DCL50-CPP. Do not define a C-style variadic function

Functions can be defined to accept more formal arguments at the call site than are specified by the parameter declaration clause. Such functions are called variadic functions because they can accept a variable number of arguments from a caller. C++ provides two mechanisms by which a variadic function can be defined: function parameter packs and use of a C-style ellipsis as the final parameter declaration.

Variadic functions are flexible because they accept a varying number of arguments of differing types. However, they can also be hazardous. A variadic function using a C-style ellipsis (hereafter called a C-style variadic function) has no mechanisms to check the type safety of arguments being passed to the function or to check that the number of arguments being passed matches the semantics of the function definition. Consequently, a runtime call to a C-style variadic function that passes inappropriate arguments yields undefined behavior. Such undefined behavior could be exploited to run arbitrary code.

Do not define C-style variadic functions. (The declaration of a C-style variadic function that is never defined is permitted, as it is not harmful and can be useful in unevaluated contexts.)

Issues with C-style variadic functions can be avoided by using variadic functions defined with function parameter packs for situations in which a variable number of arguments should be passed to a function. Additionally, function currying can be used as a replacement to variadic functions. For example, in contrast to C’s printf() family of functions, C++ output is implemented with the overloaded single-argument std::cout::operator<<() operators.

Noncompliant Code Example

This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior.

```cpp
#include <cstdarg>

int add(int first, int second, ...) {  
    int r = first + second;
    va_list va;
    va_start(va, second);
    while (int v = va_arg(va, int)) {  
        r += v;
    }
    va_end(va);
    return r;
}
```

Compliant Solution (Recursive Pack Expansion)

In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior.

```cpp
#include <type_traits>

template <typename Arg, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Arg s) { return f + s; }

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Ts... rest) {  
    return f + add(rest...);
}
```

This compliant solution makes use of std::enable_if to ensure that any nonintegral argument value results in an ill-formed program.

Compliant Solution (Braced Initializer List Expansion)

An alternative compliant solution that does not require recursive expansion of the function parameter pack instead expands the function parameter pack into a list of values as part of a braced initializer list. Since narrowing conversions are not allowed in a braced initializer list, the type safety is preserved despite the std::enable_if not involving any of the variadic arguments.

```cpp
#include <type_traits>

template <typename Arg, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Arg s) { return f + s; }

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Ts... rest) {  
    return f + add(rest...);
}
```
```cpp
#include <type_traits>

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg i, Arg j, Ts... all) {
    int values[] = { j, all... };  
    int r = i;
    for (auto v : values) {
        r += v;
    }
    return r;
}

Exceptions

DCL50-CPP-EX1: It is permissible to define a C-style variadic function if that function also has external C language linkage. For instance, the function may be a definition used in a C library API that is implemented in C++.

DCL50-CPP-EX2: As stated in the normative text, C-style variadic functions that are declared but never defined are permitted. For example, when a function call expression appears in an unevaluated context, such as the argument in a `sizeof` expression, overload resolution is performed to determine the result type of the call but does not require a function definition. Some template metaprogramming techniques that employ SFINAE use variadic function declarations to implement compile-time type queries, as in the following example.

```cpp
template <typename Ty>
class has_foo_function {
    typedef char yes[1];
    typedef char no[2];

    template <typename Inner>
    static yes& test(Inner *I, decltype(I->foo()) * = nullptr); // Function is never defined.

    template <typename>
    static no& test(...); // Function is never defined.

    public:
    static const bool value = sizeof(test<Ty>(nullptr)) == sizeof(yes);
};
```

In this example, the value of `value` is determined on the basis of which overload of `test()` is selected. The declaration of `Inner *I` allows use of the variable `I` within the `decltype` specifier, which results in a pointer of some (possibly `void`) type, with a default value of `nullptr`. However, if there is no declaration of `Inner::foo()`, the `decltype` specifier will be ill-formed, and that variant of `test()` will not be a candidate function for overload resolution due to SFINAE. The result is that the C-style variadic function variant of `test()` will be the only function in the candidate set. Both `test()` functions are declared but never defined because their definitions are not required for use within an unevaluated expression context.

Risk Assessment

Incorrectly using a variadic function can result in abnormal program termination, unintended information disclosure, or execution of arbitrary code.

<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>DCL50-CPP</td>
<td>High</td>
<td>Probable</td>
<td>Medium</td>
<td>P12</td>
<td>L1</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>22.10</td>
<td>function-ellipsis</td>
<td>Fully checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC++-DCL50</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>cert-dcl50-cpp</td>
<td>Checked by clang-tidy.</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.2p0</td>
<td>LANG.STRUCT.ELLIPSIS</td>
<td>Ellipsis</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.3</td>
<td>C++2012, C++2625</td>
<td></td>
</tr>
</tbody>
</table>
### Related Vulnerabilities

Search for other vulnerabilities resulting from the violation of this rule on the [CERT website](https://cert.net).  

### Bibliography

| ISO/IEC 14882-2014 | Subclause 5.2.2, “Function Call”  
|-------------------|---------------------------------  
|                   | Subclause 14.5.3, “Variadic Templates” |