WHAT'S NEW IN UNDEFINED BEHAVIOR?

Robert C. Seacord

woven BY TOYOTA

PRESENTER BIO: ROBERT C. SEACORD

Senior Staff Software Engineer at Woven by Toyota

Convenor of the C Standards Committee

Author of Effective C, Secure Coding in C and C++, and the CERT C Coding Standard

IEEE Computer Society Cybersecurity Award for Practice Recipient

WHAT'S NEW IN C?

BEHAVIORS [ISO/IEC 9899]

implementation-defined behavior - Unspecified behavior whereby each implementation documents how the choice is made.

unspecified behavior - Behavior for which the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

undefined behavior - Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements. An example of undefined behavior is the behavior on integer overflow.

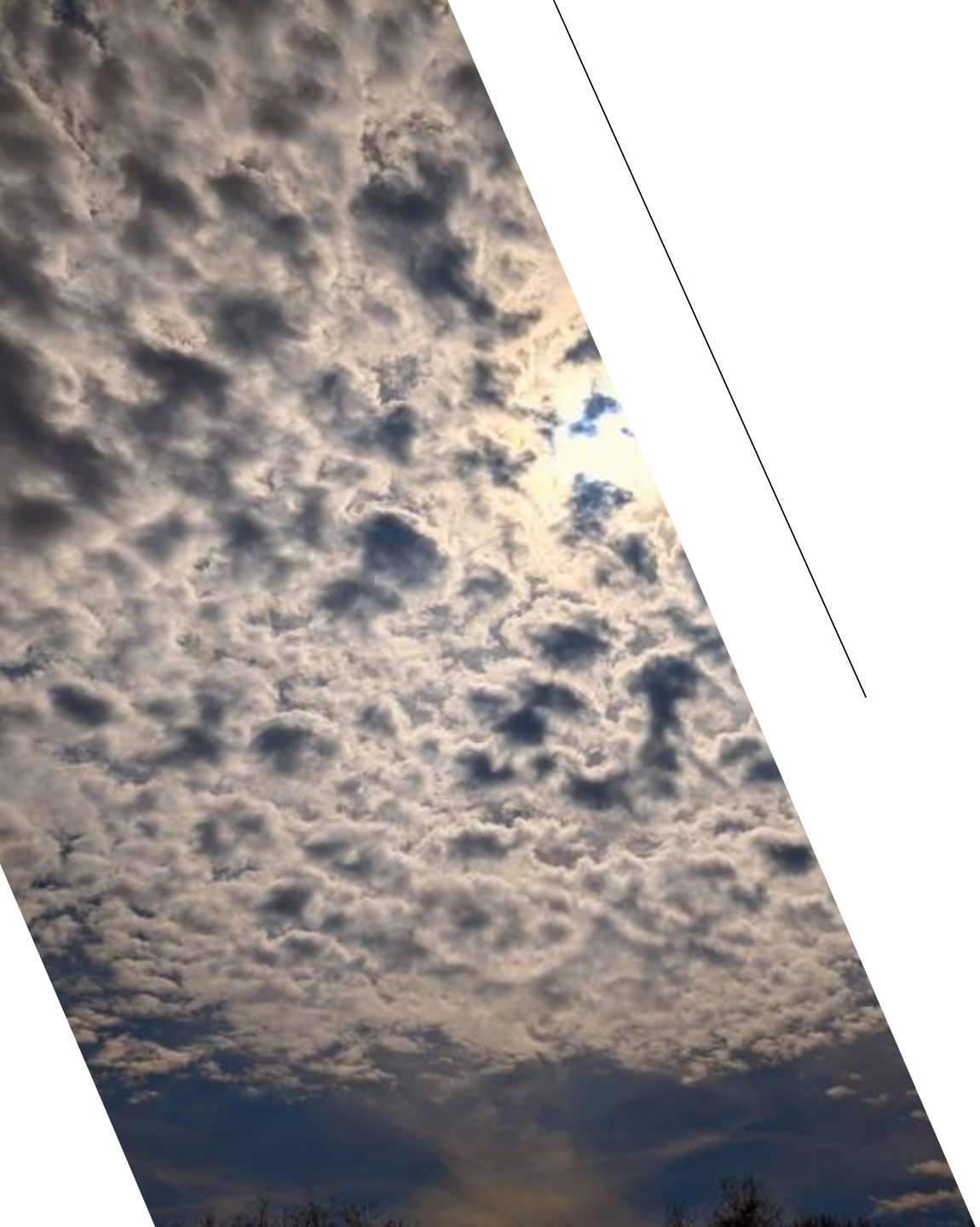
IDENTIFYING UNDEFINED BEHAVIORS

Undefined behaviors are identified in the standard:

- **If a “shall” or “shall not” requirement is violated, and that requirement appears outside of a constraint, the behavior is undefined.**
- **Undefined behavior is otherwise indicated in this International Standard by the words “undefined behavior”**
- **by the omission of any explicit definition of behavior.**

There is no difference in emphasis among these three; they all describe “behavior that is undefined”.

The C Standard Annex J.2, “Undefined behavior,” contains a list of explicit undefined behaviors in C.



UNDEFINED BEHAVIOR CLASSIFICATIONS

Behaviors are classified as undefined by the standards committees to:

- give the implementer license not to catch certain program errors that are difficult to diagnose;
- avoid defining obscure corner cases which would favor one implementation strategy over another;
- identify areas of possible conforming language extension: the implementer may augment the language by providing a definition of the officially undefined behavior.

OBSCURE CORNER CASES

To build a portable system, the language standard could impose uniform behavior over erroneous or non-portable constructs, as many higher-level languages do.

Hardware Platform Divergence

`1 << 32`

- ARM: yields 0
- x86 / PowerPC yields 1

`1 << 64`

- ARM / PowerPC yields 0
- x86: yields 1

Mandating a single defined behavior would require compilers to synthesize extra instructions whenever hardware semantics differ.

```
if ((y < 0) || (y ≥ n)) then 0 else x << y // x86
```

UNDEFINED BEHAVIORS

Implementations may

- ignore undefined behavior completely with unpredictable results
- behave in a documented manner characteristic of the environment (with or without issuing a diagnostic)
- terminate a translation or execution (with issuing a diagnostic).

C and C++ trusts (e.g., shifts responsibility to) programmers to avoid undefined behaviors.

CONSTRAINTS

- A constraint is a restriction, either syntactic or semantic, by which the exposition of language elements is interpreted.
- A conforming implementation must produce at least one diagnostic message if a translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined.
- Diagnostic messages are not required to be produced in other circumstances.
- A shall requirement within a constraint section is a constraint violation and not undefined behavior.

ERROR MESSAGE TYPES

Command line errors result from incorrect command line options.

Lexical errors occur when the compiler encounters an unidentified or incomplete token. While they do not immediately terminate compilation, lexical errors do prevent the compiler from generating executable code.

Preprocessor errors can prevent the compiler from generating executable code.

While they do not immediately terminate compilation, syntax errors can prevent the compiler from generating executable code.

Warnings occur when the compiler finds a statement that is legitimate but probably not what you intended. Warnings are not errors and do NOT terminate compilation or prevent code generation.

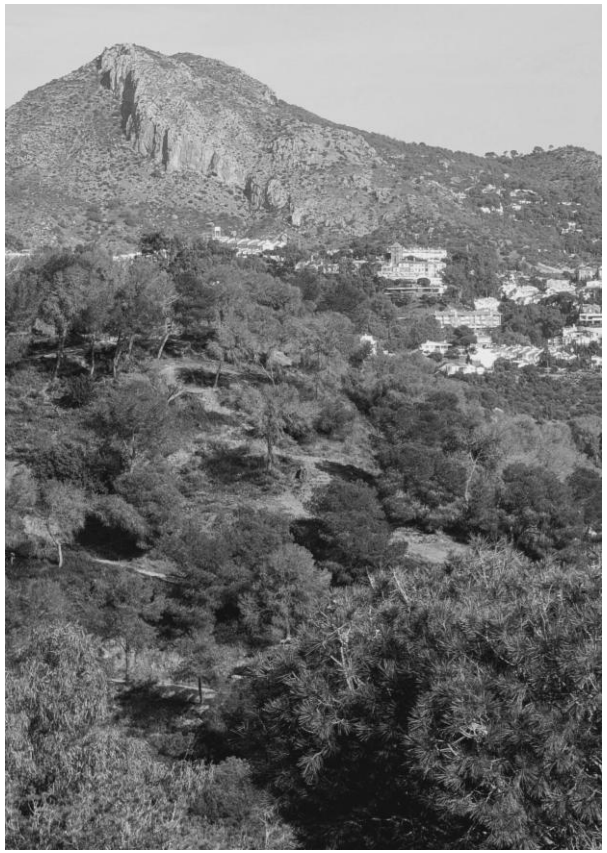
Fatal errors immediately terminate compilation, i.e., the compiler runs out of memory. Includes internal errors.

CONSTRAINT VIOLATIONS

The diagnostic message required by a constraint violation may be a warning or a fatal error.

If a warning is issued for a constraint violation, the compiler will still issue code which has undefined behavior.

OPTIMIZATIONS



Increasingly, compiler writers take advantage of UB in C and C++ to improve optimizations.

These optimizations can interfere with the ability to perform cause-effect analysis on source code, that is, analyzing the dependence of downstream results on prior results.

Consequently, these optimizations eliminate causality in software and increase the probability of software faults, defects, and vulnerabilities.

COMPILER OPTIMIZATIONS

The basic design of an optimizer for a C compiler is largely the same as an optimizer for any other procedural programming language.

The fundamental principle of optimization is to replace a computation with a more efficient method that computes the same result.

“AS IF” RULE

The C Standard specifies the results of computations as if on an abstract machine, but the methods used by the compiler are not specified.

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

The compiler’s optimizer is free to choose any method that produces the correct result.

IMPLEMENTATION STRATEGIES

Hardware behavior

- Generate the assembler code and let the hardware do what it does.
- Near universal policy for years, so many C and C++ programmers assume this is the way.

Super debug

- Provide an intensive debugging environment to trap (nearly) every undefined behavior.
- Severely degrades performance, so is seldom used for building applications.

Total license

- Treat any possible UB as a “can’t happen” condition
- Permits aggressive optimizations

C STANDARD COMMITTEE STUDY GROUPS

WG14 in Strasbourg, France January 2024

Photo Credit: Unattributed at request of the photographer.
https://www.open-std.org/jtc1/sc22/wg14/

Study Groups

WG14 has constituted an Undefined Behavior Study Group that is working on eliminating undefined behaviors from C2Y.

32 separate UBs removed from C2Y

woven BY TOYOTA

ABSTRACT UB

There is no complete and universally-accepted consensus about the interpretation of UB in the standard.

Prior to C23, a popular interpretation was that a program becomes completely invalid as soon as there is potentially an operation with undefined behavior at run-time.

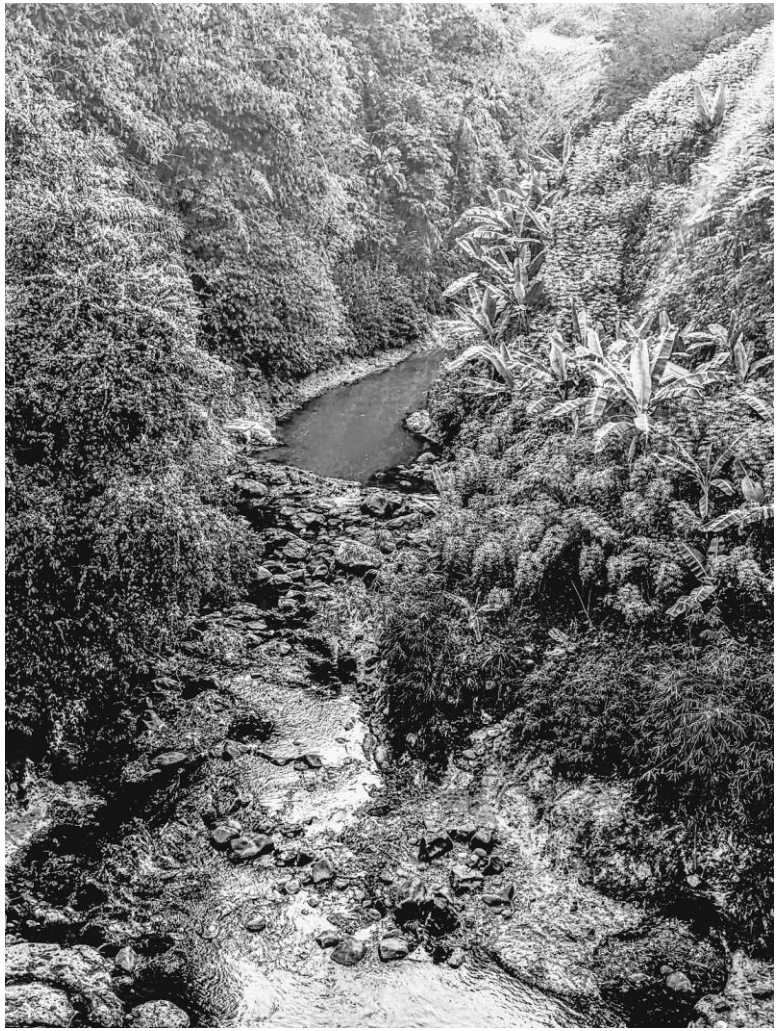
Among other things, this allowed “time traveling” undefined behavior

HOISTING OF LOOP-INVARIANT

```
signed int si1 = atoi(argv[1]);
  signed int si2 = atoi(argv[2]);
  signed int result = 8;
  size_t i;

  for (i = 0; i < 100; ++i) {
    if (argc == 8)
      i++;
    result += i + si1 % si2;
  }
```

Loop invariant; can be evaluated early



```
            for (i = 0; i < MAX; ++i) {

00401033  xor   ecx, ecx
00401035  idiv  eax,esi
00401037  mov   eax,dword ptr [esp+10h]
0040103B  mov   edi,8

                        if (argc == 8)

00401040  cmp   eax,8
00401043  jne   main+37h (401047h)

                                    i++;

00401045  inc   ecx
00401046  inc   edx


                        result += i + si1 % si2;

00401047  add   edi, edx
00401049  inc   ecx
0040104A  inc   edx
0040104B  cmp   ecx,64h
0040104E  jb    main+30h (401040h)

            }
```

Invariant subexpression evaluated once before start of loop causing undefined behavior and fault to occur much earlier in program.

OBSERVABLE SIDE-EFFECTS

```
signed int si1 = atoi(argv[1]);
signed int si2 = atoi(argv[2]);
signed int result = 8;
size_t i;

puts("log message one.\n");

for (i = 0; i < MAX; ++i) {
  puts("log message two.\n"); fflush(stdout);
  if (argc == 8) i++;
  result += i + si1 % si2;
  puts("log message three.\n"); fflush(stdout);
}
printf("Result = %d.\n", result); fflush(stdout);
```

UB can be retroactive with respect to program output, so it's OK to hoist inevitable UB before a call to a function declared in **<stdio.h>**.

EXECUTION OUTPUT

```
$ gcc optimize.c
$ ./a.out -2147483648 -1
log message one.
log message two.
Floating exception
$ gcc -O2 optimize.c
$ ./a.out -2147483648 -1
log message one.
Floating exception
```

CONCRETE UB IN C23

Any other behavior during execution of a program is only affected as a direct consequence of the concrete behavior that occurs when encountering the erroneous or non-portable program construct or data.

All observable behavior appears as specified in this document when it happens before an operation with UB in the execution of the program.

Eliminates “time traveling” UB among other things.

UB (40) GONE

A pointer is converted to other than an integer or pointer type.

```
<source>:6:13: error: conversion to non-scalar type requested
6 | (struct f)p;
  |          ^
```

A pointer type shall ~~not~~ be converted only to ~~any floating~~ an integer or pointer type....

By adding to the constraint section, a diagnostic (warning or error) is required.

ELIMINATING UB IN C

Some UB in C

- can be defined or made a constraint violation.
- can be detected at runtime for existing code.
- are memory safety issues that are difficult or expensive to detect

WHAT'S NEW IN C++?

UNDEFINED BEHAVIOR & ILL-FORMED, NO DIAGNOSTIC REQUIRED

CORE CATEGORIES

The C++ language has two large categories of “don’t do that” known as ***undefined behavior*** and ***ill-formed program, no diagnostic required***.

01 / NATURE

Undefined behavior is a runtime concept.

02 / PERMISSIONS

If the behavior is undefined by the standard, then the program is permitted by the standard to do anything it wants.

PROPOSAL ENUMERATION

P3596R3 “Undefined Behavior and IFNDR” 2026-05-11 enumerates all UB and IFNDR

03 / EXECUTION

There is no undefined behavior if the program doesn’t execute code paths with undefined behavior.

04 / MEANING

Undefined behavior often cannot be meaningful defined, such as divide by zero.

© 2026 Robert C. Seacord

UNDEFINED BEHAVIOR

DEFINITION & CAUSES

Behavior for which the C++ standard imposes no requirements.

Expected when explicit definition is omitted or when a program uses an incorrect construct or invalid data.

RULES & EXCEPTIONS

Diagnostics Required: Many incorrect constructs do not cause undefined behavior and must be diagnosed.

Constant Expressions: Evaluation of a constant expression never exhibits undefined behavior.

PERMISSIBLE UNDEFINED BEHAVIOR

01 / Unpredictable Execution

Ignoring the situation completely with unpredictable results.

02 / Documented Behavior

Behaving during translation or execution in a documented manner characteristic of the environment (with/without diagnostic).

03 / Program Termination

Terminating translation or execution with the issuance of a diagnostic message.

© 2026 Robert C. Seacord

ERRONEOUS BEHAVIOR (C++26)

DEFINITION & CONCEPT

Well-defined behavior that the implementation is **recommended to diagnose**.

Bridges the gap between fully valid code and completely unpredictable **Undefined Behavior (UB)**.

Describes code that is unmistakably a bug (**"Just Wrong"**) but is **not subject to the risks** associated with undefined behavior.

01 / DIAGNOSTIC PERMISSION

Incorrect Program Code

Erroneous behavior is always the consequence of incorrect program code.

Implementations are **allowed, but not required**, to diagnose it.

02 / CONSTANT EVALUATION

Compile-Time Exemption

Evaluation of a constant expression **never exhibits** erroneous behavior.

© 2026 Robert C. Seacord

ERRONEOUS BEHAVIOR EXAMPLE

> Erroneous behavior exists to fix a specific, real-world safety problem: reading a default-initialized local variable of scalar type.

erroneous_read_example.cpp

```cpp
void h() {
  int d1, d2;
  int e1 = d1; // erroneous behavior (reading d1 is a bug)
  int e2 = d1; // erroneous behavior
  assert(e1 == e2); // holds — same uninit object always yields the same value
  assert(e1 == d1); // holds, erroneous behavior (reading d1 again)
  std::memcpy(&d2, &d1, sizeof(int)); // NOT erroneous — copying bytes is fine
  assert(e1 == d2); // holds, erroneous behavior
}
```

RUNTIME-UNDEFINED BEHAVIOR

Behavior that is undefined **except** when it occurs during constant evaluation

01 During **constant evaluation**, it is **implementation-defined** whether **runtime-undefined behavior** results in the expression being deemed non-constant (i.e., rejected) or if it's just evaluated anyway.

02 There are currently two uses:

- `[[assume(expr)]]`, where the assumption is false
- returning from a function marked `[[noreturn]]`

03 A compiler that is evaluating a constant expression with an incorrect assumption can just ignore it and continue or declare it non-constant (like it would for "normal" undefined behavior), as well as a return from a function marked `[[noreturn]]`

C++26 CONTRACTS Feature Overview

Design

`pre()`/`post()`/`contract_assert()` lets any precondition/postcondition be made explicit, structurally part of a function's declaration

Standard Behavior

Doesn’t erase undefined behavior (UB) from standard text — UB remains as fallback when checking is off.

Practical Impact

Convert **silent, undiagnosed violations** into **explicit, checkable assertions**.

In practice, far fewer programs ever reach the undefined behavior state at runtime.

C++26 CONTRACTS

Practical Example

The Problem: Undocumented UB

Most real-world UB today comes from undocumented or prose-only preconditions (e.g. `v[i]` requires `i < v.size()`). Callers violate it, resulting in raw UB with no diagnostic.

The Solution: Early Diagnostics

By attaching `pre(...)` to the declaration, violation becomes something a debug/test build catches deterministically with a clear diagnostic before degrading into "anything can happen".

example.cpp

```
int& at(std::vector<int>& v, size_t i)
  pre(i < v.size()) // precondition
  post(r: &r == &v[i]) // postcondition
{
  contract_assert(
    i < v.size()
  ); // mid-function assertion
  return v[i];
}
```

SUMMARY & CONCLUSIONS

What's new in Undefined Behavior — C

- systematically remove unnecessary UB
- Make abstract UB concrete

What's new in Undefined Behavior — C++

- Erroneous behavior
- Runtime-undefined behavior
- Undefined Behavior and IFNDR Annexes
- Contracts `pre()`/`post()`/`contract_assert()` let any precondition/postcondition be made explicit, structurally part of a function's declaration

THANKS!

Robert C. Seacord

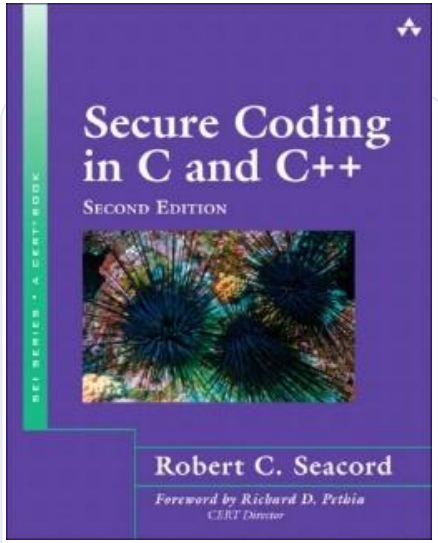
Senior Staff Software Engineer | Woven by Toyota

CONNECT & CONTACT

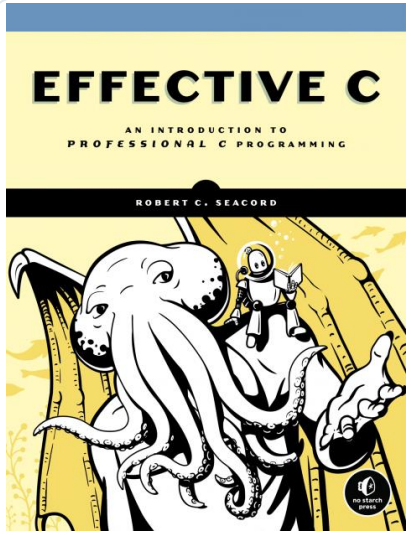
LinkedIn: LinkedIn.com/in/robertseacord/

Email: robert.seacord@woven.toyota

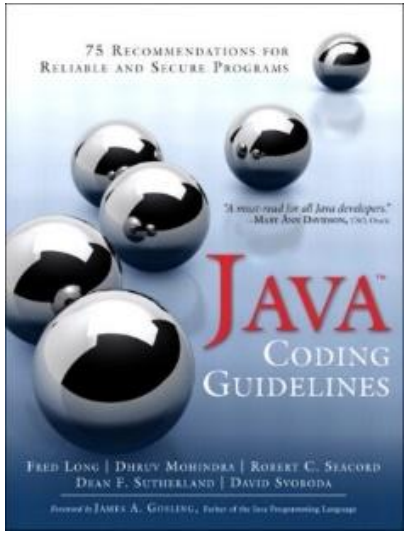
FEATURED PUBLICATIONS



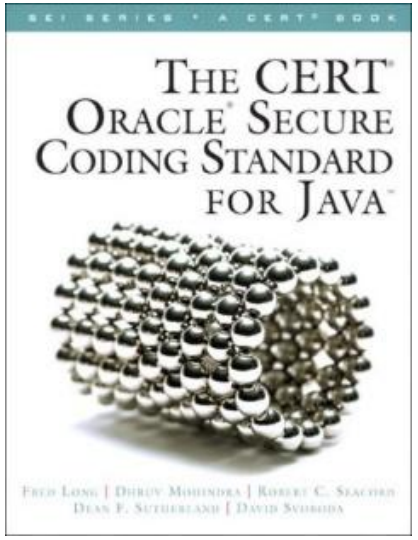
Secure Coding in C and C++
Second Edition



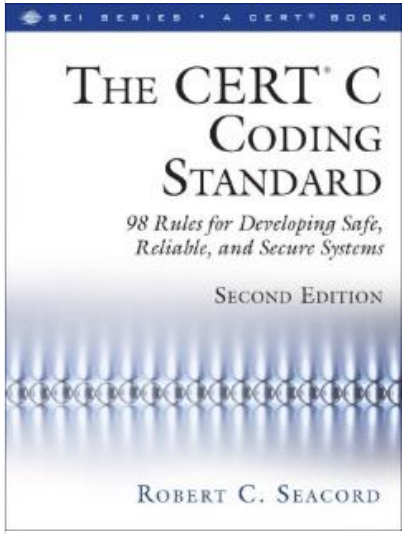
Effective C
Intro to Professional C



Java Coding Guidelines
75 Recommendations



CERT Oracle Secure Coding
Standard for Java



The CERT C Coding Standard
Second Edition