DCL56-CPP. Avoid cycles during initialization of static objects

The C++ Standard, [stmt.dcl], paragraph 4 [ISO/IEC 14882-2014], states the following:

> *The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise such a variable is initialized the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization. If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.*

Do not reenter a function during the initialization of a static variable declaration. If a function is reentered during the constant initialization of a static object inside that function, the behavior of the program is undefined. Infinite recursion is not required to trigger undefined behavior, the function need only recur once as part of the initialization. Due to thread-safe initialization of variables, a single, recursive call will often result in a deadlock due to locking a non-recursive synchronization primitive.

Additionally, the C++ Standard, [basic.start.init], paragraph 2, in part, states the following:

> *Dynamic initialization of a non-local variable with static storage duration is either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other non-local variables with static storage duration have ordered initialization. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread, the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization.*

Do not create an initialization interdependency between static objects with dynamic initialization unless they are ordered with respect to one another. Unordered initialization, especially prevalent across translation unit boundaries, results in unspecified behavior.

Noncompliant Code Example

This noncompliant example attempts to implement an efficient factorial function using caching. Because the initialization of the static local array `cache` involves recursion, the behavior of the function is undefined, even though the recursion is not infinite.

```
#include <stdexcept>

int fact(int i) noexcept(false) {
  if (i < 0) {
    // Negative factorials are undefined.
    throw std::domain_error("i must be >= 0");
  }

  static const int cache[] = {
    fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),
    fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),
    fact(12), fact(13), fact(14), fact(15), fact(16)
  };

  if (i < (sizeof(cache) / sizeof(int))) {
    return cache[i];
  }

  return i > 0 ? i * fact(i - 1) : 1;
}
```

Implementation Details

In Microsoft Visual Studio 2015 and GCC 6.1.0, the recursive initialization of `cache` deadlocks while initializing the static variable in a thread-safe manner.

Compliant Solution

This compliant solution avoids initializing the static local array `cache` and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet and, if not, recursively computes its value. It then returns the cached value when possible or computes the value as needed.

```
#include <stdexcept>

int fact(int i) noexcept(false) {
   if (i < 0) {
    // Negative factorials are undefined.
    throw std::domain_error("i must be >= 0");
  }

  // Use the lazy-initialized cache.
  static int cache[17];
  if (i < (sizeof(cache) / sizeof(int))) {
    if (0 == cache[i]) {
      cache[i] = i > 0 ? i * fact(i - 1) : 1;
    }
    return cache[i];
  }

  return i > 0 ? i * fact(i - 1) : 1;
}
```

Noncompliant Code Example

In this noncompliant code example, the value of `numWheels` in `file1.cpp` relies on `c` being initialized. However, because `c` is defined in a different translation unit (`file2.cpp`) than `numWheels`, there is no guarantee that `c` will be initialized by calling `get_default_car()` before `numWheels` is initialized by calling `c.get_num_wheels()`. This is often referred to as the "static initialization order fiasco," and the resulting behavior is unspecified.

```
// file.h
#ifndef FILE_H
#define FILE_H

class Car {
  int numWheels;

public:
  Car() : numWheels(4) {}
  explicit Car(int numWheels) : numWheels(numWheels) {}

  int get_num_wheels() const { return numWheels; }
};
#endif // FILE_H

// file1.cpp
#include "file.h"
#include <iostream>

extern Car c;
int numWheels = c.get_num_wheels();

int main() {
  std::cout << numWheels << std::endl;
}

// file2.cpp
#include "file.h"

Car get_default_car() { return Car(6); }
Car c = get_default_car();
```

Implementation Details

The value printed to the standard output stream will often rely on the order in which the translation units are linked. For instance, with Clang 3.8.0 on x86 Linux, the command `clang++ file1.cpp file2.cpp && ./a.out` will write `0` while `clang++ file2.cpp file1.cpp && ./a.out` will write 6.

Compliant Solution

This compliant solution uses the "construct on first use" idiom to resolve the static initialization order issue. The code for `file.h` and `file2.cpp` are unchanged; only the static `numWheels` in `file1.cpp` is moved into the body of a function. Consequently, the initialization of `numWheels` is guaranteed to happen when control flows over the point of declaration, ensuring control over the order. The global object `c` is initialized before execution of `main()` begins, so by the time `get_num_wheels()` is called, `c` is guaranteed to have already been dynamically initialized.

```
// file.h
#ifndef FILE_H
#define FILE_H

class Car {
  int numWheels;

public:
  Car() : numWheels(4) {}
  explicit Car(int numWheels) : numWheels(numWheels) {}

  int get_num_wheels() const { return numWheels; }
};
#endif // FILE_H

// file1.cpp
#include "file.h"
#include <iostream>

int &get_num_wheels() {
  extern Car c;
  static int numWheels = c.get_num_wheels();
  return numWheels;
}

int main() {
  std::cout << get_num_wheels() << std::endl;
}

// file2.cpp
#include "file.h"

Car get_default_car() { return Car(6); }
Car c = get_default_car();
```

Risk Assessment

Recursively reentering a function during the initialization of one of its static objects can result in an attacker being able to cause a crash or denial of service. Indeterminately ordered dynamic initialization can lead to undefined behavior due to accessing an uninitialized object.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL56-CPP	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
LDRA tool suite	9.7.1	**6 D**	Enhanced Enforcement
Parasoft C /C++test	10.4.2	**CERT_CPP-DCL56-a**	Avoid initialization order problems across translation units by replacing non-local static objects with local static objects

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

CERT Oracle Coding Standard for Java	DCL00-J. Prevent class initialization cycles

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.6.2, "Initialization of Non-local Variables" Subclause 6.7, "Declaration Statement"