A Vision For std2

Alisdair Meredith
Bloomberg LP

What is this talk?

- My own *personal* vision
  - Does *not* represent ISO committee positions in any way
  - Does not represent Bloomberg official positions in any way!
- A plan to get folks with their own visions talking
  - Hope to inspire a broad vision for a new library, and supporting language, from the folks best placed to drive it

Format

- Wednesday:
  Language Features for C++20 (and beyond)
- Thursday:
  Design pressures, pitfalls, and opportunities for std2
- Friday:
  “Workshop” is a feedback session to try to inspire a collective vision paper for ISO in July

Inspiration

- First attempt to constrain standard library with concepts, around a decade ago
- Failed for several reasons
    - Concepts were not ready, and becoming more complex by patching them late
    - Library insisted on complete compatibility with existing code
        - produced too many concepts
        - deprived concepts of power, by adding concepts till nothing broke
- Concern that a concept-based library would have to break backwards compatibility, or compromise itself so badly it may not be worth the effort

Why Now?

- C++17 reserved a family of namespaces for future libraries
- We have long dreamed of a future incompatible library, fixing many defects
- Concepts are coming, and that seems the likely spark to start such an overwhelming project

What is std2?

- A new library in a new namespace
  - Opportunity to ‘break’ API and ABI
- Successor to ‘std’, inspired by modern language features
- Compatible with ‘std’
  - Similar APIs/vocabulary so porting is easy
  - Expected to coexist for many years

What Should std2 Contain?

- A replacement for `std`, done right
- A (mostly) source compatible update for `std`, constrained by concepts and contracts
- Entirely new components using modern idioms, to complement `std`, not to replace it

Personal Vision

Somewhere between the first two options.
That is the focus for the rest of this talk.

- A replacement for `std`, done right
- A (mostly) source compatible update for `std`, constrained by concepts and contracts

When Should We Ship std2?

- C++20 : it is essential we start to deliver the new library (incrementally) as soon as possible
- C++23 : build up a good foundation (via TS) before adopting something solid
- When It Is ReadyTM : don’t release until we have a complete library and a coherent user experience, that may completely replace `std`

Personal Vision

- Get the language features right in C++20
- Build std2 in an experimental namespace, building a document around the Ranges TS
- Land the experimental library in 2022 for C++23, once it has sufficient substance to stand alone

Expected Direction

- Land Ranges for C++20
  - Establishes basic/foundation library concepts
  - Incrementally build out the library
- Ongoing concerns whether new functionality goes into `std` or `std2`
- Continue retrofitting new language features onto `std` library
  - Concerns about concepts in `std`, such as for containers

Language Drivers

- Concepts :
- Contracts :
- Coroutines :
- Modules :

Language Drivers

- Concepts : constrain all our templates
- Contracts : revisit requirement on *all* of our code
- Coroutines : cleaner model for concurrency; expect generators to drive ranges for algorithms
- Modules : rethink how we aggregate and deliver all of our functionality

Question

- How far should disruptive language features be applied to existing std?
  - Modules?
  - Concepts?
  - Contracts?

Modules

- Approach: Replace each header with a single `import std_module;` and deprecate headers
  - macros may need a little more attention
- What is the right level of complexity/functionality for a single module in the standard library?
  - one std module?
  - fine grained per-existing header approach?
  - Balanced functionality, such as algorithms, containers, text-handling, streams, etc.

Concepts

- Can we apply concepts to existing library without breaking users?
  - Many places library requires SFINAE-like tricks today, easily expressed with concepts
  - We may get a poor concept vocabulary if we restrict ourselves to existing SFINAE contraints
  - Existing constraints can all be expressed with `requires` on type traits

Personal Concerns

- Concepts vocabulary in std
  - std templates would (mostly) satisfy concepts in a proposed std2 library, and vice-versa
    - concepts are not a type hierarchy, so it is not a problem of deriving from wrong abstract type
- use of concepts to implement existing “shall not participate in overload resolution” constraints?
  - clearly permitted, but should it be mandated in library specfication?

Contracts

- Opportunity to move standard wording directly into code
- Could significantly impact how we render the specification of a library
    - Concern about inconsistent presentation if not approached uniformly across a whole library
- How detailed a contract becomes distracting when applied to every library function?

Concerns

- Can vendors add their own contract annotation reflecting library documentation?
  - Or are they limited to `[[assert]]` in function definitions?
- Are checking levels an essential part of library contracts?
  - Do L(E)WG need to review the right checking level per-predicate-per-function?
  - Do vendors have the freedom to strengthen or weaken checking levels?

Namespace Issues

Questions

- Can we provide a compete replacement library in a new namespace?
- Can two top-level namespaces coexist?
- Can we import elements of one namespace into the other with `using` (and `typedef`)?

Concerns: compatibility

- Compiler ABI in `std`
  - exception hierarchy and `terminate`
  - `initializer_list`
  - `type_info`
  - `new`
  - atomics?

ADL Customization Points

- Language follows special rules when needing operators/begin/end/get<>
- ADL enabled by `using`, but ambiguities if using both namespaces `std` and `std2`
    - resolve ambiguity with explicit qualification, which disables ADL
- `swap` is problematic:
    - Keep it in a fundamental ‘std’ module?
    - Revisit swaperator?

ADL Solutions

- Look for something new in the language?
- Never call ADL customization points directly
  - Provide an ADL invoker in the `std` (or `std2`) namespace
- Ranges TS is providing more function objects that solve similar problems

Freestanding Module

Proposed Solution

- Move all fundamental ABI libraries into a freestanding module
  - freestanding is the name for a library with minimal language support
- Implicit import of `core` module?
  - Or perhaps just require that import on each std header?

Contents

Freestanding headers in C++17

<cfloat>
<climits>
<cstddef>
<cstdint>
<cstdlib>

<atomic>
<exception>
<initializer_list>
<limits>
<new>
<type_traits>
<typeinfo>

Proposed Additions

- declval, forward, move, swap
- get<>, tuple_element, tuple_size
- addressof
- equal_to, hash, less?
- iterator traits?
- begin/end etc? (library includes these in many headers)
- `array`? Or do we want a new class in `std2`?

Suggested Restriction

- Implementations must ensure a strict separation of `core` module and rest of library

- Makes it easier for a vendor to supply their own library implementation on top of compiler provided core functionality

Niggles and Nudges

tuple and pair

- `template <typename T1, typename T2>`
  `using pair = tuple<T1, T2>;`
- do we still need both names?
  - if so, would we specialize tuple of two elements to have named members `first` and `second`?
  - wholesale replacement of `pair` through standard APIs in new namespace
  - with structured bindings, and aggregate structs with named members a better choice than `tuple`?

What Can We Learn From vector?

- vector<bool> and proxies
- Constraints vs. support for incomplete types
  - Cannot constrain a constructor (concepts/SFINAE) if constraint relies on an incomplete type
  - Need to constrain to correctly respond to traits
  - Need to support incomplete types for recursive data structures
- Range constructors can replace the special case for `initializer_list`
- Deduction guides and concepts?

What can we learn from map?

- `pair` is a `tuple`, don't need a distinct type
- Iterator should return a projection (tuple of references) rather than force data to be stored together, with physically `const` key
    - Better key locality when searching
    - can splice nodes without laundering
- `default_order` makes a better customization point
    - by default heterogeneous like the diamond functors
    - note: functors in `std2` should be non-template classes with diamond behavior

To What Extent Should We Support *Evil* Types

- Without restriction, there are many types with wrinkles that greatly complicate library support and testing

- If we are going to constrain the library, provide a set of concept that do not bend over backwards to support corner cases

- Remember, these same issues apply to anyone trying to write a class using the standard concepts, so not just a concern for std vendors

To What Extent Should We Support *Evil* Types

- overload operator & (only with non-standard semantic?)
  - workaround is to use std::addressof, making code less readable
  - should we allow overloads that return true address (as correct pointer type)?
- Overload operator&& and operator||
  - breaks short circuit evaluation
  - also, implies support for predicates returning non-bool types
  - typical workaround is explicitly casting to bool everywhere
- Overload operator,
  - breaks sequencing guarantees, although maybe fixed in C++17

To What Extent Should We Support *Evil* Types

- Explicit copy/move/default constructors
  - break all sorts of assumptions in surprising ways
- Explicit constructors in general
  - Different syntax for initialization
- Aggregates
  - As above, but worse!
- Default constructor has different semantics to list initiation with an empty list
- Throwing move/swap (c.f. allocators)
  - Especially on members with joint constraints e.g., unordered hash/equality
- swap is different to repeated moves (c.f. allocator traits)

To What Extent Should We Support *Evil* Types

- Convert to/from anything
  - Can create awkward ambiguities, and steal operations unexpectedly
- member and free function have different semantics
- Volatile types
  - Need to support volatile qualifiers in many implementation details for widespread support, rarely tested
- Cv-ref qualified member functions and overloads
  - notably on operator()
- References as data members
  - interesting copy/assign behavior
  - reference_wrapper may be a good-enough substitute

Why is volatile evil?

```
template <typename T>
struct wrap {
   T volatile x;
};
```

```
static_assert(is_convertible_v<wrap<T>, wrap<T>>);
```

PASS

```
int
```

FAIL

```
struct empty {};
```

PASS

```
struct X {
   template <typename T> X(T&&){}
};
```

PASS?

```
struct X {
   X(...){}
};
```

What about awkward predicates?

- Predicates taking reference to non-const-qualified arguments
  - may cause issues for const-qualified member functions
- Predicates with non-const qualified operator()
  - *does* cause issues for const-qualified member functions
- Predicates with cv-qualified operator()
- Predicates returning non-bool
- Predicates returning evil-value non-bool types
- Move-only predicates
- Throwing on copy/move/swap

Wholesale

Replacement

Which Facilities Might We Replace Entirely?

- Concurrency
- Text handling
- Streaming
- valarray

Concurrency

Existing Issues

- async creates objectionable futures
- Semantics of future destructor
- pending support for executors
- Ranges TS and parallel algorithms

Opportunities

- A new parallel vocabulary and framework built around executors
- Constrain with appropriate concepts from the start
- Coroutines will further encourage a cleaner API

Text Handling

basic_string

- Fat class - too many members
- too many template parameters
  - allocator (see later)
  - char traits (rarely customized)
  - char type (do we really need 4 vocabulary types)
- Competing with `string_view` as fundamental abstraction
  - OK, 8 vocabulary types
  - Which template gets to be the primary “`s`” literal

Unicode

- Portable solution to international text
  - portable across programming languages and platforms
  - assumed for common standards like xml, html, and javascript
  - many resources online for solving problems with Unicode - no longer a C++ specific problem

Unicode Issues

- Multiple encoding: UTF8,16,32 with endian properties
- Must assume multibyte encodings
  - no longer simple to index into arrays/strings
  - true even with UTF32
- Multiple representations for the same character
  - May need some normalization to process text

Proposed Solution

- New classes for std2: `text` and `text_view`
- Simple classes, not templates
- text classes are immutable (so thread-safe)
- Composable like a rope
- All text is stored in a native representation, like `file_path`
- APIs to export as specific UTF encoding
- Maybe even different iterators for each UTF encoding

Ripple Effect

- All text APIs in std2 should expect text_view or (rarely) text
- Greatly simplify regex interface
- Simplify streams
  - but that is a whole new can of worms…

Obvious Problems

- Needs to be championed by an expert in the field
  - NOT me!
    - Experts are a limited pool right now: Unicode experts have little incentive to use C++ today
- Interoperability with `std::string`
  - Notably, strings in exception classes (ABI)

Allocators

Why Allocators?

- Performance
- Performance
- Performance
- Additional capabilities

Performance

- Well chosen allocators can greatly improve memory locality
    - See John Lakos talk earlier this week (may need time machine, or online video)
    - Memory pools minimize the effect of *diffusion* on a single task
    - Memory pools on the stack reduce fragmentation of long running processes
    - Keeping memory in L1/L2 cache has an enormous impact on runtime performance
        - although CPU is trying to manage cache to make this happen anyway, a local memory pool goes a long way to help

Utility

- Custom allocators can add extra functionality in addition to supplying memory
  - Logging
  - Instrumentation
    - Debugging
    - Profiling
    - Test drivers

Complexity

- allocator traits are flexible in too many fine-grained ways, with a small subset of useful combinations
- allocator spam in constructors for types that allocate
- generic wrappers need allocator-aware constructors to pass allocators to wrapped types
- aggregates (including arrays) cannot support custom allocators, as no constructors

Missing features

- No easy ability to query an object for its allocator
- Can ask “do you use this type of allocator?”
- Cannot ask “Which allocator(s) do you support, if any?”
  - no allocator support can be valuable optimization
- No guarantee that I can ask an object for its allocator, even when trait says it uses one

Personal Vision

- Adopt a single allocator model
  - Pro: reduces complexity
  - Anti: reduces customizability
  - Suggestion: pick a flexible model (`pmr`)
- Keep allocators out of the type system

Ideal Model

- No allocator spam in the interface
- a single data structure all uses the same allocator
    - e.g., container and its elements, a graph, its nodes, and their contents, etc.
- If a type manages dynamic memory, it uses an allocator
- “allocator aware” types are known to the type system
    - can query if a type is allocator aware

Allocator Awareness

- A type is allocator aware if:
  - it says so (need a way to mark a class)
  - it derives from an allocator-aware class
  - it has data members that are allocator aware

Implementing Awareness

- Stash an allocator pointer at construction, much like a vtable pointer
  - does not vary through constructing a hierarchy though
- If awareness is derived, access the allocator through the entity requiring the allocator
  - do not pay to store excess copies of the pointer
- Customization API to give precise control of storage if needed
  - e.g., optional object needs to stash allocator when no engaged, but can re-use storage for missing object

Allocator Injection

- Inject an allocator at object creation time, in parallel to constructor
  - needs language support with an extension syntax
  - *Implicitly* propagate that injection through member initializers for all bases and members
  - query function to retrieve the installed allocator
  - do not pay storage for allocator that may be retrieved from a base or member

What is an Allocator?

- allocator is an abstract base class
- injectable allocators derive (pubicly, unambiguously) from that base class
- system has a default allocator, that typically calls the `operator new` and `operator delete`.
- default allocator is used if no other allocator is injected

Generalized Feature

- Property injection is generally useful facility
- Find a general mechanism, with allocators as the motivating example

Example Code

```
template <Movable Type>
class vector [[injects(allocator)]];

using   not_aware = tuple <int>;
using alloc_aware = tuple <int, vector<int>>;
```

```
LocalAllocator from_stack;

vector<int> data{1,2,3};  // default allocator
vector<int> more{4,5,6}[[inject(from_stack)]];

alloc_aware simple{0, more}; // copy more, default allocator
alloc_aware movable{0, move(more)}; // use from_stack
alloc_aware* ptr =
             new[[injects(from_stack)]] alloc_aware{0, data};
```

Don’t Pay if you Don't Use

- If you never inject an allocator, change ABI to avoid storing the pointer
- Probably a compiler switch as it affects ABI
  - Can warn/error on attempt to inject
  - Otherwise as no effect on allocator-aware types
  - Maybe a compiler optimization for local classes such as lambdas, via static analysis
- Problem of mixing ABIs if you do not control your whole world

Which problems are solved?

- type erased allocators for `function`, `any`, and `packaged_task`
- injecting allocators to `optional` and `variant` only if members use allocators
- allocator support for `array` and other aggregates
- Eliminates redundant allocator pointers
- Eliminates allocator spam in interfaces

Wrap Up

Project Plan

- Create a freestanding `core` module, common to `std` and `std2`
  - migrate essential functionality from `std` into freestanding for C++20
- Adopt Ranges into `std2` with minimal regard to std algorithms - let it excel as its own thing
- Next project should be a new concurrency library, unhindered by suboptimal choices of `std`
  - fully exploit new language features like concepts and coroutines
- Similarly, define the basic set of vocabulary types, most of which will be constrained templates
  - `any`, `array`, `optional`, `tuple`, `variant`, and `vector`
- explore right balance for scale of modules, use of contracts, etc.

Ongoing Research

- Create a better solution for text handling
- Create a better solution for streaming and serialization
- Flesh out an implementation of allocator injection (really needs proof of concept before proposal) before finalizing new containers
- Assuming allocator injection, explore many small cleanups to container library

Other Essentials

- Style guide for new library
  - design guidelines
  - naming guidelines
  - documentation guidelines
- Migration guide and interoperability with `std`
  - Guidance for which namespace new proposals should target