`noexcept` Prevents Library Validation

Document: N3248=11-0018
Date: 2011-02-28
Authors: Alisdair Meredith (ameredith1@bloomberg.net)
John Lakos (jlakos@bloomberg.net)

Abstract

The `noexcept` language facility was added at the Pittsburg meeting immediately prior to the FCD to solve some very specific problems with move semantics. This new facility also addresses a long-standing desire for many libraries to flag which functions can and cannot throw exceptions in general, opening up optimization opportunities.

The Library Working Group is now looking for a metric to decide when it is appropriate to apply the `noexcept` facility, and when to be conservative and say nothing. After spending some time analyzing the problem, the authors have concluded that the current specification for `noexcept` greatly restricts the number of places it can be used safely in a library specification such as (but not limited to) the standard library.

In this paper we propose a strict set of criteria to test before the Library Working Group should mark a function as `noexcept`. We further propose either lifting the requirement that throwing exceptions from a `noexcept` function must terminate a program (in favor of general undefined behavior), or adopting additional criteria that severely restrict the use of `noexcept` in the standard library.

Is this in scope?

At this stage of the standards process, we can respond only to issues raised by comments on the FCD ballot. The Core part of this proposal is comment CH-10, which was rejected at Rappersweil. The meeting wiki records no more than a desire not to re-open a discussion from the previous meeting before closing as NAD. Meanwhile, the library working group are responding to several comments demanding the library adopt the new noexcept facility. Feedback from this effort suggests Core may want to revisit CH-10.

What is the problem?

In order to form a rational guideline on how to apply the new feature in the library, we need to understand what benefits it offers, and what the risks of using, not using, or abusing the feature would be.

What are the benefits offered by `noexcept`?

`noexcept` is two features bound up in the same keyword, designed to preserve the user-code optimizations that rely on the principle that move operations do not throw. The basic problem, which has been well documented by David Abrahams, is that there is a category of C++ objects that do not implicitly provide move constructors when compiled with a C++0x compiler, and whose copy constructor (used in place of the move) is likely to throw. If this "move" construction throws, then program invariants may break. Notably, many library operations trying to deliver the strong exception safety guarantee rely on non-throwing moves.

The solution adopted at Pittsburg is to add a new operator, and a new form of exception specification. The operator simply checks an arbitrary expression to see that all observable operations are either built into the language (and non-throwing), or annotated with the new `noexcept` keyword to indicate that the associated operation cannot throw.

This solution also opens up a long desired optimization for compilers to exploit, where marking a function that cannot throw allows the suppression of code to handle exceptional stack unwinding. This option is appealing enough that the Library Working Group are expending effort to track down all reasonable candidate functions to mark up with the new syntax.

What are the risks associated with `noexcept`?

Unfortunately the feature is so new that there is very little field experience to develop a coherent set of guidelines. The risk from overly aggressive use of `noexcept` specifications is that programs with hidden terminate calls are produced. The risk of under-specifying `noexcept` specifications is that they become difficult to add in a later revision of the standard, as the `noexcept` operator becomes an observable part of the ABI.

What are the implications?

Functions marked `noexcept` are difficult to test

When a function is marked with `noexcept` it becomes impossible to flag test failures, notably in test drivers, by throwing an exception. A common example would be code that validates preconditions on entry to a function:

```
T& std::vector<T>::front() noexcept {
   assert(!this->empty());
   return *this->data();
}
```

When validating such defensive checks from a test driver, a reasonable approach is to register an assert-handler that throws a well-defined precondition-violated exception, which the test driver catches to ensure that the appropriate `assert`s are indeed in place.

```
struct assert_record {
   const char * expression;
   const char * filename;
   unsigned int line_number;
};

void assert_failed(const char *e, const char *f, unsigned n) {
   throw assert_record{e, f, n};
};

bool testFront() {
   register_assert_handler(&assert_failed); // see below
   std::vector<int> v;
   if(!v.empty()) {
      return false;
   }
   try {
      int x = v.front();
      return false;
   }
   catch(assert_failed const &ex) {
   }
   v.push_back(0);
   if(v.empty()) {
      return false;
   }
   try {
      int x = v.front();
      if(0 != x) {
         return false;
      }
   }
   catch(assert_failed const &ex) {
      return false;
   }
   return true;
}
```

Now we might argue that calling the function out-of-contract, when the `vector` is `empty`, is undefined behavior so we should not expect any guarantees. The problem is

that undefined behavior is being specified by the library; to the compiler, this code is perfectly well defined and, if `assert` throws an exception, the program must terminate in a well-specified manner, thwarting the test driver.

Note that the issue here is not that we are using assertions to find bugs in our own library implementations, but rather in user code that incorrectly calls into our library. If we remove the ability to test these defensive assertions, we may get them wrong, and thus put our users at risk for committing far more serious errors than propagating an unexpected exception.

Also note that there are several well-known "safe" STL implementations that detect use of invalid iterators and other precondition violations, which will be broken by this rule unless we are extremely conservative in where we adopt `noexcept` in the standard library.

What Is Required To Support Testing?

In order to test defensive assertions without crashing the test driver, we need the following facilities:

i) The ability to register an "assertion handler" for the assertion macro.
   As we are testing our own library, we are responsible for supplying the assert macro. It is simple to support registration of an assertion handler within a facility we control.

ii) The ability for the handler to return control back to the test driver.
   Once an assertion has triggered, it is important to return control to the test-case without further execution of the code under test. Ideally, the return path will also contain context information about the failing assertion, such as the `__function__`, `__FILE__` and `__LINE__` values used by the standard `assert` macro.

iii) Test drivers must handle unexpected assertions
   After returning to the test-case, the test driver must be able to report errors with both unexpected or missing assertions. This aspect may involve testing the additional context information return in (ii).

The simplest mechanism to achieve this is to throw an exception that captures the information about the assertion, and is unique to the test framework (i.e., does not derive from `std::exception` or any similar base class.) This technique, which is used widely in our current test drivers, will clearly not be compatible with additional exception specifications when migrating to C++0x.

What Workarounds Have We Tried?

We have tried a few things to work around this limitation of the language:

i/ Conditionally define `noexcept` as a macro

This macro-centric approach fundamentally cannot work, as there are multiple form of `noexcept` in the language, some taking either no argument, a constant boolean expression, or an arbitrary expression when invoked as an operator. It is not possible to overload a single macro for all choices.

ii/ Consistently use library-specific macros instead of the `noexcept` keyword

Choosing to use macros to replace keywords impairs readability of the code. That might be acceptable if this were limited in scope to the standard library, but this is an issue every library author will face.

The real problem is that these macros distort regular code-paths, by disabling `noexcept` exception specifications in test builds. For example, move constructors will always select the potentially-throwing code-path.

iii/ Install a handler using `setjmp`/`longjmp` to return control

The final experiment we tried used `setjmp`/`longjmp` instead of exceptions to return control to the test driver. This successfully defeated the exception specification, at the expense of entering undefined behavior (from the language) and omitting stack unwinding on the compilers we tried. Future compilers may even be helpful enough to catch this trick and abort the program for us, unless this testing technique is explicitly supported.

What Are We Proposing

In order to support effective testing, compilers should offer two 'modes' for handling `noexcept` violations.

i/ a "production" mode, the default, which guarantees to terminate the process rather than allow an exception to propagate past a `noexcept` exception specification.

ii/ a "testing" mode, which performs regular stack unwinding if an exception propagates beyond a `noexcept` exception specification. This would imply disabling any optimizations based on static analysis of exception specifications, while ensuring that the `noexcept` operator continues to see the same result.

The Standard Does Not Explicitly Talk About Modes

This paper does not propose standard wording, as to date the committee has actively chosen not to standardize compiler switches and multiple modes. The closest we come to date is the conditional behavior of the `assert` macro, which is tied to the `NDEBUG` macro. Multiple modes are usually supported by specifying undefined behavior, unspecified behavior, or implementation-defined behavior.

It is understood that there are strong reservations about opening this part of the specification to the full consequences of undefined behavior – indeed the lack of the "production" mode guarantee would raise NO votes on the FDIS from several national bodies.

Conversely, there is a need to support a "test" mode, which may require some form of novel specification to enable an additional compilation mode without falling back on a total lack of specification, or explicitly mandating compiler switches.

How Much Of The Standard Is Affected?

A review of the wording impact points to a single paragraph that needs updating, so the wording impact should be relatively contained:

15.4 Exception specifications [except.spec]

9 Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an *exception-specification* that does not allow the exception, then,
— if the exception-specification is a *dynamic-exception-specification*, the function `std::unexpected()` is called (15.5.2),
— otherwise, the function `std::terminate()` is called (15.5.1).

The second bullet is the only place to mandate the calling of terminate in this case. There may be a couple of minor Editorial tweaks around the specification of terminate, to reflect that it would no longer be guaranteed to be called in these circumstances. Those edits would be limited to **15.5.1 [except.terminate]**.

Is There An Alternative?

The standard library is no different from any other when it comes to testing. If we cannot support a suitable testing mode, then we should be much more careful about when we choose to apply `noexcept` to standard library functions. A detailed set of guidelines, and proposed library wording, follows. Note, however, that the library wording below applies only if the "testing" mode is not available.

Narrow and Wide Contracts

To ease the following discussion, we define two terms of art, 'narrow contract' and 'wide contract'. These terms are widely used with differing informal definitions, so we will define precisely what they mean (in this document) to avoid confusion.

Wide Contracts

A wide contract for a function or operation does not specify any undefined behavior. Such a contract has no preconditions: A function with a wide contract places no additional constraints on its arguments, on any object state, nor on any external global state. Examples of functions having wide contracts would be `vector<T>::begin()` and `vector<T>::at(size_type)`. Examples of functions not having a wide contract would be `vector<T>::front()` and `vector<T>::operator[](size_type)`.

Narrow Contracts

A narrow contract is a contract which is not wide. Narrow contracts for a functions or operations result in undefined behavior when called in a manner that violates the documented contract. Such a contract specifies at least one precondition involving its arguments, object state, or some external global state, such as the initialization of a static object. Good examples of standard functions with narrow contracts are `vector<T>::front()` and `vector<T>::operator[](size_type)`.

Recommendations

Core Recommendation

The key recommendation is that violating a `noexcept` specification should be either undefined behavior, or implementation defined behavior. While it is perfectly reasonable for the default behavior to terminate the application, there are real business-driven reasons to support additional build modes.

Note that while the standard does not talk about build modes and switches, undefined behavior and implementation defined behavior are the terms that give us freedom to add such modes. This recommendation is not trying to mandate a compiler-switch in the standard (something that we do not do) but rather to provide the freedom for vendors to supply such a switch if they choose.

Library Recommendations

The remaining recommendations of this document are broken into two groups, and focussed on the Library. All changes are relative to the post-Batavia Working Paper, N3225.

The first group recommends a set of guidelines that should apply regardless of other outcomes.

The second group is conditional, consisting of a simple and permissive set of library guidelines if the critical language rule is relaxed, or a more detailed and far more restrictive set of library guidelines to follow if the core language is not changed.

Basic Recommendations

- No library destructor should throw.
- The library should be required to support only those user types whose destructors cannot throw. Note that this waiver is currently handled by the Destructible requirements of clause 20, [destructible].

- Requirements on allocators should include that no de-allocation function throws. Note that this restriction is already part of the allocator requirements - Table 44.
- All library move-constructor and move-assignment operators should be marked with a `noexcept` specification qualifying when they might throw.
- All library `swap` functions should be marked with a `noexcept` specification qualifying when they might throw. Note that `swap` may have a narrow contract, for example attempting to `swap` two containers having allocators that do not compare equal would result in undefined behavior.
- Each library function having a **wide** contract, that clearly cannot throw, should be marked as unconditionally `noexcept`.

Preferred additional recommendation

If the core language can be amended to support a testing mode, we recommend the following guideline:

- Each operation whose behavior is such that it clearly cannot throw, when called with arguments satisfying function preconditions (and its own object state invariants), should be marked as unconditionally `noexcept`.

Alternative additional recommendation

If the core language is not amended to support a testing mode, we recommend the following guideline:

- Remove `noexcept` specifications from each library function having a **narrow** contract, typically (but not always) indicated by the presence of a Requirements: clause.

Library Review

Taking account of the recommendations above, what changes would we make to the library? The following review considers the library as currently amended by the original sequence of papers proposing `noexcept` annotations in Batavia.

Note that this review assumes that the core language is not amended to support testing. If the core language is so amended, then the following should be discarded.

Clause 17

No changes

Clause 18

No changes

Most of the applications of `noexcept` are adopting the new syntax in preference to the old empty throw specification. The remaining examples, such as `terminate` and `exit`, are either already defined to terminate the application if an exception escapes, or to have a wide interface that returns failure results by some means other than an exception, such as `at_exit`.

Clause 19

No changes

The `noexcept` feature is used widely in the new system error facility, but in each case is applied to a function with a wide contract. Two functions worth calling out are `make_error_code` and `make_error_condition`, both of which take an `errc` argument and cast it to an `int`. This is required to work, as `errc` is an `enum class` with an (implied) base type of `int`.

Clause 20

<utility>

No changes. The conditional `nothrow` on `swap` restricts the guarantee to the subset where we can syntactically prove the contract will hold, and similarly for the move-constructor/assignment operator of `pair`, and `make_pair`. All other usages are with wide contracts, which cannot fail.

<tuple>

No changes. The signatures annotated with `nothrow` in the tuple header are constrained in the cases where the contract is narrow, as with `pair`. The requires clause on `get` will produce a compile-time error if violated, safely allowing the unconditional `noexcept` qualification for the valid cases.

<bitset>

This header offers two problems. The first is how to indicate a lack of resources. If the user tries to create an object of `bitset<numeric_limits<size_t>::max()>` then we are likely to see a runtime failure of some kind - but the only way to fail a constructor is by throwing an exception, which is no longer permitted in this case. As a pragmatic decision, we might deem such resource failures as running out of stack memory to be undefined behavior beyond our control, but we would also be disallowing systems that might provide the ability to check their available stack space and throw, or move to a dynamic allocation scheme for large `bitset`s.

The other concern in this class is the `operator[]` overload, which now prohibits checked implementations. Unless the core language behavior can be weakened, this operator should not be declared `noexcept`.

<function>

no change to the `refererence_wrapper` class and factory functions, as these are a classic wide interface. Note there is an outstanding question of adding `noexcept` to the move constructor and move-assignment operator for this class. Such a change would impact the implementation's ability to choose to apply the small-object optimization, but has no impact on the width of the contract. This paper supports making such a change, but does not propose it.

`mem_fn` is troublesome, as the standard makes no claim as to what happens if we pass `nullptr` to any of these factory functions. I believe this should be undefined behavior, in which case we should strike the `noexcept` annotation on each signature. An alternative interpretation would be that we should get a valid object returned, giving us a wide contract, but that it is undefined behavior to invoke the function-call operator. For the sake of this paper, we are assuming the former, and will open a Library Issue if needed after this paper is reviewed.

`function` again provides us with a minimal set of `noexcept` annotations which apply only to cases with wide contracts that truly cannot throw. There is an ongoing discussion as to whether the move constructor should be marked `noexcept` which is beyond this paper. It is certainly possible to implement that contract, but it may require a change to existing implementations already in the field, as target-objects with potentially-throwing move constructors would not be allowed to take advantage of the small-object optimization (when the initial `function` object is constructed).

<memory>

There are a number of optimistic uses of `noexcept` in <memory> that should be scaled back, notably for any deallocation function taking a custom pointer, or with a requirement that the pointer was initially allocated from a matching allocator.

The `raw_storage_iterator` presents an interesting case. As an iterator adaptor, it holds an iterator of unknown type '`T`' and so is subject to any issues that might be reported when invoking that object's operations, such as overflow detection in `operator++`. We note that it should be perfectly valid to pass a null pointer to the constructor of this class though, as it is reasonable to denote an empty range with a pair of such values. However, if the underlying object has a copy constructor that throws, it is still not possible to provide a `noexcept` guarantee for the constructor, so for expedience we strike the annotation here as well. Note that it would be possible to provide a conditional `noexcept` in this case.

The `return_temporary_buffer` function needs to be able to validate that the returned pointer does indeed reference a temporary buffer allocated by a `get_temporary_buffer` call.

Most applications of `noexcept` for unique_ptr and shared_ptr are on functions with wide contracts. However, there are preconditions on the atomic access functions, so these should lose the specification.

There is an important issue with allocators and allocator adaptors, that there must be a requirement that the `deallocate` function does not throw (Allocator requirements), but the function should not be annotated with `noexcept` in order to allow implementations diagnosing bad pointer values that are not managed by this allocator object.

The pointer-safety (garbage collection) APIs need the ability to validate that passed pointers are not null, and in the '`undeclare`' cases that the passed pointer was previously registered with a matching '`declare`' call.

The `align` function has a narrow contract requiring that only certain valid alignments be requested. A better solution that removing the `noexcept` specification would be to widen the contract to return a null pointer in such cases, just as when there is insufficient space available to honor the request.

Clause 21(basic string)

The `char_traits` functions that work with characters are fine, but those taking `charT*` have narrow contracts that may want to validate the passed string(s).

There are a number of narrow contracts marked with `noexcept` in `basic_string`: `operator[]` should have freedom to validate the passed index. `pop_back` requires a non-empty string. The `find` family of functions taking a `const charT*`, and similarly `compare` and the comparison operators, need freedom to defend against null pointers.

Clause 22

No changes. Locale applies `noexcept` only to cases that were previously marked with an empty throw specification.

Clause 23

<array>

No changes for `array` (tuple interface)

<deque>

No changes. `noexcept` has not yet been applied to this container.

<forward_list> / <list>

`list`/`forward_list` : `erase` and `splice` take iterators that must be valid. `pop_front`/`pop_back` require non-empty lists. clear and reverse have wide contracts, so are fine.

<vector>

No changes for vector, as only the `data` function has been updated, and it has a wide contract.

Clause 24

`iostreambuf_iterator` constructors unfortunately must preserve the existing `throw` specification from the 2003 standard, although they would violate our guidelines.

Clause 26

<random>

No changes for the one usage in the random number facility.

<valarray>

`valarray` will require further analysis by someone familiar with the implementation details of this component. This paper makes no recommendations.

Clause 27

<ios>

`basic_ios::set_rdbuf` takes a pointer which may not be valid, narrowing the contract.

Clause 28

<regex>

No changes. The only use of `noexcept` with regular expressions is for move operations that must succeed, and `swap`.

Clause 30

<thread>

No changes for `thread` or `thread::id`.

<mutex>

`mutex::unlock` and `lock_guard` have preconditions that the current thread owns the mutex. Similarly but in more detail for unique_lock

<condition_variable>

No changes for `condition_variable`, or `condition_variable_any`.

<future>

No changes for `future_error`, `promise`, `future`, `shared_future` or `packaged_task`.

Proposed Library Changes

20.5 [template.bitset]

```
#include <string>
#include <iosfwd> // for istream, ostream
namespace std {
  template <size_t N> class bitset;

  // 20.5.4 bitset operators:
  template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
  template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
  template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
  template <class charT, class traits, size_t N>
    basic_istream<charT, traits>&
      operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
  template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```
namespace std {
  template<size_t N>
  class bitset {
  public:
    // ...
    // 20.5.1 constructors:
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;

    // ...
    bitset<N> operator~() const noexcept;
```

```
  // ...
  // element access:
  constexpr bool operator[](size_t pos) const noexcept;
  reference operator[](size_t pos) noexcept;
 };
}
```

20.5.1 [bitset.cons]

onstexpr bitset() noexcept;

1 *Effects*: Constructs an object of class bitset<N>, initializing all bits to zero.

constexpr bitset(unsigned long long val) noexcept;

2 *Effects*: Constructs an object of class bitset<N>, initializing the first M bit positions to the corresponding bit values in val. M is the smaller of N and the number of bits in the value representation (3.9) of unsigned long long. If M<N, the remaining bit positions are initialized to zero.

20.5.2 [bitset.members]

bitset<N> operator~() const noexcept;

23 *Effects*: Constructs an object x of class bitset<N> and initializes it with *this.

24 *Returns*: x.flip().

constexpr bool operator[](size_t pos) noexcept;

49 *Requires*: pos shall be valid.
Throws: nothing.

50 *Returns*: true if the bit at position pos in *this has the value one, otherwise false.

bitset<N>::reference operator[](size_t pos) noexcept;

51 *Requires*: pos shall be valid.
Throws: nothing.

20.5.4 [bitset.operators]

bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

1 *Returns*: bitset<N>(lhs) &= rhs.

bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

2 *Returns*: bitset<N>(lhs) |= rhs.

bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

3 *Returns*: bitset<N>(lhs) ^= rhs.

20.8 [function objects]

2 Header <functional> synopsis

```
// ....
// 20.8.13, member function adaptors:
template<class R, class T> unspecified mem_fn(R T::*) noexcept;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...)) noexcept;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) const) noexcept;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) volatile) noexcept;
```

template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &&) ~~noexcept~~;

20.8.13 [func.memfn]

template<class R, class T> *unspecified* mem_fn(R T::*) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...)) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &&) ~~noexcept~~;

20.9 [memory]

Header <memory> synopsis

// 20.9.10.5, shared_ptr atomic access:
template<class T>
bool atomic_is_lock_free(const shared_ptr<T>* p) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_load(const shared_ptr<T>* p) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo) ~~noexcept~~;
template<class T> void atomic_store(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
template<class T> void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
memory_order mo) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_weak(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_strong(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_weak_explicit(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_strong_explicit(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;

// 20.9.11, Pointer safety
enum class pointer_safety { relaxed, preferred, strict }; void declare_reachable(void *p); template <class T> T
*undeclare_reachable(T *p) ~~noexcept~~;
void declare_no_pointers(char *p, size_t n) ~~noexcept~~;
void undeclare_no_pointers(char *p, size_t n) ~~noexcept~~;
pointer_safety get_pointer_safety() noexcept;

20.9.4 [allocator.traits]

```
namespace std {
  template <class Alloc> struct allocator_traits {
    // ...
    static void deallocate(Alloc& a, pointer p, size_type n) noexcept;
  };
}
```

20.9.4.2 [allocator.traits.members]

```
        static void deallocate(Alloc& a, pointer p, size_type n) noexcept;
```

3 *Effects*: calls a.deallocate(p, n).
Throws: nothing.

20.9.5 [default.allocator]

```
namespace std {
  // ...

  template <class Alloc>class allocator {
    // ...
    void deallocate(pointer p, size_type n) noexcept;
  };
}
```

20.9.6 [storage.iterator]

1 raw_storage_iterator is provided to enable algorithms to store their results into uninitialized memory. The formal template parameter OutputIterator is required to have its operator* return an object for which operator& is defined and returns a pointer to T, and is also required to satisfy the requirements of an output iterator (24.2.4).

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
    : public iterator<output_iterator_tag,void,void,void,void> {
  public:
    explicit raw_storage_iterator(OutputIterator x) noexcept;

    raw_storage_iterator<OutputIterator,T>& operator*() noexcept;
    raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
    raw_storage_iterator<OutputIterator,T>& operator++() noexcept;
    raw_storage_iterator<OutputIterator,T> operator++(int) noexcept;
  };
}
```

```
        explicit raw_storage_iterator(OutputIterator x) noexcept;
```

2 *Effects*: Initializes the iterator to point to the same value to which x points.

```
        raw_storage_iterator<OutputIterator,T>& operator*() noexcept;
```

3 *Returns*: *this

```
        raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
```

4 *Effects*: Constructs a value from element at the location to which the iterator points.

5 *Returns*: A reference to the iterator.

raw_storage_iterator<OutputIterator,T>& operator++() ~~noexcept~~;

6 *Effects*: Pre-increment: advances the iterator and returns a reference to the updated iterator.

raw_storage_iterator<OutputIterator,T> operator++(int) ~~noexcept~~;

7 *Effects*: Post-increment: advances the iterator and returns the old value of the iterator.

20.9.7 [temporary.buffer]

template <class T> void return_temporary_buffer(T* p) ~~noexcept~~;

3 *Effects*: Deallocates the buffer to which p points.

4 *Requires*: The buffer shall have been previously allocated by get_temporary_buffer.

20.9.10.5 [util.smartptr.shared.atomic]

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 29.3.

template<class T> bool atomic_is_lock_free(const shared_ptr<T>* p) ~~noexcept~~;
Requires: p shall not be null.
Returns: true if atomic access to *p is lock-free, false otherwise.
Throws: nothing.

template<class T>
shared_ptr<T> atomic_load(const shared_ptr<T>* p) ~~noexcept~~;
Requires: p shall not be null.
Returns: atomic_load_explicit(p, memory_order_seq_cst).
Throws: nothing.

shared_ptr<T>
atomic_load_explicit(const shared_ptr<T>* p, memory_order mo) ~~noexcept~~;
Requires: p shall not be null. Requires: mo shall not be memory_order_release or memory_order_acq_rel.
Returns: *p.
Throws: nothing.

template<class T>
void atomic_store(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
Requires: p shall not be null.
Effects: atomic_store_explicit(p, r, memory_order_seq_cst).
Throws: nothing.

template<class T>
void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
Requires: p shall not be null.
Requires: mo shall not be memory_order_acquire or memory_order_acq_rel.
Effects: p->swap(r).
Throws: nothing.

template<class T>
shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
Requires: p shall not be null.
Returns: atomic_exchange_explicit(p, r, memory_order_seq_cst).

Throws: nothing.

template<class T> shared_ptr<T>
  atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
Requires: p shall not be null.
Effects: p->swap(r).
Returns: the previous value of *p.
Throws: nothing.

template<class T>
  bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
Requires: p shall not be null.
Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_-order_seq_cst).
Throws: nothing.

template<class T>
  bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
Returns: atomic_compare_exchange_strong_explicit(p, v, w, memory_order_seq_cst, memory_-order_seq_cst).

template<class T>
  bool atomic_compare_exchange_weak_explicit(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
memory_order success, memory_order failure) ~~noexcept~~;
template<class T>
  bool atomic_compare_exchange_strong_explicit(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
memory_order success, memory_order failure) ~~noexcept~~;
Requires: p shall not be null. Requires: failure shall not be memory_order_release, memory_order_acq_rel, or stronger than success.
Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to the value of failure.
Returns: true if *p was equivalent to *v, false otherwise.
Throws: nothing.
Remarks: two shared_ptr objects are equivalent if they store the same pointer value and share ownership.
Remarks: the weak forms may fail spuriously. See 29.6.

20.9.11 [util.dynamic.safety]

A complete object is declared reachable while the number of calls to declare_reachable with an argument referencing the object exceeds the number of calls to undeclare_reachable with an argument referencing the object.

void declare_reachable(void *p);
Requires: p shall be a safely-derived pointer (3.7.4.3) or a null pointer value.
Effects: If p is not null, the complete object referenced by p is subsequently declared reachable (3.7.4.3).
Throws: May throw std::bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

template <class T>
  T *undeclare_reachable(T *p) ~~noexcept~~;
Requires: If p is not null, the complete object referenced by p shall have been previously declared reachable, and shall be live (3.8) from the time of the call until the last undeclare_reachable(p) call on the object.
Returns: a safely derived copy of p which shall compare equal to p.
Throws: nothing.

[Note: It is expected that calls to declare_reachable(p) will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to undeclare_reachable(p) is encountered. Long running programs should arrange that calls are matched. — end note]

void declare_no_pointers(char *p, size_t n) ~~noexcept~~;
Requires: No bytes in the specified range have been previously registered with declare_no_pointers(). If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding undeclare_no_pointers() call. [Note: In a garbage-collecting implementation, the fact that a region in an object is registered with declare_no_- pointers() should not prevent the object from being collected. — end note]

Effects: The n bytes starting at p no longer contain traceable pointer locations, independent of their type. Hence pointers located there may not be dereferenced if the object they point to was created by global operator new and not previously declared reachable. [Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — end note]

Throws: nothing.
[Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — end note]

void undeclare_no_pointers(char *p, size_t n) ~~noexcept~~;
Requires: The same range must previously have been passed to declare_no_pointers(). Effects: Unregisters a range registered with declare_no_pointers() for destruction. It must be called before the lifetime of the object ends.
Throws: nothing.

pointer_safety get_pointer_safety() noexcept;
Returns: pointer_safety::strict if the implementation has strict pointer safety (3.7.4.3). It is im- plementation defined whether get_pointer_safety returns pointer_safety::relaxed or pointer_- safety::preferred if the implementation has relaxed pointer safety.

20.9.12 [ptr.align]

void *align(std::size_t alignment, std::size_t size, void *&ptr, std::size_t& space) ~~noexcept~~;

20.10 [allocator.adaptor]

```
namespace std { template <class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor : public OuterAlloc {
public:
   void deallocate(pointer p, size_type n) noexcept;
```

21.2.3.1 [char.traits.specializations.char]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char          char_type;
    typedef int  int_type;
    typedef streamoff  off_type;
    typedef streampospos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n) noexcept;
    static size_t length(const char_type* s) noexcept;
```

```
  static const char_type* find(const char_type* s, size_t n, const char_type& a) noexcept;
  static char_type* move(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* copy(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* assign(char_type* s, size_t n, char_type a) noexcept;

  static constexpr int_type not_eof(int_type c) noexcept;
  static constexpr char_type to_char_type(int_type c) noexcept;
  static constexpr int_type to_int_type(char_type c) noexcept;
  static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
  static constexpr int_type eof() noexcept;
};
}
```

21.2.3.2 [char.traits.specializations.char16_t]

```
namespace std {
 template<> struct char_traits<char> {
  typedef char         char_type;
  typedef uint_least16_t     int_type;
  typedef streamoff  off_type;
  typedef streampospos_type;
  typedef mbstate_t state_type;

  static void assign(char_type& c1, const char_type& c2) noexcept;
  static constexpr bool eq(char_type c1, char_type c2) noexcept;
  static constexpr bool lt(char_type c1, char_type c2) noexcept;

  static int compare(const char_type* s1, const char_type* s2, size_t n) noexcept;
  static size_t length(const char_type* s) noexcept;
  static const char_type* find(const char_type* s, size_t n, const char_type& a) noexcept;
  static char_type* move(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* copy(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* assign(char_type* s, size_t n, char_type a) noexcept;

  static constexpr int_type not_eof(int_type c) noexcept;
  static constexpr char_type to_char_type(int_type c) noexcept;
  static constexpr int_type to_int_type(char_type c) noexcept;
  static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
  static constexpr int_type eof() noexcept;
};
}
```

21.2.3.3 [char.traits.specializations.char32_t]

```
namespace std {
 template<> struct char_traits<char> {
  typedef char         char_type;
  typedef uint_least32_t     int_type;
  typedef streamoff  off_type;
  typedef streampospos_type;
  typedef mbstate_t state_type;

  static void assign(char_type& c1, const char_type& c2) noexcept;
  static constexpr bool eq(char_type c1, char_type c2) noexcept;
  static constexpr bool lt(char_type c1, char_type c2) noexcept;

  static int compare(const char_type* s1, const char_type* s2, size_t n) noexcept;
  static size_t length(const char_type* s) noexcept;
  static const char_type* find(const char_type* s, size_t n, const char_type& a) noexcept;
  static char_type* move(char_type* s1, const char_type* s2, size_t n) noexcept;
```

```
  static char_type* copy(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* assign(char_type* s, size_t n, char_type a) noexcept;

  static constexpr int_type not_eof(int_type c) noexcept;
  static constexpr char_type to_char_type(int_type c) noexcept;
  static constexpr int_type to_int_type(char_type c) noexcept;
  static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
  static constexpr int_type eof() noexcept;
};
}
```

21.2.3.4 [char.traits.specializations.wchar.t]

```
namespace std {
 template<> struct char_traits<char> {
  typedef wchar_t    char_type;
  typedef wint_t     int_type;
  typedef streamoff  off_type;
  typedef streampospos_type;
  typedef mbstate_t state_type;


  static void assign(char_type& c1, const char_type& c2) noexcept;
  static constexpr bool eq(char_type c1, char_type c2) noexcept;
  static constexpr bool lt(char_type c1, char_type c2) noexcept;

  static int compare(const char_type* s1, const char_type* s2, size_t n) noexcept;
  static size_t length(const char_type* s) noexcept;
  static const char_type* find(const char_type* s, size_t n, const char_type& a) noexcept;
  static char_type* move(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* copy(char_type* s1, const char_type* s2, size_t n) noexcept;
  static char_type* assign(char_type* s, size_t n, char_type a) noexcept;

  static constexpr int_type not_eof(int_type c) noexcept;
  static constexpr char_type to_char_type(int_type c) noexcept;
  static constexpr int_type to_int_type(char_type c) noexcept;
  static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
  static constexpr int_type eof() noexcept;
};
}
```

21.3 [string.classes]

Header <string> synopsis

```
namespace std {
#include <initializer_list>  [Editorial note: move this above opening of namespace]
// 21.4, basic_string:
// ....
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
noexcept;

template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;

template<class charT, class traits, class Allocator> bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs) noexcept;
```

```
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
noexcept;

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
noexcept;

template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
noexcept;

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
noexcept;

template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,const basic_string<charT,traits,Allocator>& rhs) noexcept;

// 21.4.8.8: swap
template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs, basic_string<charT,traits,Allocator>& rhs) noexcept;
```

21.4 [basic.string]

```
namespace std {
template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT> >
class basic_string {
public:
   //...
   // 21.4.5 element access:
   const_reference    operator[](size_type pos) const noexcept;
   reference          operator[](size_type pos) noexcept;
   const_reference    at(size_type n) const;
   reference          at(size_type n);

   // 21.4.6 modifiers:
```

```
    // ...
    void pop_back() noexcept;

  //...
  void swap(basic_string& str) noexcept;

  // 21.4.7 string operations:
  const charT* c_str() const noexcept;
  const charT* data() const noexcept;
  allocator_type get_allocator() const noexcept;

  size_type find (const basic_string& str, size_type pos = 0) const noexcept;
  size_type find (const charT* s, size_type pos, size_type n) const noexcept;
  size_type find (const charT* s, size_type pos = 0) const noexcept;
  size_type find (charT c, size_type pos = 0) const noexcept;

  size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
  size_type rfind(const charT* s, size_type pos, size_type n) const noexcept;
  size_type rfind(const charT* s, size_type pos = npos) const noexcept;
  size_type rfind(charT c, size_type pos = npos) const noexcept;

  size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;
  size_type find_first_of(const charT* s, size_type pos, size_type n) const noexcept;
  size_type find_first_of(const charT* s, size_type pos = 0) const noexcept;
  size_type find_first_of(charT c, size_type pos = 0) const noexcept;

  size_type find_last_of (const basic_string& str, size_type pos = npos) const noexcept;
  size_type find_last_of (const charT* s, size_type pos, size_type n) const noexcept;
  size_type find_last_of (const charT* s, size_type pos = npos) const noexcept;
  size_type find_last_of (charT c, size_type pos = npos) const noexcept;

  size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;
  size_type find_first_not_of(const charT* s, size_type pos, size_type n) const noexcept;
  size_type find_first_not_of(const charT* s, size_type pos = 0) const noexcept;
  size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;

  size_type find_last_not_of (const basic_string& str, size_type pos = npos) const noexcept;
  size_type find_last_not_of (const charT* s, size_type pos, size_type n) const noexcept;
  size_type find_last_not_of (const charT* s, size_type pos = npos) const noexcept;
  size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;

  basic_string substr(size_type pos = 0, size_type n = npos) const;
  int compare(const basic_string& str) const noexcept;
  int compare(size_type pos1, size_type n1, const basic_string& str) const;
  int compare(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2) const;
  int compare(const charT* s) const noexcept;
  int compare(size_type pos1, size_type n1, const charT* s) const;
  int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;
 };
}
```

23.3.3 [forwardlist]

```
namespace std {
 template <class T, class Allocator = allocator<T> >
   class forward_list {
   public:
     // ...
     // 23.3.3.4 modifiers:
```

```
    // ...
    iterator erase_after(const_iterator position) noexcept;
    iterator erase_after(const_iterator position, iterator last) noexcept;

    // 23.3.3.5 forward_list operations:
    void splice_after(const_iterator position, forward_list<T,Allocator>&& x) noexcept;
    void splice_after(const_iterator position, forward_list<T,Allocator>&& x, const_iterator i) noexcept;
  };
}
```

23.3.4 [list]

```
namespace std {
  template <class T, class Allocator = allocator<T> >
    class list {
    public:
      // ...
      // 23.3.4.3 modifiers:
      template <class... Args> void emplace_front(Args&&... args);
      void pop_front() noexcept;
      template <class... Args> void emplace_back(Args&&... args);
      void push_front(const T& x);
      void push_front(T&& x);
      void push_back(const T& x);
      void push_back(T&& x);
      void pop_back() noexcept;

      // ...

      iterator erase(const_iterator position) noexcept;
      iterator erase(const_iterator position, const_iterator last) noexcept;
      void swap(list<T,Allocator>&);
      void clear() noexcept;

      // 23.3.4.4 list operations:
      void splice(const_iterator position, list<T,Allocator>& x) noexcept;
      void splice(const_iterator position, list<T,Allocator>&& x) noexcept;
      void splice(const_iterator  position, list<T,Allocator>& x, const_iterator i) noexcept;
      void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i) noexcept;
      void splice(const_iterator position, list<T,Allocator>& x, const_iterator first, const_iterator last) noexcept;
      void splice(const_iterator position, list<T,Allocator>&& x, const_iteratorfirst, const_iterator last) noexcept;
    };
}
```

27.5.4 [ios]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ios : public ios_base {
      // ....
    protected:
      basic_ios();
      void init(basic_streambuf<charT,traits>* sb);
      void move(basic_ios& rhs);
      void move(basic_ios&& rhs);
      void swap(basic_ios& rhs) noexcept;
      void set_rdbuf(basic_streambuf<charT, traits>* sb) noexcept;
    };
}
```

27.5.4.2 [basic.ios.members]

void set_rdbuf(basic_streambuf<charT, traits>* sb) ~~noexcept~~;

Requires: sb != nullptr.

Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

Postconditions: rdbuf() == sb.

Throws: Nothing.

30.4.1.2.1 [thread.mutex.class]

```
namespace std {
  class mutex {
  public:
    constexpr mutex();
    ~mutex();

    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;

    void lock(); bool try_lock() noexcept;
    void unlock() noexcept;

    typedef implementation-defined native_handle_type;  // See 30.2.3
    native_handle_type native_handle();                 // See 30.2.3
  };
}
```

30.4.1.2.2 [thread.mutex.recursive]

```
namespace std {
  class recursive_mutex {
  public:
    recursive_mutex();
    ~recursive_mutex();

    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator=(const recursive_mutex&) = delete;

    void lock();
    bool try_lock() noexcept;
    void unlock() noexcept;

    typedef implementation-defined native_handle_type;  // See 30.2.3
    native_handle_type native_handle();                 // See 30.2.3
  };
}
```

30.4.2.1 [thread.lock.guard]

```
namespace std {
  template <class Mutex>
  class lock_guard {
  public:
    typedef Mutex mutex_type;

    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t) noexcept;
    ~lock_guard();

    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;
```

```
  private:
    mutex_type& pm; // exposition only
};
}
```

30.4.2.2 [thread.lock.unique]

```
namespace std {
  template <class Mutex>
  class unique_lock {
  public:
    typedef Mutex mutex_type;

    // 30.4.2.2.1 construct/copy/destroy
    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t) noexcept;
    unique_lock(mutex_type& m, adopt_lock_t) noexcept;
    template <class Clock, class Duration>
      unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time) noexcept;
    template <class Rep, class Period>
      unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time) noexcept;
    ~unique_lock();

    unique_lock(unique_lock const&) = delete;
    unique_lock& operator=(unique_lock const&) = delete;

    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator=(unique_lock&& u) noexcept;

    // 30.4.2.2.2 locking
    void lock();
    bool try_lock();

    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

    void unlock();

    // 30.4.2.2.3 modifiers
    void swap(unique_lock& u) noexcept;
    mutex_type *release() noexcept;

    // 30.4.2.2.4 observers
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;

  private:
    mutex_type *pm;   // exposition only
    bool owns;        // exposition only
  };
}
```

Recommendations

We would like to thank the following for their helpful review comments:

Beman Dawes, Daniel Krugler, Pablo Halpern, Howard Hinnant, Bjarne Stroustrup, and Alexei Zakharov