Standard Library Compatibility Guidelines (SD-8)

Titus Winters (titus@google.com)

Google

Upgrades: Cheap?

When you upgrade to a new version of the language,

is that an easy process?

Upgrades: Cheap?

Would you like it to be easier?

Upgrades: Cheap?

Would you like me to tell you what types of changes the committee might make?

Upgrades: Cheap?

You’re in the right room.

Stable Code

Your project works, you'll never upgrade?

Stable Code?

You’re in the wrong room.

(Go see Ben’s talk.)

History: Chaos

Which of the following is UB?

- `Foo(&std::move);`
- ```
  std::vector<int> v;
  Foo(&v.size());
  ```
- ```
  namespace std {
  class MyClass { … };
  }
  ```

History: Chaos

Which of the following is UB?

- `Foo(&std::move);`
- ```
  std::vector<int> v;
  Foo(&v.size());
  ```
- ```
  namespace std {
  class MyClass { … };
  }
  ```

History: Chaos

Which of the following is UB?

- `Foo(&std::move);  // honestly, still iffy`
- ```
  std::vector<int> v;
  Foo(&v.size());
  ```
- ```
  namespace std {
  class MyClass { … };
  }
  ```

History: Chaos

The committee could do a better job.

- Tell you what is out-of-bounds
- Stick to that

Recent: Some hope

Recent: Some hope

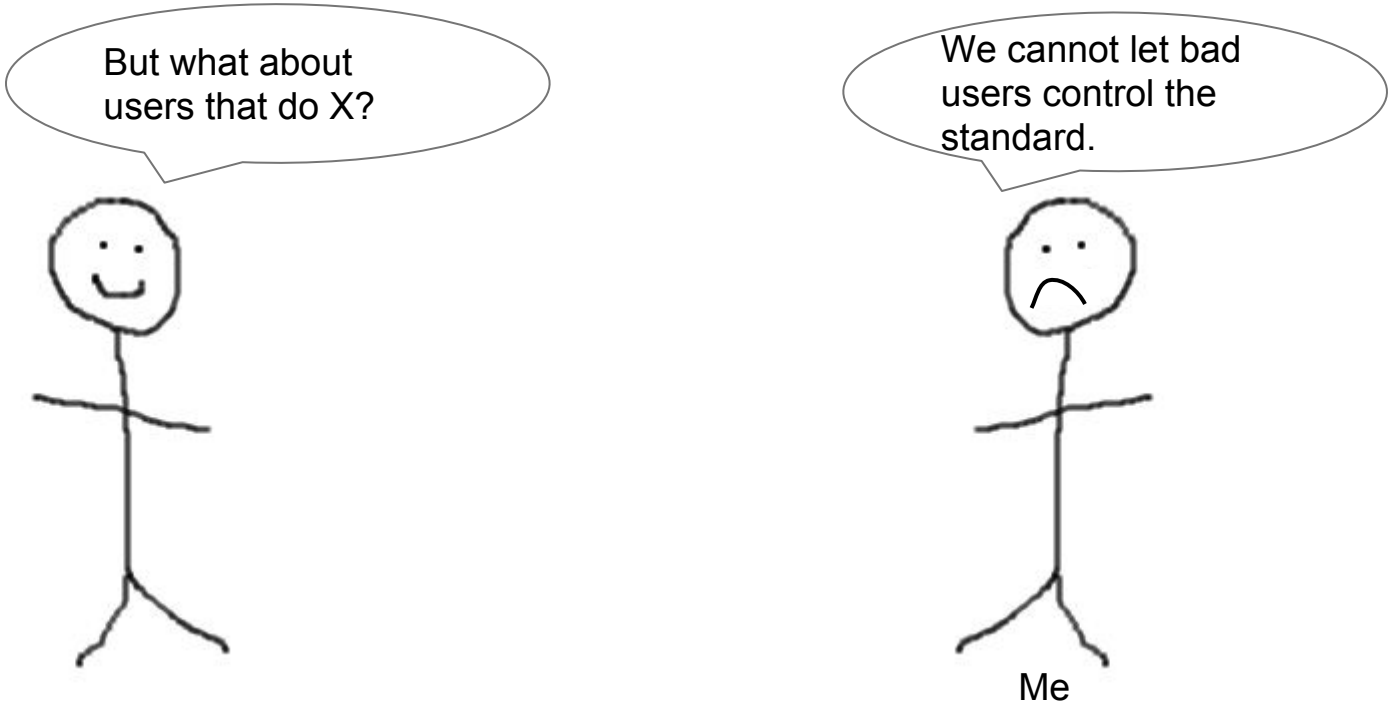
[isocpp.org SD-8]
[wg21.link/P0921r0]

SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- Add new names to namespace std
- Add new member functions to types in namespace std
- Add new overloads to existing functions
- Add new default arguments to functions and templates
- Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc).
- Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.
  a. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

SD-8 - isocpp.org



SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- Add new names to namespace std
- Add new member functions to types in namespace std
- Add new overloads to existing functions
- Add new default arguments to functions and templates
- Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc).
- Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.
  a. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

SD-8 - Add new names to std

SD-8 - Add new names to std

You

```
namespace std {
// At FooCorp, we only use optional strings.
// Pre-adopt that from the standard.
class optional {
   ...
};
}  // namespace std
```

Us

“Hey, lets provide a std::optional<T>.”

SD-8 - Add new names to std

You

```
using namespace std;

struct optional { … };

void f() {
  optional o;
}
```

Us

“Hey, lets provide a std::optional<T>.”

SD-8 - Add new names to std

You

```
using std::vector;

struct optional { … };

void f() {
  optional o;
}
```

Us

“Hey, lets provide a std::optional<T>.”

SD-8 - Add new names to std

You

```
namespace libs {
using namespace std;
struct optional {};
}
using namespace libs;

void f() {
  optional o;
}
```

Us

“Hey, lets provide a std::optional<T>.”

SD-8: Add new names to std

```
#include <string>

struct string {};
namespace foo {
using namespace std;
void f() {
string s;
}
}
```

SD-8 - Add new names to std

```
namespace libs {
bool contains(std::string_view needle,
              std::string_view haystack);

void is_polite(std::string_view haystack) {
  assert(contains(haystack, "please"));
}
}
```

SD-8 - Add new names to std

You

```
namespace libs {
bool contains(std::string_view needle,
              std::string_view haystack);

void is_polite(std::string_view haystack) {
  assert(contains(haystack, "please"));
}
}
```

Us

“Hey, lets provide a std::contains(sv1, sv2).”

SD-8 - Add new names to std

You

```
namespace libs {
bool contains(std::string_view needle,
              std::string_view haystack);

void is_polite(convert_to_sv haystack) {
  assert(contains(haystack, "please"));
}
}
```

Us

“Hey, lets provide a std::contains(str, substr).”

Side Note: Argument Dependent Lookup (ADL)

When calling an unqualified function, form overload set from:

- All enclosing scopes
- The “associated namespaces” of all arguments and template parameters.

The rules are intricate.

SD-8 - Add new names to std

```
namespace libs {
bool contains(std::string_view needle,
              std::string_view haystack);

void is_polite(std::string_view haystack) {
  assert(contains(haystack, "please"));
}
}
```

SD-8 - Add new names to std

You shouldn’t make unqualified `snake_case` function calls involving standard types

SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- **Add new names to namespace std**
- Add new member functions to types in namespace std
- Add new overloads to existing functions
- Add new default arguments to functions and templates
- Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc).
- Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.
    a. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- **Add new names to namespace std**
  a. Thou shalt not add new names to namespace std
  b. Thou shalt not add a `using namespace std` (using directive)
  c. Thou shalt not make unqualified snake_case calls to any function that accepts a type from std.

SD-8 - Add new methods to std types

Be careful applying the “detection idiom” to standard types.

SD-8 - Add new methods to std types

Be careful applying the “detection idiom” to standard types.

(Don’t metaprogram against std.)

SD-8 - Add new overloads

You

```
void f() {
    auto func_ptr = &std::vector<int>::size;
}
```

Us

“Hey, lets provide a std::vector<T>::size(tag) overload.”

SD-8 - Add new overloads

You

```
void f() {
  std::function my_isalpha = std::iswalpha;
}
```

Us

“Hey, lets provide a std::iswalpha overload.”

SD-8 - Add new overloads

Us (P0798)

```
std::optional<size_t> s =
    opt_string.map(
      &std::string::size);
```

Me

SD-8 - Add new overloads

You

```
std::ostream&
operator<< (std::ostream& os,
            const std::vector<int>& v) {
  for (int i = 0; i < v.size() - 1; ++i) {
    os << v[i] << ", ";
  }
  os << *v.back();
  return os;
}
```

Us

“Hey, we should make it easier to print containers.”

SD-8 - Add new overloads

Best case - Build break

```
<source>:18:15: error: use of overloaded operator '<<' is ambiguous (with operand
types 'std::ostream' (aka 'basic_ostream<char>') and 'std::vector<int>')
   std::cout << v << std::endl;
   ~~~~~~~~~ ^  ~
<source>:5:10: note: candidate function
ostream& operator<< (ostream& os, const vector<int>& v) { return os; }
         ^
<source>:8:15: note: candidate function
std::ostream& operator<< (std::ostream& os, const std::vector<int>& v) {
```

SD-8 - Add new overloads

Worst case - ODR Violation

¯_(ツ)_/¯

Side-Note: One Definition Rule (ODR)

- Everything used has to be defined at least once.
- Some things (functions, variables) must be defined **exactly** once.
- Other things (classes, templates, inline functions) may be defined more than once ...
  - So long as they are identical in each definition
  - And mean the same thing each time they are evaluated

Side-Note: One Definition Rule (ODR)

```
int x;

int x;
```

```
<source>:2:5: error: redefinition of 'x'
int x;
    ^
<source>:1:5: note: previous definition is here
int x;
    ^
```

Side-Note: One Definition Rule (ODR)

```
extern int x;

int x;
```

Side-Note: One Definition Rule (ODR)

foo.h

```
extern const int a;

inline bool IsGood(int b) {
  return a == b;
}
```

foo.cc

```
const int a = 42;
```

bar.cc

```
#include "foo.h"

const int a = 17;
```

Side-Note: One Definition Rule (ODR)

foo.h

```
extern const int a;

inline bool IsGood(int b) {
  return 42 == b;
}
```

foo.cc

bar.cc

```
#include "foo.h"

const int a = 17;
```

Side-Note: One Definition Rule (ODR)

You

```
std::ostream&
operator<< (std::ostream& os,
            const std::vector<int>& v) {
  for (int i = 0; i < v.size() - 1; ++i) {
    os << v[i] << ", ";
  }
  os << *v.back();
  return os;
}
```

Some library

```
std::ostream&
operator<< (std::ostream& os,
            const std::vector<int>& v) {
  for (int i = 0; i < v.size() - 1; ++i)
{
    os << v[i] << " ";
  }
  os << *v.back();
  return os;
}
```

SD-8 - Add new overloads

You

```
std::ostream&
operator<< (std::ostream& os,
            const std::vector<int>& v) {
  for (int i = 0; i < v.size() - 1; ++i) {
    os << v[i] << ", ";
  }
  os << *v.back();
  return os;
}
```

Us

“Hey, we should make it easier to print containers.”

SD-8 - Add new overloads

```
namespace mine {
void Print(const std::vector<int>&);
void Print(const std::vector<int>&,
           std::ostream&);
}
```

SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- Add new names to namespace std
- Add new member functions to types in namespace std
- **Add new overloads to existing functions**
- Add new default arguments to functions and templates
- Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc).
- Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.
  a. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

SD-8 - isocpp.org

Add new overloads to existing functions

- Don’t take the address of functions/member functions in `std`.
- Don’t define/specialize anything for standard types
  a. iostreams
  b. swap

SD-8 - Add default arguments

Similar to the previous:

- Function pointers to things in `std` are bad

Also:

- Don’t forward declare things from `std`.

SD-8 - Add default arguments

```
namespace std {

template <typename T, class Allocator>

class vector;

}


void f(const std::vector<int>& v) {}
```

SD-8 - Add default arguments

```
<source>:5:1: error: too few template parameters in template
redeclaration
template <typename T, typename Allocator>
```

SD-8 - Change Return Types

We may sometimes change return types.

SD-8 - isocpp.org

Primarily, the standard reserves the right to:

- Add new names to namespace std
- Add new member functions to types in namespace std
- Add new overloads to existing functions
- Add new default arguments to functions and templates
- **Change return-types of functions in compatible ways (void to anything, numeric types in a widening fashion, etc).**
- Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.
  a. Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

SD-8 - isocpp.org

Change return-types of functions in compatible ways

- Don’t metaprogram against the standard library
- Don’t write things like this:

```
void MySwap(int& a, int& b) {

  return std::swap(a, b);

}
```

SD-8 - Change how things are implemented

Make changes to existing interfaces in a fashion that will be backward compatible, if those interfaces are solely used to instantiate types and invoke functions.

Implementation details (the primary name of a type, the implementation details for a function callable) may not be depended upon.

Likely changes to SD-8

For types that have both copy and move, don’t rely on how many copies/moves happen.

Likely changes to SD-8

For types that have both copy and move, don’t rely on how many copies/moves happen.

(Don’t make types where move is more expensive than copy and complain to us.)

But what about Hyrum’s Law?

With a sufficient number of users of an API,

it does not matter what you promise in the contract:

all observable behaviors of your system

will be depended on by somebody.

But what about Hyrum’s Law?

vs

Future

Improved Tooling

sd-8 static-analysis warnings

tool-assisted upgrades

Future

Improve Clarity about Compatibility
(It’s a two-way street)