DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope

Allocation and deallocation functions can be overloaded at both global and class scopes.

If an allocation function is overloaded in a given scope, the corresponding deallocation function must also be overloaded in the same scope (and vice versa).

Failure to overload the corresponding dynamic storage function is likely to violate rules such as MEM51-CPP. Properly deallocate dynamically allocated resources. For instance, if an overloaded allocation function uses a private heap to perform its allocations, passing a pointer returned by it to the default deallocation function will likely cause undefined behavior. Even in situations in which the allocation function ultimately uses the default allocator to obtain a pointer to memory, failing to overload a corresponding deallocation function may leave the program in an unexpected state by not updating internal data for the custom allocator.

It is acceptable to define a deleted allocation or deallocation function without its corresponding free store function. For instance, it is a common practice to define a deleted non-placement allocation or deallocation function as a class member function when the class also defines a placement new function. This prevents accidental allocation via calls to new for that class type or deallocation via calls to delete on pointers to an object of that class type. It is acceptable to declare, but not define, a private allocation or deallocation function without its corresponding free store function for similar reasons. However, a definition must not be provided as that still allows access to the free store function within a class member function.

Noncompliant Code Example

In this noncompliant code example, an allocation function is overloaded at global scope. However, the corresponding deallocation function is not declared. Were an object to be allocated with the overloaded allocation function, any attempt to delete the object would result in undefined behavior. For instance, if an overloaded allocation function uses a private heap, passing a pointer returned by it to the default deallocation function will likely cause undefined behavior. Even in situations in which the allocation function ultimately uses the default allocator to obtain a pointer to memory, failing to overload a corresponding deallocation function may leave the program in an unexpected state by not updating internal data for the custom allocator.

```cpp
#include <Windows.h>
#include <new>

void *operator new(std::size_t size) noexcept(false) {
    static HANDLE h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
    if (h) {
        return ::HeapAlloc(h, 0, size);
    }
    throw std::bad_alloc();
}

// No corresponding global delete operator defined.
```

Compliant Solution

In this compliant solution, the corresponding deallocation function is also defined at global scope.
```cpp
#include <Windows.h>
#include <new>

class HeapAllocator {
    static HANDLE h;
    static bool init;

public:
    static void *alloc(std::size_t size) noexcept(false) {
        if (!init) {
            h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
            init = true;
        }

        if (h) {
            return ::HeapAlloc(h, 0, size);
        }

        throw std::bad_alloc();
    }

    static void dealloc(void *ptr) noexcept {
        if (h) {
            (void)::HeapFree(h, 0, ptr);
        }
    }

    HANDLE HeapAllocator::h = nullptr;
    bool HeapAllocator::init = false;

    void *operator new(std::size_t size) noexcept(false) {
        void *ptr = ::operator new(size);
        update_bookkeeping(ptr, size, true);
        return ptr;
    }

    void operator delete(void *ptr) noexcept {
        return HeapAllocator::dealloc(ptr);
    }
};
```

Noncompliant Code Example

In this noncompliant code example, `operator new()` is overloaded at class scope, but `operator delete()` is not similarly overloaded at class scope. Despite that the overloaded allocation function calls through to the default global allocation function, were an object of type `S` to be allocated, any attempt to delete the object would result in leaving the program in an indeterminate state due to failing to update allocation bookkeeping accordingly.

```cpp
#include <new>

extern "C++" void update_bookkeeping(void *allocated_ptr, std::size_t size, bool alloc);

struct S {
    void *operator new(std::size_t size) noexcept(false) {
        void *ptr = ::operator new(size);
        update_bookkeeping(ptr, size, true);
        return ptr;
    }
};
```

Compliant Solution

In this compliant solution, the corresponding `operator delete()` is overloaded at the same class scope.
#include <new>

extern "C++" void update_bookkeeping(void *allocated_ptr, std::size_t size, bool alloc);

struct S {
  void *operator new(std::size_t size) noexcept(false) {
    void *ptr = ::operator new(size);
    update_bookkeeping(ptr, size, true);
    return ptr;
  }

  void operator delete(void *ptr, std::size_t size) noexcept {
    ::operator delete(ptr);
    update_bookkeeping(ptr, size, false);
  }
};

Exceptions

**DCL54-CPP-EX1**: A placement deallocation function may be elided for a corresponding placement allocation function, but only if the object placement allocation and object construction are guaranteed to be `noexcept(true)`. Because placement deallocation functions are automatically invoked when the object initialization terminates by throwing an exception, it is safe to elide the placement deallocation function when exceptions cannot be thrown. For instance, some vendors implement compiler flags disabling exception support (such as `-fno-cxx-exceptions` in Clang and `/EHs-c-` in Microsoft Visual Studio), which has implementation-defined behavior when an exception is thrown but generally results in program termination similar to calling `abort()`.

Risk Assessment

Mismatched usage of `new` and `delete` could lead to a denial-of-service attack.

<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>DCL54-CPP</td>
<td>Low</td>
<td>Probable</td>
<td>Low</td>
<td>P6</td>
<td>L2</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>Astrée</td>
<td>20.10</td>
<td>new-delete-pairwise</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC++-DCL54</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>misc-new-delete-overloads</td>
<td>Checked with clang-tidy</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.2</td>
<td>C++2160</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2021.4</td>
<td>CERT_DCLSAME_SCOPE_ALLOC_DEALLOC</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++ test</td>
<td>2021.2</td>
<td>CERT_CPP-DCL54-a</td>
<td>Always provide new and delete together</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021b</td>
<td>CERT C++: DCL54-CPP</td>
<td>Checks for mismatch between overloaded operator new and operator delete (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>2160</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>20.10</td>
<td>new-delete-pairwise</td>
<td>Partially checked</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>4.10</td>
<td>S1265</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard | MEM51-CPP. Properly deallocate dynamically allocated resources
<table>
<thead>
<tr>
<th>Bibliography</th>
</tr>
</thead>
</table>
| **[ISO/IEC 14882-2014]** | Subclause 3.7.4, "Dynamic Storage Duration"  
Subclause 5.3.4, "New"  
Subclause 5.3.5, "Delete" |