DCL51-CPP. Do not declare or define a reserved identifier

The C++ Standard, [reserved.names] [ISO/IEC 14882-2014], specifies the following rules regarding reserved names:

> - *A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.*
> - *A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 3, or to the* attribute-token*s described in 7.6.*
> - *Each name that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.*
> - *Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.*
> - *Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage, both in namespace `std` and in the global namespace.*
> - *Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.*
> - *Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with `extern "C"` linkage, both in namespace `std` and in the global namespace.*
> - *Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern "C"` and `extern "C++"` linkage, or as a name of namespace scope in the global namespace.*
> - *For each type T from the Standard C library, the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.*
> - *Literal suffix identifiers that do not start with an underscore are reserved for future standardization.*

The identifiers and attribute names referred to in the preceding excerpt are `override`, `final`, `alignas`, `carries_dependency`, `deprecated`, and `noreturn`.

No other identifiers are reserved. Declaring or defining an identifier in a context in which it is reserved results in undefined behavior. Do not declare or define a reserved identifier.

Noncompliant Code Example (Header Guard)

A common practice is to use a macro in a preprocessor conditional that guards against multiple inclusions of a header file. While this is a recommended practice, many programs use reserved names as the header guards. Such a name may clash with reserved names defined by the implementation of the C++ standard template library in its headers or with reserved names implicitly predefined by the compiler even when no C++ standard library header is included.

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

// Contents of <my_header.h>

#endif // _MY_HEADER_H_
```

Compliant Solution (Header Guard)

This compliant solution avoids using leading or trailing underscores in the name of the header guard.

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>

#endif // MY_HEADER_H
```

Noncompliant Code Example (User-Defined Literal)

In this noncompliant code example, a user-defined literal `operator"" x` is declared. However, literal suffix identifiers are required to start with an underscore; literal suffixes without the underscore prefix are reserved for future library implementations.

```
#include <cstddef>

unsigned int operator"" x(const char *, std::size_t);
```

Compliant Solution (User-Defined Literal)

In this compliant solution, the user-defined literal is named `operator"" _x`, which is not a reserved identifier.

```
#include <cstddef>

unsigned int operator"" _x(const char *, std::size_t);
```

The name of the user-defined literal is `operator"" _x` and not `_x`, which would have otherwise been reserved for the global namespace.

Noncompliant Code Example (File Scope Objects)

In this noncompliant code example, the names of the file scope objects `_max_limit` and `_limit` both begin with an underscore. Because it is `static`, the declaration of `_max_limit` might seem to be impervious to clashes with names defined by the implementation. However, because the header `<cstddef>` is included to define `std::size_t`, a potential for a name clash exists. (Note, however, that a conforming compiler may implicitly declare reserved names regardless of whether any C++ standard template library header has been explicitly included.) In addition, because `_limit` has external linkage, it may clash with a symbol with the same name defined in the language runtime library even if such a symbol is not declared in any header. Consequently, it is unsafe to start the name of any file scope identifier with an underscore even if its linkage limits its visibility to a single translation unit.

```
#include <cstddef> // std::for size_t

static const std::size_t _max_limit = 1024;
std::size_t _limit = 100;

unsigned int get_value(unsigned int count) {
  return count < _limit ? count : _limit;
}
```

Compliant Solution (File Scope Objects)

In this compliant solution, file scope identifiers do not begin with an underscore.

```
#include <cstddef> // for size_t

static const std::size_t max_limit = 1024;
std::size_t limit = 100;

unsigned int get_value(unsigned int count) {
  return count < limit ? count : limit;
}
```

Noncompliant Code Example (Reserved Macros)

In this noncompliant code example, because the C++ standard template library header `<cinttypes>` is specified to include `<cstdint>`, as per [c.files] paragraph 4 [ISO/IEC 14882-2014], the name `MAX_SIZE` conflicts with the name of the `<cstdint>` header macro used to denote the upper limit of `std:size_t`.

```
#include <cinttypes> // for int_fast16_t

void f(std::int_fast16_t val) {
  enum { MAX_SIZE = 80 };
  // ...
}
```

Compliant Solution (Reserved Macros)

This compliant solution avoids redefining reserved names.

```
#include <cinttypes> // for std::int_fast16_t

void f(std::int_fast16_t val) {
  enum { BufferSize = 80 };
  // ...
}
```

Exceptions

DCL51-CPP-EX1: For compatibility with other compiler vendors or language standard modes, it is acceptable to create a macro identifier that is the same as a reserved identifier so long as the behavior is semantically identical, as in this example.

```
// Sometimes generated by configuration tools such as autoconf
#define const const

// Allowed compilers with semantically equivalent extension behavior
#define inline __inline
```

DCL51-CPP-EX2: As a compiler vendor or standard library developer, it is acceptable to use identifiers reserved for your implementation. Reserved identifiers may be defined by the compiler, in standard library headers, or in headers included by a standard library header, as in this example declaration from the libc++ STL implementation.

```
// The following declaration of a reserved identifier exists in the libc++ implementation of
// std::basic_string as a public member. The original source code may be found at:
// http://llvm.org/svn/llvm-project/libcxx/trunk/include/string

template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_string {
  // ...

  bool __invariants() const;
};
```

Risk Assessment

Using reserved identifiers can lead to incorrect program operation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL51-CPP	Low	Unlikely	Low	**P3**	**L3**

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC++-DCL51**	
Clang	3.9	`-Wreserved-id-macro` `-Wuser-defined-literals`	The `-Wreserved-id-macro` flag is not enabled by default or with `-Wall`, but is enabled with `-Weverything`. This flag does not catch all instances of this rule, such as redefining reserved names.
CodeSonar	5.2p0	**LANG.ID.NU.MK** **LANG.STRUCT.DECL.RESERVED**	Macro name is C keyword Declaration of reserved name
LDRA tool suite	9.7.1	**86 S, 218 S, 219 S, 580 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_CPP-DCL51-a** **CERT_CPP-DCL51-b** **CERT_CPP-DCL51-c** **CERT_CPP-DCL51-d** **CERT_CPP-DCL51-e** **CERT_CPP-DCL51-f**	Do not #define or #undef identifiers with names which start with underscore Do not redefine reserved words Do not #define nor #undef identifier 'defined' The names of standard library macros, objects and functions shall not be reused The names of standard library macros, objects and functions shall not be reused (C90) The names of standard library macros, objects and functions shall not be reused (C99)

Polyspace Bug Finder	R2019b	CERT C++: DCL51-CPP	Checks for redefinitions of reserved identifiers (rule partially covered)
PRQA QA-C++	4.4	**5003**	
SonarQube C/C++ Plugin	4.10	**978**	

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	DCL58-CPP. Do not modify the standard namespaces
SEI CERT C Coding Standard	DCL37-C. Do not declare or define a reserved identifier PRE06-C. Enclose header files in an include guard
MISRA C++:2008	Rule 17-0-1

Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.4.3, "Reserved Names"
[ISO/IEC 9899:2011]	Subclause 7.1.3, "Reserved Identifiers"