OOP50-CPP. Do not invoke virtual functions from constructors or destructors

Virtual functions allow for the choice of member function calls to be determined at run time based on the dynamic type of the object that the member function is being called on. This convention supports object-oriented programming practices commonly associated with object inheritance and function overriding. When calling a nonvirtual member function or when using a class member access expression to denote a call, the specified function is called. Otherwise, a virtual function call is made to the final overrider in the dynamic type of the object expression.

However, during the construction and destruction of an object, the rules for virtual method dispatch on that object are restricted. The C++ Standard, [class.cdtor], paragraph 4 [ISO/IEC 14882-2014], states the following:

Member functions, including virtual functions, can be called during construction or destruction. When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it \( \times \)) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access and the object expression refers to the complete object of \( \times \) or one of that object's base class subobjects but not \( \times \) or one of its base class subobjects, the behavior is undefined.

Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. The derived class has not had the opportunity to initialize its resources, which is why calling a virtual function from a constructor does not result in a call to a function in a more derived class. Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released.

Noncompliant Code Example

In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the `B::B()` constructor calls `B::seize()` rather than `D::seize()`. Likewise, the `B::~B()` destructor calls `B::release()` rather than `D::release()`.

```cpp
struct B {
    B() { seize(); }
    virtual ~B() { release(); }
    protected:
    virtual void seize();
    virtual void release();
};
struct D : B {
    virtual ~D() = default;
    protected:
    void seize() override {
        B::seize();
        // Get derived resources...
    }
    void release() override {
        // Release derived resources...
        B::release();
    }
};
```

The result of running this code is that no derived class resources will be seized or released during the initialization and destruction of object of type `D`. At the time of the call to `seize()` from `B::B()`, the `D` constructor has not been entered, and the behavior of the under-construction object will be to invoke `B::seize()` rather than `D::seize()`. A similar situation occurs for the call to `release()` in the base class destructor. If the functions `seize()` and `release()` were declared to be pure virtual functions, the result would be undefined behavior.

Compliant Solution

In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with `mine`) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources.
class B {
    void seize_mine();
    void release_mine();

public:
    B() { seize_mine(); }
    virtual ~B() { release_mine(); }

protected:
    virtual void seize() { seize_mine(); }
    virtual void release() { release_mine(); }
};

class D : public B {
    void seize_mine();
    void release_mine();

public:
    D() { seize_mine(); }
    virtual ~D() { release_mine(); }

protected:
    void seize() override {
        B::seize();
        seize_mine();
    }

    void release() override {
        release_mine();
        B::release();
    }
};

Exceptions

OOP50-CPP-EX1: Because valid use cases exist that involve calling (non-pure) virtual functions from the constructor of a class, it is permissible to call the
virtual function with an explicitly qualified ID. The qualified ID signifies to code maintainers that the expected behavior is for the class under construction or
destruction to be the final overrider for the function call.

```cpp
struct A {
    A() {
        // f();   // WRONG!
        A::f();   // Okay
    }
    virtual void f();
};
```

OOP50-CPP-EX2: It is permissible to call a virtual function that has the `final` `virt-specifier` from a constructor or destructor, as in this example.

```cpp
struct A {
    A();
    virtual void f();
};

struct B : A {
    B() : A() {
        f();   // Okay
    }
    void f() override final;
};
```

Similarly, it is permissible to call a virtual function from a constructor or destructor of a class that has the `final` `class-virt-specifier`, as in this example.
struct A {
    A();
    virtual void f();
};

struct B final : A {
    B(): A() {
        f(); // Okay
    }
    void f() override;
};

In either case, \( f() \) must be the final overrider, guaranteeing consistent behavior of the function being called.

**Risk Assessment**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOP50-CPP</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC++-OOP50</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>467 S, 92 D</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_CPP-OOP50-a, CERT_CPP-OOP50-b, CERT_CPP-OOP50-c, CERT_CPP-OOP50-d</td>
<td>Avoid calling virtual functions from constructors Avoid calling virtual functions from destructors Do not invoke class's virtual functions from any of its constructors Do not invoke class's virtual functions from its destructor</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.3</td>
<td>4260, 4261, 4273, 4274, 4275, 4276, 4277, 4278, 4279, 4280, 4281, 4282</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>clang-analyzer-alpha.cplusplus.VirtualCall</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>4.10</td>
<td>S1699</td>
<td></td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

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

**Bibliography**

[Dewhurst 2002] Gotcha #75, "Calling Virtual Functions in Constructors and Destructors"
[ISO/IEC 14882-2014] Subclause 5.5, "Pointer-to-Member Operators"
[Lockheed Martin 2005] AV Rule 71.1, "A class's virtual functions shall not be invoked from its destructor or any of its constructors"
[Sutter 2004] Item 49, "Avoid Calling Virtual Functions in Constructors and Destructors"