CON02-C. Do not use volatile as a synchronization primitive

The C Standard, subclause 5.1.2.3, paragraph 2 [ISO/IEC 9899:2011], says,

> *Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.*

The `volatile` keyword informs the compiler that the qualified variable may change in ways that cannot be determined; consequently, compiler optimizations must be restricted for memory areas marked as `volatile`. For example, the compiler is forbidden to load the value into a register and subsequently reuse the loaded value rather than accessing memory directly. This concept relates to multithreading because incorrect caching of a shared variable may interfere with the propagation of modified values between threads, causing some threads to view stale data.

The `volatile` keyword is sometimes misunderstood to provide atomicity for variables that are shared between threads in a multithreaded program. Because the compiler is forbidden to either cache variables declared as `volatile` in registers or to reorder the sequence of reads and writes to any given volatile variable, many programmers mistakenly believe that volatile variables can correctly serve as synchronization primitives. Although the compiler is forbidden to reorder the sequence of reads and writes to a particular volatile variable, it may legally reorder these reads and writes with respect to reads and writes to *other* memory locations. This reordering alone is sufficient to make volatile variables unsuitable for use as synchronization primitives.

Further, the `volatile` qualifier lacks any guarantees regarding the following desired properties necessary for a multithreaded program:

- **Atomicity:** Indivisible memory operations.
- **Visibility:** The effects of a write action by a thread are visible to other threads.
- **Ordering:** Sequences of memory operations by a thread are guaranteed to be seen in the same order by other threads.

The `volatile` qualifier lacks guarantees for any of these properties, both by definition and by the way it is implemented in various platforms. For more information on how `volatile` is implemented, consult DCL17-C. Beware of miscompiled volatile-qualified variables.

Noncompliant Code Example

This noncompliant code example attempts to use `flag` as a synchronization primitive:

```
bool flag = false;

void test() {
  while (!flag) {
    sleep(1000);
  }
}

void wakeup(){
  flag = true;
}

void debit(unsigned int amount){
  test();
  account_balance -= amount;
}
```

In this example, the value of `flag` is used to determine whether the critical section can be executed. Because the `flag` variable is not declared `volatile`, it may be cached in registers. Before the value in the register is written to memory, another thread might be scheduled to run, resulting in that thread reading stale data.

Noncompliant Code Example

This noncompliant code example uses `flag` as a synchronization primitive but qualifies `flag` as a `volatile` type:

```
volatile bool flag = false;

void test() {
  while (!flag){
    sleep(1000);
  }
}

void wakeup(){
  flag = true;
}

void debit(unsigned int amount) {
  test();
  account_balance -= amount;
}
```

Declaring `flag` as `volatile` solves the problem of values being cached, which causes stale data to be read. However, `volatile flag` still fails to provide the atomicity and visibility guarantees needed for synchronization primitives to work correctly. The `volatile` keyword does not promise to provide the guarantees needed for synchronization primitives.

Compliant Solution

This code uses a mutex to protect critical sections:

```
#include <threads.h>

int account_balance;
mtx_t flag;

/* Initialize flag */

int debit(unsigned int amount) {
  if (mtx_lock(&flag) == thrd_error) {
    return -1;  /* Indicate error */
  }

  account_balance -= amount; /* Inside critical section */

  if (mtx_unlock(&flag) == thrd_error) {
    return -1;  /* Indicate error */
  }

  return 0;
}
```

Compliant Solution (Critical Section, Windows)

This compliant solution uses a Microsoft Windows critical section object to make operations involving `account_balance` atomic [MSDN].

```
#include <Windows.h>

static volatile LONG account_balance;
CRITICAL_SECTION flag;

/* Initialize flag */
InitializeCriticalSection(&flag);

int debit(unsigned int amount) {
  EnterCriticalSection(&flag);
  account_balance -= amount; /* Inside critical section */
  LeaveCriticalSection(&flag);

  return 0;
}
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
CON02-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.4.2	CERT_C-CON02-a	Do not use the volatile keyword

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C	CON01-CPP. Do not use volatile as a synchronization primitive	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[IEEE Std 1003.1:2013]	Section 4.11, "Memory Synchronization"
[ISO/IEC 9899:2011]	Subclause 5.1.2.3, "Program Execution"
[MSDN]	