Visual C++ 6.0 (Service Pack 5)	Windows 2000 Pentium 3	2
		bcc32.exe Borland C++ 5.5	Windows 2000 Pentium 3	3

C++	#include <iostream.h>
	int main() {
	    int v, sum;
	    sum = (v=2,v) + (v=1,v);
	    cout<<"The value of (2 + 1) is "
	        <<sum<<endl;
	    return 0;
	    }
		GNU g++, version 3.0.3	Solaris 8 UltraSparc	4

		Sun WorkShop 6 CC (update 2 C 5.3 2001/05/15)	Solaris 8 UltraSparc	2
		Microsoft Visual C++ 6.0 (Service Pack 5)	Windows 2000 Pentium 3	2
		bcc32.exe Borland C++ 5.5	Windows 2000 Pentium 3	3

Perl	#! /usr/bin/perl
	    $sum = ($v=2) + ($v=1);
	    print "The value of (2 + 1) is “;
	    print “$sum\n”;
		perl, version 5.005_03	Solaris 8 UltraSparc	2

Java	import java.util.*;
	public class Bug1 {
	    public static void
	        main(String[]args) {
	        int v, sum;
	        sum = (v=2) + (v=1);
	        System.out.println("The value”
	            + “ of (2 + 1) is " + sum);
	        }
	    }
		GNU gcj, version 3.0.3	Solaris 8 UltraSparc	3

		Sun Java 1.4.0 javac (build 1.4.0-b92)	Solaris 8 UltraSparc	3
		Microsoft Visual J++ 6.0 (Service Pack 5)	Windows 2000 Pentium 3	3

The only language in which the intuitive order of evaluation was followed by all current language processors was Java. The Java Language Specification [Gosling et al, 2000] guarantees that Java code will be “evaluated in a specific evaluation order, namely, from left to right.” This specification eliminates the need for reliance on sequence points. The Java compilers will not compile code that contains otherwise redundant (sequence-point only) comma operators. Moreover, current Java compilers produce code for the Java virtual machine, rather than the actual target hardware platform on which they happen to execute.

V. COUNTERMEASURES

Computational errors due to non-sequential evaluation in common C language processors are a threat to the integrity of information systems. To safeguard the system integrity, the following steps need to be taken:

1. examine all code in affected languages that is in current use,

2. replace references to variables that are modified more than once in the same statement with references to separate variables.

For example, replace

```
 ( (actual(&returnOnInvestment),returnOnInvestment) -
 (expected(&returnOnInvestment),returnOnInvestment) )
```

with

```
    ( (actual(&actualReturnOnInvestment),actualReturnOnInvestment) –
expected(&expectedReturnOnInvestment),expectedReturnOnInvestment) )
```

Tools that can flag statements in which the same variable is modified more than once can be written by most software vendors. They should be demanded from language processor vendors and deployed without delay when they become available. Language processors should, at minimum, provide a warning in such cases.

Using additional variables, as illustrated above, is not the only possible approach to this requirement, but other approaches have significant disadvantages. Thus, one might be seduced by the relative simplicity of programming guidelines that totally forbid sub-expressions with side effects. Such guidelines, however, are likely to compromise the programmer’s ability to communicate intentions and logics within the code, to its future maintainers and re-users. The use of additional, explicitly differentiated variables adds some typing but does not compromise clarity of expression.We offer four recommendations:

1. The potential conflict between programming guidelines, which are advocated by some leading academic researchers, and the programmer’s ability to communicate with future maintainers and re-users, is a particularly important consideration for the praxis of organizational IS. In organizations that are not dedicated to creating new software, and even in some that are, the most important factor, in individual and organizational programming productivity, is to eliminate the waste of time and effort that would be unavoidable with writing software from scratch. When software is needed, individual and organizational productivity is maximized by starting with existing software components, and using the least additional programming effort to add the newly specified functionality. At the same time, future maintainability, extensibility and reusability require that the software remain maximally readable and understandable after the changes are performed. But to make the augmented software readable and comprehensible, modified and added code must fit into the pre-existing patterns, conventions, idioms, and techniques of the original code. If new programming guidelines are applied to added and modified code only, then the resulting code when read as a whole (in the context of future maintenance, augmentation or re-use) will be incoherent and incomprehensible. If the whole re-used component were re-written to comply with the newly imposed guidelines, then most of the potential productivity gains from the strategy of software re-use will be lost. For this reason we recommend the use of explicitly differentiated variables, rather than the imposition of coding guidelines.

2. IS managers ought to make sure that every programmer who writes code in the affected languages is aware of the subtle and counterintuitive hazards implicit in the current standards and implementations of C, C++ and Perl. In particular, the fact that the use of precedence and

sequence points is not sufficient to guarantee a desired order of evaluation needs to be understood, and applied whenever code in those languages is being written or reviewed.

3. IS managers should persuade language processor vendors to update their products to guarantee sequential evaluation in C, C++ and Perl. Some language processors for C and C++ may already implement sequential evaluation in practice (Table 1) but all need a systematic re-check of processor internals before such a guarantee could be provided by their vendors. Other language processors may have to be re-designed or re-implemented before such guarantees could be made. In the meantime, all production code in C and C++ should be re-tested by re-compilation with multiple language processors.

4. IS managers should consider writing new application code in Java, rather than C or C++. The Java Language Specification (Gosling et al, 2000) guarantees that Java code will be "evaluated in a specific evaluation order, namely, from left to right." While Gosling et al [1998] recommend "that code not rely crucially on that specification," their recommendation is based on a perceived need to write code with a view to the possibility of code being retro-ported back to C++, rather than on any limitation in the applicability of the guarantee of sequential evaluation in Java. As long as the bytecode sequence produced by the compiler conforms to the language specification, a uniform order of evaluation across all platforms is further assured by execution in a functionally identical Java Virtual Machine on any platform. Although execution in the Java Virtual Machine is significantly slower than the execution of code compiled for each native processor, the rapid growth in processor speed and in the availability of large multi-processor platforms makes for new tradeoffs in business applications, in which the integrity of financial computations is of paramount importance. Java compilers that produce platform-specific binary code for maximum efficiency will become usable in business computing only when they are guaranteed to conform to the Java Language Specification. We do not recommend jumping into the more innovative parts of Java, such as threads, because the IS discipline does not, at this time, have enough experience on the interaction of programmer intuitions with non-deterministic aspects of threading. We do recommend the use of Java to implement non-threaded code that might otherwise be implemented in C or C++.

VI. INSTRUCTIONAL IMPLICATIONS

Computer languages that compile into assembler and machine code, such as C and C++, rather than virtual machine or interpreter code, such as Java or Perl, remain indispensable for high machine cycle efficiency, real-time performance, and communication with hardware registers. Thus C and C++ will remain, for the foreseeable future, the languages of choice for the implementation of operating system components, libraries, device drivers, language processors and virtual machines, as well as for computer games, hardware control, and real-time application software. All of the above software is produced primarily by researchers and software vendors; it is seldom written in typical business IS environments. The demand for C and C++ programmers will continue to be met by graduates of Computer Science departments. In business, and especially in financial calculations, we believe that, because the Java Language specification guarantees sequential evaluation on all platforms, fluency in Java (and in not necessarily in any other programming language) is an overwhelming advantage. It is now the indispensable qualification for a career in business information systems.

The counterintuitive hazards implicit in the C and C++ standards are a good reason to switch to Java as the primary instructional computer language. Introductory programming instruction inculcates habits that are meant to last a lifetime. One of those essential habits of effective programmers and programming managers is always to be aware of the operations implicit in code. But when the language standard permits these operations to depend on idiosyncrasies of processors and environments, this is not possible. With C and C++, habits that ought to make programmers effective become counterproductive to the goal of writing reliable code.

At our institution, Cal State LA, after the fall quarter of 2003 the introductory programming course required in the undergraduate IS curricula is no longer taught in C or C++. Students who would have taken the introductory programming course in those languages are now advised to take it in Java. Our courses in Perl and in Visual Basic will not change, since in some environments, such as Microsoft Windows scripting in the case of Visual Basic, or CGI scripting in the case of Perl, those languages are likely to remain dominant for the foreseeable future. Elective courses in C++ and COBOL will remain available.

In our courses in C++ and Perl, students are required to learn about the possibility that expressions, even when equipped with sequence points, may be evaluated by some language processors in counterintuitive orders. This problem will be pointed out, when C++ or Perl are first introduced, as a salient difference between those languages, and the Java language with which the student is already familiar. It is of course our hope that actual language processors, for C, C++, and Perl will be modified to guarantee sequential evaluation. Until this change occurs, the counterintuitive hazards documented in this article must be attended to.

Editor's Note: This article was received on October 23, 2003. It was with the authors for approximately three and a half months for two revisions. It was published on June 8, 2004.

REFERENCES

Bolsky, M. I. (1985). *The C Programmer's Handbook.* Englewood Cliffs, N.J.: Prentice-Hall,

Dewhurst, S. C. (2003). *C++ Gotchas: Avoiding Common Problems in Coding and Design.* Boston, MA: Addison-Wesley

Gosling, J., Joy, W., Steele, G., and Bracha, G. (2000). *The Java Language Specification, Second Edition.* Cupertino, CA: Sun Microsystems.

ISO (1990) International Organization for Standardization and International Electrotechnical Commission *International Standard ISO/IEC 9899:1990 (E) - Programming Languages - C.* Geneva, Switzerland: International Organization for Standardization.

ISO (1999)International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 9899:1999 (E) - Programming Languages - C, Second Edition.* Geneva, Switzerland: International Organization for Standardization.

Kernighan, B. W., and D. M. Ritchie (1978). *The C Programming Language.* Englewood Cliffs, N.J.: Prentice-Hall

Koenig, A. (1988). *C Traps and Pitfalls.* Reading, MA: Addison-Wesley.

Ritchie, D. M. (1984). The Evolution of the UNIX Time-sharing System. *AT&T Bell Laboratories Technical Journal*, (63) 8.2 , pp. 1577-1593. October

Rosen K. H., R. R. Rosinski, J. S. Farber and D. A. Host, eds. (1996), UNIX *System V Release 4, An Introduction, Second Edition*. Berkeley, CA: Osborne McGraw-Hill.

ABOUT THE AUTHORS

Adam Reed is Associate Professor of Information Systems at California State University, Los Angeles. His current research interests include productivity and accessibility of information systems, epistemology of knowledge representation, programming languages, operating systems, network protocols, and network security administration. His research publications have appeared in *Science, Memory and Cognition, Contemporary Psychology, Journal of Verbal*

Learning and Verbal Behavior, and in *Behavior Research Methods and Instrumentation.* He is the author of three chapters on Unix in Rosen et.al [1996], He holds 3 US Patents.

Knox Wasley is Associate Professor of Computer Information Systems at California State University, Los Angeles. He received an MBA (Accounting) and and MS (Information Science) and earned CPA, CISA and CDP certifications. His teaching interests include Programming, Analysis and Design, I.T. Auditing, and Entrepreneurship. His research interests include how to include the topic of Application Systems in CIS curricula and studies that measure and lead to improvements in CIS majors' impacts in the workplace. He held positions at Affiliated BancShares of Colorado, Touche Ross & Co., Union Bank, and First Interstate Bank.

Communications of the Association for Information Systems

ISSN: 1529-3181