C++20 Language Features for a new Library

Alisdair Meredith
BloombergLP

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

Why *this* talk?

- New library should have solid foundations
- What are the missing or anticipated language features that should be integral to any new library design?
- Can we identify those that are critical, and ensure they land in C++20?

How to pick features

- Language feature will change the nature of abstraction in the language

- Language feature would have a clear impact on library interfaces

- Language feature would have a fundamental impact on library organization

Format of this talk

- Assume a broad familiarity with the ideas behind many of the proposals
- Will zoom in on a couple of interesting that merit further discussion as part of standardization
- Will not provide in depth tutorials - each might be a session in its own right
- Try to bring focus back to library design

Modules

Modules

- The single, most fundamental change
- Affects every library entity, whether a class, function, template, or alias
  - Apart from (maybe) macros
- Also the smallest, as more about packaging than content

State of the art

- Clang has been evolving their model of modules for around 5 years

- MS have been shipping their experimental modules since VC2015, and upgrade in subsequent releases

- ISO close to finalizing a modules TS, drawing on experience from both schools of thought

- gcc now looking to implement modules TS

What is a module?

- Library level abstraction
- A collection of multiple TUs that can be accessed through a single import directive
- C++ code, not repeated textual copy/paste of #include
    - Ideally one trip to file system to access file, not hundreds of chained headers per TU

Basic syntax

-
  ```
  module identifier;  // names the module
  ```

-
  ```
  export namespace blah {
     // exported declarations, and class definitions
  }
  ```

-
  ```
  import identifier;  // access contents of the
                      // named module
  ```

Module contents

- a single interface file declares all exports
- definitions for all exported contents, potentially split across multiple TUs
- Additional declarations and definitions are not exported, so not accessible outside the module
  - truly private, not subject to ODR *across modules*
  - Obviously ODR applies within a module

Module Constraints

- No cycles between modules
    - Import graph must be a DAG
      Directed Acyclic Graph
    - Mutually dependent classes must reside in the same module, but maybe different TUs
    - Templates may circumvent these rules, but retain the import DAG

How to split a module?

- How do multiple TUs access the class definitions from their interface file?
    - reinventing #include?
    - more syntax needed?
    - build system recognizes non-interface TUs, and implicitly imports the interface for every TU with that module identifier?

Open questions: 1

- Can a module export a macro?
  - *My* preference: no
    modules export code, macros are preprocessor utilities that manipulate source text
    access macros using preprocessor, with a traditional #include
- Transition to from C++17 to modules may require mixed mode support, including macros

Open Questions: 2

- Must the module declaration be the first substantive line in a TU?

Open Questions: 3

- templates?!
  - how do we avoid recreating the problems of C++98 exported templates?

Bloomberg experience

- Bloomberg architecture:
  - A component is a .h/.cpp pair
  - A package comprises multiple components
  - A package group comprises multiple packages
  - No higher level of aggregation
  - No cycles between components

Bloomberg Modules?

- A component forms a single module
- A package is a module comprising component modules
- A package group is a module comprising package modules (never components directly)

Early Experience

- Package modules export a *dependency* on component modules
  - Modules are always assembled by-reference, never by-value
- Result is we would distribute thousands of modules - granularity feels wrong
- Suggestion this might be addressed by a future module partitions feature

Concepts

Concepts

- Major feature, long time coming
- No checked definitions (and that is a good thing)
- Ranges library as pilot proof-of-concept(s)
- Terse syntax essential for lambdas
- Forwarding problem with terse syntax

Assumptions

- This is a technically literate audience that is aware, in a broad sense, that concepts is a language feature in development for over a decade

- Understand the basic idea of a syntax to constrain parameters of generic code

- Serve as documentation for both the user and the compiler

Current State of the Art

- Concepts TS published November 15, 2015
  - No longer Concepts *Lite*
- gcc 6.1 shipped a supported implementation April 2016
- L(E)WG hard at work building on this with Ranges TS as foundation on library work

Features of Note

- Concepts can be declared with either a variable or a function syntax
- Concepts always return `bool`, or are `constexpr bool` variables, but must still state the `bool`
- `requires` expressions are usable only in requires clauses
  - relaxed since TS was published

Features of Note

- A much more expressive and efficient syntax for SFINAE conditions

- Constraint-based overloading on otherwise identical signatures e.g., overloading for forward iterators and random access iterators

- Extremely flexible syntax for declaring constraints

And there’s more…

- “normal” form uses concepts like types and does away with the template head
- `auto` can now be used for a function parameter to create an unconstrained template
  - Would this be better handled by an `Auto` library concept?

A quick example

```
void foo(ArgType&& arg) {
   bar(arg);
}
```

A quick example

Q: Can we call `foo` with an lvalue?

```
void foo(ArgType&& arg) {
    bar(arg);
}
```

A quick example

Q: Can we call `foo` with an lvalue?

```
void foo(ArgType&& arg) {
    bar(arg);
}
```

A: It depends…

A quick example

Q: Can we call `foo` with an lvalue?

```
void foo(ArgType&& arg) {
    bar(arg);
}
```

A: It depends…

Is `ArgType` a type or a concept?

Possible Solution?

Real syntax for forwarding references

```
void foo(ArgType&&& arg) {
    bar(arg);
}
```

- `arg` behaves strictly as an lvalue or an rvalue, depending on call context
- Whether `ArgType` is a type or a concept
- Essentially a mini-template when `ArgType` is a type
- We need to find a better syntax!

Possible Solution?

Real syntax for forwarding references

```
void foo(ArgType&&& arg) {
    bar(arg);
}
```

- Eliminates need for `std::forward`?
  - Unconstrained templates retain, but deprecate, old behavior
- && is always an rvalue unless template keyword is present
  - not an easy thing to express today without SFINAE hacks

Multiple Deductions of a Concept

```
template < typename ExecutionPolicy
         , typename ForwardIterator1
         , typename ForwardIterator2
         >
void copy( ExecutionPolicy&& exec
         , FowardIterator1   first
         , FowardIterator1   last
         , FowardIterator2   result
         );
```

Multiple Deductions of a Concept

```
template < ExecPol  ExecutionPolicy
         , FwdIter  ForwardIterator1
         , FwdIter  ForwardIterator2
         >
void copy( ExecutionPolicy&& exec
         , FowardIterator1   first
         , FowardIterator1   last
         , FowardIterator2   result
         );
```

Multiple Deductions of a Concept

Do `first`, `last`, and `result` deduce to the same or different types?

```
void copy( ExecPol&& exec
         , FwdIter   first
         , FwdIter   last
         , FwdIter   result
         );
```

Multiple Deductions of a Concept

I propose adding a tag to guide deduction

```
void copy( ExecPol&& exec
         , FwdIter.1 first
         , FwdIter.1 last
         , FwdIter.2 result
         );
```

Multiple Deductions of a Concept

I propose adding a tag to guide deduction

```
void copy( ExecPol&& exec
         , FwdIter.1 first
         , FwdIter.1 last
         , FwdIter.2 result
         );
```

Without a tag, all named concepts deduce to the same type, and so behave on the page just like a type does today.

Tagged Concepts

For consistency, tag would apply through whole deduced scope, e.g., for variable declarations

```
void func(ArgType.1 x, ArgType.2 y) {
    ArgType.1  z{x};  // copy x
    ArgType.2 *p{&y}; // take address of y
    ArgType.1  q{y};  // ...
}
```

Contracts

Contracts

- Run-time analog of concepts
- Support better testing/support for software
- Support static analysis tools
- Clearly specify the rights and responsibilities of the caller of a function

Basics

- Proposal uses modified attribute syntax
- [[expects : predicate]] indicate preconditions the caller must satisfy
- [[ensures : predicate]] gives a guarantee from the function if it returns
- [[assert : predicate]] can be used in a function body to check preconditions that are in the doc.

Violation Handlers

- Predicates may be checked at runtime, and a handler will be called if the predicate is false
- Specify (on the build line) the handler to respond to a failed check
  - defaults to calling `abort`
  - if a handler throws out of a `noexcept` function, it violates the exception specification, and terminates
- Extra flag to allow handlers to return back to caller
  - e.g., to install a logging handler when annotating pre-existing code

Restrictions

- Handler cannot be changed at runtime
    - although could install a handler with its own registration scheme
- predicates respect the access specifier of the function
    - public functions can check only against public members
    - protected functions cannot check against private members

Checking Levels

- Not all contract checks are cheap
  - e.g., post-condition that a sort function produces a permutation of the original sequence
- Allow annotation to express a simple cost
  - default : always check
  - audit : check only if command line enables expensive checks, typically violating complexity guarantees
  - axiom : never run, but useful information, e.g., for static analysers, `is_valid_pointer()`
- Allow the whole feature to be disabled on the command line

Concerns

- What information should be passed to the handler?
    - Should we have options to restrict detail to save program size?
- Are we concerned about violation handlers racing?
    - (should not be messing with shared state anyway)
- Should library document checking strength?
- Are highly detailed contracts on function signatures helpful or harmful?
    - How much library doc should move into code?

Quick Example

Could this replace `at()` in a new library?

Should we mark the functions `noexcept`?

```
template <typename T, typename Alloc>
struct vector {
   T & operator[](size_type index)
      [[expects : index < size()]];

   T & insert(T const &)
      [[ensures : !empty()]];
};
```

Coroutines

Coroutines

- Championed by Gor Nishanov, with a TS currently under ballot in ISO
- Going TS route, rather than straight-to-standard, to allow time for a competing proposal from Google/ASIO folks

Coroutines

- Allow you to exit/return from a function, and then resume where you left off

- enables *generators* : ranges defined to execute a function

- Asynchronous coroutines return a future-like type

Language Impact

- 3 new keywords
  - `co_await`
  - `co_return`
  - `co_yield`
- Allow `co_await` on range-based `for` loops
- Coroutine traits allow for custom implementations

Why coroutines are important

- Simplify use of asynchronous callbacks in concurrent code

- Library interface should assume feeding generators into algorithms, much like we might expect iterator adaptors to avoid many variations in C++98 algorithms

Quick Example

```
generator<int> coro(int from, int to) {
   while (from != to) {
      co_yield from++;
   }
}

int main() {
   for (int x : coro(3, 7)) {
      cout << x << endl;
   }
}
```

Second example

```
async_stream<int> Ticks() {
   for (int tick = 0; ; ++tick) {
      co_yield tick;
      co_await 1ms;
   }
}

future<int> Sum(asynch_stream<int> & input) {
   int sum = 0;
   for co_await(auto&& v : input) {
      sum += v;
   }
   co_return sum;
}
```

Unified Call Syntax

Unified Call Syntax

- There is a split in preferred convention between libraries preferring member functions over free-functions accessing the public API of a class

- std library is split on this, duplicating more members as free functions to be called via ADL

- It would be nice if the language would transparently allow both free and member-syntax invocation for the same function

    - Simpler to write generic code without requiring a function to be a member (or not)

Concerns

- Managing the overload set of members/non-members without introducing ambiguities or other surprises
    - One suggestion to use a new .call(x,y) syntax to invoke the uniform rules
- Does every member name become an extension point, making generic code nigh impossible to write in the abstract?

Why it matters?

- Do we want to carry on extending the idiom of begin/end/cbegin/cend/rbegin/rend/crbegin/crend/size/empty/data?
- Issue matters less if we know uniform syntax available

Default Comparisons

Comparison Operators

- Long standing desire to generate, or default, comparison operators
- `std::relops` has not been a popular choice for this, over the years
  - we are lacking a tag-type to derive from, which would drive ADL
- Can simplify implementation by using `std::tie` on members, but fragile if class evolves.

Spaceship Operator

- Proposed by Herb Sutter at most recent Kona meeting, and well received

- Basic idea: `operator<=>` returns -ve, 0, +ve to indicate less-than, equal-to/equivalent, or greater-than

Devil is in the Details

- Not all types support ordering, so want boolean result if types are only equality comparable
- May need to know if types are strictly ordered, or only partially ordered
- Result of operator is not an int, but a tag-type wrapping an int, that encodes this behavior

Broader Semantics

- If a type defines only `operator<=>`, the other operators can be synthesized on demand

- Demand a consistent definition

`a == b`	`a <=> b == 0`	`0 == b <=> a`
`a != b`	`a <=> b != 0`	`0 != b <=> a`
`a <  b`	`a <=> b <  0`	`0 <  b <=> a`
`a <= b`	`a <=> b <= 0`	`0 <= b <=> a`
`a >  b`	`a <=> b >  0`	`0 >  b <=> a`
`a >= b`	`a <=> b >= 0`	`0 >= b <=> a`

Example Implementation

Example taken directly from P0515R0

```
std::strong_ordering   operator <=> (const TotallyOrdered& that) const {
  if (auto cmp = (Base&)(*this) <=> (Base&)that;     cmp != 0) return cmp;
  if (auto cmp = last_name.     <=> that.last_name;  cmp != 0) return cmp;
  if (auto cmp = first_name     <=> that.first_name; cmp != 0) return cmp;
  return         tax_id         <=> that.tax_id;
}
```

Example Implementation

Example taken directly from P0515R0

```
std::strong_ordering   operator <=> (const TotallyOrdered& that) const {
  if (auto cmp = (Base&)(*this) <=> (Base&)that;     cmp != 0) return cmp;
  if (auto cmp = last_name.     <=> that.last_name;  cmp != 0) return cmp;
  if (auto cmp = first_name     <=> that.first_name; cmp != 0) return cmp;
  return         tax_id         <=> that.tax_id;
}
```

Note neat use of C++17 `if` with a variable declaration

Default Definition?

- Request a default definition with `= default`

- Like existing special members, the exception specification is also deduced for defaulted definitions

Overloading `constexpr`

`constexpr` overloads

- A real desire to make more of the library `constexpr`, or at least `constexpr`-friendly
- don't want to pay a cost for less efficient run-time behavior, restricted to `constexpr`-friendly syntax
    - e.g., `asm`, or use of intrinsics for `memcpy` etc.

`constexpr` overloads

- solution 1:
  allow overloading functions on `constexpr`

- solution 2:
  introduce an `is_constexpr_eval` intrinsic that can be used as predate for `if constexpr`

  - and perhaps assume all functions may be `constexpr`, and fail only if not usable at call-point

Quick example

```
template <>
constexpr
char * char_traits<char>::copy( char         * s1
                              , char const   * s2
                              , size_t         n)
{
   if constexpr (is_constexpr_eval) {
      while (n--) { *s1++ = *s2++; }
      return s1;
   }
   else {
      return strcpy(s1, s2, n);
   }
}
```

Reflection

Reflection

- Reflection is a popular facility in many languages with a managed runtime

- C++ libraries emulate compile-time reflection with SFINAE tricks, some of which is obviated by concepts

- Runtime reflection would (probably) add considerable cost to object sizes, and may start to dictate more implementation details for language representation than compiler vendors are happy with

Why do we want it?

- Adapting software on the fly at compile and runtime
  - e.g., creating data bindings after querying a database for its schema at runtime
  - e.g., querying a class for its members and signatures, to drive an automated test driver
  - etc.

Progress

- Reflections study group is broadening scope to cover meta-programming in general, as a typical participant in compile-time reflection schemes

- First results will be reviewed in EWG over the next few meetings…

Transactional Memory

Transactional Memory

- An active research topic to produce a simpler concurrent programming model
- Modeled on database transactions
  - optimistically run some code, and commit to memory if no races are detected
  - otherwise roll back, and try again
- Transaction-safe code must be explicitly marked
  - Library must document/mark transaction safety

Transactional Memory

- Expected to have a minor cost with hardware support, larger impact with software solution
    - Hardware support has been coming for some time
- We have an experimental TS
    - Implementation available since gcc 6.1
- Expecting further work

Proxy Support

Proxy Support

- how can we better support iterators returning proxies?
- should we be able to overload operator dot?
  - can we write generic code when both `operator.` and `operator&` may be overloaded?
  - Nervous about `addressof` all over again - how do I get address out of my proxy?
  - (not possible for true rvalues)

Proxy Support

- I got scared and stopped looking at this point
- Consider me a source of FUD!

What do we want?

- Looking ahead to next standards, which features are critical for a new library
  - Which features did I miss?
- Push for those in C++20
  - get involved
- Which features would redirect significant effort if they arrived later?
- Which features can we absorb incrementally in C++23 and beyond?

My Priorities

- Modules
- Concepts
- Contracts
- Coroutines

My Wish List

- Runtime Reflection
- My own enhancements to concepts ;)
- Allocator support (see tomorrow!)
- Transactional memory

Nice to Have

- Spaceship operator
- Better control of constexpr code generation

Would rather rule out

- Universal calling syntax
- Overloading operator dot