DCL59-CPP. Do not define an unnamed namespace in a header file

Unnamed namespaces are used to define a namespace that is unique to the translation unit, where the names contained within have internal linkage by default. The C++ Standard, [namespace.unnamed], paragraph 1 [ISO/IEC 14882-2014], states the following:

> *An* unnamed-namespace-definition *behaves as if it were replaced by:*
>
> ```
> inline namespace unique { /* empty body */ }
> using namespace unique ;
> namespace unique { namespace-body }
> ```
>
> *where* `inline` *appears if and only if it appears in the* unnamed-namespace-definition*, all occurrences of* `unique` *in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the entire program.*

Production-quality C++ code frequently uses *header files* as a means to share code between translation units. A header file is any file that is inserted into a translation unit through an `#include` directive. Do not define an unnamed namespace in a header file. When an unnamed namespace is defined in a header file, it can lead to surprising results. Due to default internal linkage, each translation unit will define its own unique instance of members of the unnamed namespace that are ODR-used within that translation unit. This can cause unexpected results, bloat the resulting executable, or inadvertently trigger undefined behavior due to one-definition rule (ODR) violations.

Noncompliant Code Example

In this noncompliant code example, the variable v is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of v and then assigns a new value into it. However, because v is defined within an unnamed namespace, each translation unit operates on its own instance of v, resulting in unexpected output.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f();
  g();
}
```

When executed, this program prints the following.

```
f(): 0
g(): 0
f(): 42
g(): 100
```

Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

int v; // Definition of global variable v

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f(); // Prints v, sets it back to 42
  g(); // Prints v, sets it back to 100
}
```

When executed, this program prints the following.

```
f(): 0
g(): 42
f(): 100
g(): 42
```

Noncompliant Code Example

In this noncompliant code example, the variable `v` is defined in an unnamed namespace within a header file, and an inline function, `get_v()`, is defined, which accesses that variable. ODR-using the inline function from multiple translation units (as shown in the implementation of `f()` and `g()`) violates the one-definition rule because the definition of `get_v()` is not identical in all translation units due to referencing a unique `v` in each translation unit.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

inline int get_v() { return v; }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"

void f() {
  int i = get_v();
  // ...
}

// b.cpp
#include "a.h"

void g() {
  int i = get_v();
  // ...
}
```

See DCL60-CPP. Obey the one-definition rule for more information on violations of the one-definition rule.

Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units and can be accessed from the inline `get_v(` function.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

inline int get_v() {
  return v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"

// Externally used by get_v();
int v;

void f() {
  int i = get_v();
  // ...
}

// b.cpp
#include "a.h"

void g() {
  int i = get_v();
  // ...
}
```

Noncompliant Code Example

In this noncompliant code example, the function `f()` is defined within a header file. However, including the header file in multiple translation units causes a violation of the one-definition rule that usually results in an error diagnostic generated at link time due to multiple definitions of a function with the same name.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

void f() { /* ... */ }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Noncompliant Code Example

This noncompliant code example attempts to resolve the link-time errors by defining `f()` within an unnamed namespace. However, it produces multiple, unique definitions of `f()` in the resulting executable. If `a.h` is included from many translation units, it can lead to increased link times, a larger executable file, and reduced performance.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
void f() { /* ... */ }
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Compliant Solution

In this compliant solution, `f()` is not defined with an unnamed namespace and is instead defined as an inline function. Inline functions are required to be defined identically in all the translation units in which they are used, which allows an implementation to generate only a single instance of the function at runtime in the event the body of the function does not get generated for each call site.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

inline void f() { /* ... */ }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Risk Assessment

Defining an unnamed namespace within a header file can cause data integrity violations and performance problems but is unlikely to go unnoticed with sufficient testing. One-definition rule violations result in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL59-CPP	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC++-DCL59**	
Clang	3.9	`cert-dcl59-cpp`	Checked by `clang-tidy`
LDRA tool suite	9.7.1	**286 S, 512 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_CPP-DCL59-a**	There shall be no unnamed namespaces in header files
Polyspace Bug Finder	R2019b	CERT C++: DCL59-CPP	Checks for unnamed namespaces in header files (rule fully covered)
SonarQube C/C++ Plugin	4.10	**UnnamedNamespaceInHeader**	
PRQA QA-C++	4.4	**2518**	

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	DCL60-CPP. Obey the one-definition rule

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.2, "One Definition Rule" Subclause 7.1.2, "Function Specifiers" Subclause 7.3.1, "Namespace Definition"