DCL53-CPP. Do not write syntactically ambiguous declarations

It is possible to devise syntax that can ambiguously be interpreted as either an expression statement or a declaration. Syntax of this sort is called a *vexing parse* because the compiler must use disambiguation rules to determine the semantic results. The C++ Standard, [stmt.ambig], paragraph 1 [ISO/IEC 14882-2014], in part, states the following:

> *There is an ambiguity in the grammar involving* expression-statement*s and declarations: An* expression-statement *with a function-style explicit type conversion as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a* `(`*. In those cases the statement is a declaration. [Note: To disambiguate, the whole statement might have to be examined to determine if it is an* expression-statement *or a declaration. ...*

A similarly vexing parse exists within the context of a declaration where syntax can be ambiguously interpreted as either a function declaration or a declaration with a function-style cast as the initializer. The C++ Standard, [dcl.ambig.res], paragraph 1, in part, states the following:

> *The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration.*

Do not write a syntactically ambiguous declaration. With the advent of uniform initialization syntax using a braced-init-list, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name.

Noncompliant Code Example

In this noncompliant code example, an anonymous local variable of type `std::unique_lock` is expected to lock and unlock the mutex `m` by virtue of RAII. However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named `m` and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex>(m);
  shared_resource += 42;
}
```

Compliant Solution

In this compliant solution, the lock object is given an identifier (other than `m`) and the proper converting constructor is called.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex> lock(m);
  shared_resource += 42;
}
```

Noncompliant Code Example

In this noncompliant code example, an attempt is made to declare a local variable, `w`, of type `Widget` while executing the default constructor. However, this declaration is syntactically ambiguous where the code could be either a declaration of a function pointer accepting no arguments and returning a `Widget` or a declaration of a local variable of type `Widget`. The syntax used in this example defines the former instead of the latter.

```
#include <iostream>

struct Widget {
  Widget() { std::cout << "Constructed" << std::endl; }
};

void f() {
  Widget w();
}
```

As a result, this program compiles and prints no output because the default constructor is never actually invoked.

Compliant Solution

This compliant solution shows two equally compliant ways to write the declaration. The first way is to elide the parentheses after the variable declaration, which ensures the syntax is that of a variable declaration instead of a function declaration. The second way is to use a *braced-init-list* to direct-initialize the local variable.

```
#include <iostream>

struct Widget {
  Widget() { std::cout << "Constructed" << std::endl; }
};

void f() {
  Widget w1; // Elide the parentheses
  Widget w2{}; // Use direct initialization
}
```

Running this program produces the output `Constructed` twice, once for `w1` and once for `w2`.

Noncompliant Code Example

This noncompliant code example demonstrates a vexing parse. The declaration `Gadget g(Widget(i));` is not parsed as declaring a `Gadget` object with a single argument. It is instead parsed as a function declaration with a redundant set of parentheses around a parameter.

```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl; }
};

struct Gadget {
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" << std::endl; }
};

void f() {
  int i = 3;
  Gadget g(Widget(i));
  std::cout << i << std::endl;
}
```

Parentheses around parameter names are optional, so the following is a semantically identical spelling of the declaration.

```
Gadget g(Widget i);
```

As a result, this program is well-formed and prints only 3 as output because no `Gadget` or `Widget` objects are constructed.

Compliant Solution

This compliant solution demonstrates two equally compliant ways to write the declaration of `g`. The first declaration, `g1`, uses an extra set of parentheses around the argument to the constructor call, forcing the compiler to parse it as a local variable declaration of type `Gadget` instead of as a function declaration. The second declaration, `g2`, uses direct initialization to similar effect.

```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl; }
};

struct Gadget {
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" << std::endl; }
};

void f() {
  int i = 3;
  Gadget g1((Widget(i))); // Use extra parentheses
  Gadget g2{Widget(i)}; // Use direct initialization
  std::cout << i << std::endl;
}
```

Running this program produces the expected output.

```
Widget constructed
Gadget constructed
Widget constructed
Gadget constructed
3
```

Risk Assessment

Syntactically ambiguous declarations can lead to unexpected program execution. However, it is likely that rudimentary testing would uncover violations of this rule.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL53-CPP	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
LDRA tool suite	9.7.1	**296 S**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_CPP-DCL53-a** **CERT_CPP-DCL53-b**	Always declare functions at file scope Identifier declared in a local or function prototype scope shall not hide an identifier declared in a global or namespace scope
Polyspace Bug Finder	R2019a	CERT C++: DCL53-CPP	Checks for declarations that can be confused between: • Function and object declaration • Unnamed object or function parameter declaration Rule fully covered.
PRQA QA-C++	4.3	**2502, 2510**	
Clang	3.9	`-Wvexing-parse`	
SonarQube C/C++ Plugin	4.10	**S3468**	

Related Vulnerabilities

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

Bibliography

[ISO/IEC 14882-2014]	Subclause 6.8, "Ambiguity Resolution" Subclause 8.2, "Ambiguity Resolution"

[Meyers 2001]	Item 6, "Be Alert for C++'s Most Vexing Parse"