2.9 DCL58-CPP. Do not modify the standard namespaces

Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate translation units. For instance, the following declarations are well-formed.

```
namespace MyNamespace {
int i;
}

namespace MyNamespace {
int i;
}

void f() {
  MyNamespace::i = MyNamespace::i = 12;
}
```

The standard library introduces the namespace `std` for standards-provided declarations such as `std::string`, `std::vector`, and `std::for_each`. However, it is undefined behavior to introduce new declarations in namespace `std` except under special circumstances. The C++ Standard, [namespace.std], paragraphs 1 and 2 [ISO/IEC 14882-2014], states the following:

> 1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.
>
> 2 The behavior of a C++ program is undefined if it declares
>
> - an explicit specialization of any member function of a standard library class template, or
> - an explicit specialization of any member function template of a standard library class or class template, or
> - an explicit or partial specialization of any member class template of a standard library class or class template.

In addition to restricting extensions to the namespace `std`, the C++ Standard, [namespace.posix], paragraph 1, further states the following:

> The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified. The namespace `posix` is reserved for use by ISO/IEC 9945 and other POSIX standards.

Do not add declarations or definitions to the standard namespaces `std` or `posix`, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.

The Library Working Group, responsible for the wording of the Standard Library section of the C++ Standard, has an unresolved issue on the definition of *user-defined type*. Although the Library Working Group has no official stance on the definition [INCITS 2014], we define it to be any `class`, `struct`, `union`, or `enum` that is not defined within namespace `std` or a namespace contained within namespace `std`. Effectively, it is a user-provided type instead of a standard library–provided type.

2.9.1 Noncompliant Code Example

In this noncompliant code example, the declaration of `x` is added to the namespace `std`, resulting in undefined behavior.

```
namespace std {
int x;
}
```

2.9.2 Compliant Solution

This compliant solution assumes the intention of the programmer was to place the declaration of `x` into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace `std`, the declaration is placed into a namespace without a reserved name.

```
namespace nonstd {
int x;
}
```

2.9.3 Noncompliant Code Example

In this noncompliant code example, a template specialization of `std::plus` is added to the namespace `std` in an attempt to allow `std::plus` to concatenate a `std::string` and `MyString` object. However, because the template specialization is of a standard library–provided type (`std::string`), this code results in undefined behavior.

```
#include <functional>
#include <iostream>
#include <string>

class MyString {
  std::string data;

public:
  MyString(const std::string &data) : data(data) {}

  const std::string &get_data() const { return data; }
};

namespace std {
template <>
struct plus<string> : binary_function<string, MyString, string> {
  string operator()(const string &lhs, const MyString &rhs) const {
    return lhs + rhs.get_data();
  }
};
}

void f() {
  std::string s1("My String");
  MyString s2(" + Your String");
  std::plus<std::string> p;

  std::cout << p(s1, s2) << std::endl;
}
```

2.9.4 Compliant Solution

The interface for `std::plus` requires that both arguments to the function call operator and the return type are of the same type. Because the attempted specialization in the noncompliant code example results in undefined behavior, this compliant solution defines a new `std::binary_function` derivative that can add a `std::string` to a `MyString` object without requiring modification of the namespace `std`.

```
#include <functional>
#include <iostream>
#include <string>

class MyString {
  std::string data;

public:
  MyString(const std::string &data) : data(data) {}

  const std::string &get_data() const { return data; }
};

struct my_plus
      : std::binary_function<std::string, MyString, std::string> {
    std::string operator()(
      const std::string &lhs, const MyString &rhs) const {
    return lhs + rhs.get_data();
  }
};

void f() {
  std::string s1("My String");
  MyString s2(" + Your String");
  my_plus p;

  std::cout << p(s1, s2) << std::endl;
}
```

2.9.5 Compliant Solution

In this compliant solution, a specialization of `std::plus` is added to the `std` namespace, but the specialization depends on a user-defined type and meets the Standard Template Library requirements for the original template, so it complies with this rule. However, because `MyString` can be constructed from `std::string`, this compliant solution involves invoking a converting constructor whereas the previous compliant solution does not.

```
#include <functional>
#include <iostream>
#include <string>

class MyString {
  std::string data;

public:
  MyString(const std::string &data) : data(data) {}

  const std::string &get_data() const { return data; }
};

namespace std {
template <>
struct plus<MyString> {
  MyString operator()(const MyString &lhs, const MyString &rhs)
const {
    return lhs.get_data() + rhs.get_data();
  }
};
}

void f() {
  std::string s1("My String");
  MyString s2(" + Your String");
  std::plus<MyString> p;

  std::cout << p(s1, s2).get_data() << std::endl;
}
```

2.9.6 Risk Assessment

Altering the standard namespace can cause undefined behavior in the C++ standard library.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL58-CPP	High	Unlikely	Medium	**P6**	**L2**

2.9.7 Related Guidelines

SEI CERT C++ Coding Standard	DCL51-CPP. Do not declare or define a reserved identifier

2.9.8 Bibliography

[INCITS 2014]	Issue 2139, "What Is a *User-Defined* Type?"
[ISO/IEC 14882-2014]	Subclause 17.6.4.2.1, "Namespace `std`" Subclause 17.6.4.2.2, "Namespace `posix`"

2.10 DCL59-CPP. Do not define an unnamed namespace in a header file

Unnamed namespaces are used to define a namespace that is unique to the translation unit, where the names contained within have internal linkage by default. The C++ Standard, [namespace.unnamed], paragraph 1 [ISO/IEC 14882-2014], states the following:

> An *unnamed-namespace-definition* behaves as if it were replaced by:
>
> ```
> inline namespace unique { /* empty body */ }
> using namespace unique ;
> namespace unique { namespace-body }
> ```
>
> where `inline` appears if and only if it appears in the *unnamed-namespace-definition*, all occurrences of `unique` in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the entire program.

Production-quality C++ code frequently uses *header files* as a means to share code between translation units. A header file is any file that is inserted into a translation unit through an `#include` directive. Do not define an unnamed namespace in a header file. When an unnamed namespace is defined in a header file, it can lead to surprising results. Due to default internal linkage, each translation unit will define its own unique instance of members of the unnamed namespace that are ODR-used within that translation unit. This can cause unexpected results, bloat the resulting executable, or inadvertently trigger undefined behavior due to one-definition rule (ODR) violations.