A review of C++ 11/14 only Boost libraries

Boost.Fiber Boost.AFIO Boost.DL Boost.APIBind

Niall Douglas

Contents:

1. Libraries examined, and rationale for excluding some and not others
5. Boost.Fiber (provisionally approved)
4. Boost.AFIO (review queue, Oct 2013)
3. Boost.DL (review queue, Jan 2015)
2. Boost.APIBind (née BindLib)
6. Big picture answers - and why we need a C++ 11/14 only modular Boost 2.0

C++ 11/14 only Libraries examined

15 Libraries:

- AFIO ****
- BindLib (APIBind) ****
- DI **
- Dispatch
- Expected ***
- Fiber ***
- Fit *
- Hana *
- Http **
- Proto ox
- Range
- SIMD
- Spirit X3
- Sqlpp11 *
- Tick *

How close are these to entering Boost?

Name	Authors	Min C++	Headers required	Entered peer review queue	Boost ready source code	Boost ready unit testing	Boost ready docs	Boost ready build	Uses Travis & Appveyor	Uses valgrind memcheck	Uses coveralls
Boost.Fiber	Oliver Kowalke	14	Context, Config	Conditionally accepted	1	1	1	1	0	0	0
Boost.AFIO	Niall Douglas, Paul Kirth	11	none	2013-10	1	1	1	header only	1	1	1
Boost.DI	Krzysztof Jusiak	14	none	2015-01	1	0.9	1	1	1	1	1
Boost.Hana	Louis Dionne	14	none	2015-04	1	0.9	0.6	header only	1	1	0
Boost.Http	Vinícius dos Santos Oliveira	11	asio, filesystem, system, datetime, utility	2015-04	1	0.3	0.3	0	0	0	0
Boost.APIBind	Niall Douglas	11	none	0	1	0.1	0	header only	0.5	1	1
Boost.Expected	Pierre Talbot, Vicente J. Botet Escriba	11	none	0	1	1	0.1	header only	0	0	0
Boost.Tick	Paul Fultz II	11	none	0	0	0.9	0.9	header only	1	0	0
Boost.Fit	Paul Fultz II	11	none	0	0	0.9	0.9	header only	1	0	0
Boost.Sqlpp11	Roland Bock	11	none	0	0	0.5	0.4	header only	1	0	0

And the others ...

- Proto 0x, Range, Spirit X3
  - Not approaching a Boost peer review yet I felt
- Hana, Tick, Fit
  - Authors are presenting at this conference, no need to (badly) duplicate them …
- SIMD and Dispatch
  - C++ 14 version reimplements 98 version without many surprises
- So this talk covers in this reversed order: APIBind, AFIO, Fiber and DI

Boost.APIBind

(still BindLib at time of writing)
Lead maintainer: Niall Douglas (me)

Boost.APIBind - what is it factual?

“toolkit for making Boost vs STL 11 dependency injectable”

“toolkit for versioning library APIs and ABIs”

“toolkit for hard version API binds rather than ‘whatever available’ API version”

“toolkit for explicitly encoding dependencies between libraries”

Boost.APIBind - what is it philosophically?

"toolkit for enabling a Boost library to be optionally standalone from Boost i.e. **modular Boost made real**"

Requires: No Boost dependencies (obviously)

Min >= Compilers: Any C++ 98

Max >= Compilers: clang 3.2, GCC 4.7, VS2013*

* VS2013 doesn't support API/ABI versioning due to lack of inline namespace support in the compiler.

Parts

Boost.APIBind - parts 1/3

- Absolute minimal Boost emulation headers
  <boost/config.hpp>
  <boost/test/unit_test.hpp>
- cpp_feature.h
  Provides consistent SG-10 compiler feature checking macros for all compilers
- import.h
  Preprocessor macros easing loose coupling

Boost.APIBind - parts 2/3

- Incomplete STL11 binds for array, atomic, chrono, condition_variable, functional, future, mutex, random, ratio, regex, system_error, thread, tuple, type_traits, typeindex
  - Can bind to either the STL or Boost
- Incomplete STL1z binds for filesystem, networking
  - Can bind to the STL (Dinkumware), Boost, or standalone ASIO the Networking TS reference impl

Boost.APIBind - parts 3/3

- A python script for helping autogenerate macro boilerplate for ABI signature generation
- libclang AST parsing tool which converts library header files into API binds
  - This is used to break up legacy monolithic namespaces into modular rebindable parts
  - Any library in its own namespace doesn’t need this

Layer 1: Versioning API and ABI

Enabling your library to coexist with itself using the C preprocessor, inline namespaces, and namespace aliasing

Boost.APIBind - macros from import.h

- `BOOST_BINDLIB_NAMESPACE(sig)`
- `BOOST_BINDLIB_NAMESPACE_BEGIN (sig)`
- `BOOST_BINDLIB_NAMESPACE_END(sig)`
- `BOOST_BINDLIB_INCLUDE_STL11 (prefix, impl, lib)`
- `BOOST_BINDLIB_INCLUDE_STL1z (prefix, impl, lib)`

Boost.APIBind - version signatures

```
#define BOOST_AFIO_V1
(boost), (afio), (v1, inline)
#define BOOST_AFIO_V1_NAMESPACE
BOOST_BINDLIB_NAMESPACE(BOOST_AFIO_V1)
#define BOOST_AFIO_V1_NAMESPACE_BEGIN
BOOST_BINDLIB_NAMESPACE_BEGIN(BOOST_AFIO_V1)
#define BOOST_AFIO_V1_NAMESPACE_END
BOOST_BINDLIB_NAMESPACE_END(BOOST_AFIO_V1)
```

Boost.APIBind - in your library

Traditional:

```
namespace boost { namespace afio { struct foo; } }
boost::afio::foo;
```

APIBind:

```
BOOST_AFIO_V1_NAMESPACE_BEGIN
struct foo;
BOOST_AFIO_V1_NAMESPACE_END
BOOST_AFIO_V1_NAMESPACE::foo
```

Boost.APIBind - use of your library 1/3

I want to use latest AFIO in mylib please:

```
#include <boost/afio/afio.hpp>
namespace mylib {
  namespace afio = boost::afio;
}
```

The inline namespacing hides the real namespace. Latest AFIO version therefore always appears at boost::afio.

Boost.APIBind - use of your library 2/3

I want to use specifically only **v1** of AFIO in mylib please:

```
#include <boost/afio/afio_v1.hpp>
namespace mylib {
  namespace afio = BOOST_AFIO_V1_NAMESPACE;
}
```

Boost.APIBind - use of your library 3/3

```
namespace boost { namespace afio {
  namespace v1 { … }           // legacy
  inline namespace v2 { … }    // latest
} }
#include <boost/a> // uses AFIO v1. Works!
#include <boost/b> // uses AFIO v2. Works!
boost::afio::foo;  // Finds latest (v2)
BOOST_AFIO_V1_NAMESPACE::foo;  // v1
```

Layer 2: Dependency Injection of which STL to use per version of your library

Enabling any user specified configuration of your library to coexist with itself

Boost.APIBind - Multi ABI 1/8

- What if you would like your library to use **either** Boost.Thread or the STL 11 Thread?
- What if you would like your library to use **either** Boost.Filesystem or the STL 1z Filesystem TS?
- What if you would like your library to use **either** Boost.ASIO or the STL 1z Networking TS?

Boost.APIBind - Multi ABI 2/8

What if *header only* library A is dependent on Boost.AFIO v1 configured with Boost.Thread, Boost.Filesystem and Boost.ASIO

BUT

Header only library B is dependent on Boost.AFIO v1 configured with STL 11 Thread, STL 1z Filesystem and STL 1z Networking?

Boost.APIBind - Multi ABI 3/8

This problem is *highly* likely in future Boost libraries

- Only Boost.ASIO and Boost.AFIO currently let external code *dependency inject* Boost OR STL11
- Of the five libraries in the review queue, three only use STL11 and one only uses Boost - this is a big future problem to fix

Boost.APIBind - Multi ABI 4/8

1. Decide on user set macros for each ABI config option:
   - BOOST_AFIO_USE_BOOST_THREAD = 0|1
   - BOOST_AFIO_USE_BOOST_FILESYSTEM = 0|1
   - ASIO_STANDALONE = 0|1
2. Have config.hpp convert those into:
   - BOOST_AFIO_V1_STL11_IMPL = std|boost
   - BOOST_AFIO_V1_FILESYSTEM_IMPL = std|boost
   - BOOST_AFIO_V1_ASIO_IMPL = asio|boost

Boost.APIBind - Multi ABI 5/8

3. Instead of

```
#define BOOST_AFIO_V1 (boost), (afio), (v1, inline)
```

do:

```
#define BOOST_AFIO_V1 (boost), (afio),
(BOOST_BINDLIB_NAMESPACE_VERSION(v1,
BOOST_AFIO_V1_STL11_IMPL,
BOOST_AFIO_V1_FILESYSTEM_IMPL,
BOOST_AFIO_V1_ASIO_IMPL), inline)
```

Boost.APIBind - Multi ABI 6/8

4. Call gen_guard_matrix.py with the user settable ABI config macros:

```
./gen_guard_matrix.py
BOOST_AFIO_NEED_DEFINE
BOOST_AFIO_USE_BOOST_THREAD
BOOST_AFIO_USE_BOOST_FILESYSTEM
ASIO_STANDALONE
```

```
#undef BOOST_AFIO_NEED_DEFINE

#if !BOOST_AFIO_USE_BOOST_THREAD && !
BOOST_AFIO_USE_BOOST_FILESYSTEM && !ASIO_STANDALONE
# ifndef BOOST_AFIO_NEED_DEFINE_000
#  define BOOST_AFIO_NEED_DEFINE_000
#  define BOOST_AFIO_NEED_DEFINE 1
# endif

#elif BOOST_AFIO_USE_BOOST_THREAD && !
BOOST_AFIO_USE_BOOST_FILESYSTEM && !ASIO_STANDALONE
# ifndef BOOST_AFIO_NEED_DEFINE_100
#  define BOOST_AFIO_NEED_DEFINE_100
#  define BOOST_AFIO_NEED_DEFINE 1
# endif

#elif !BOOST_AFIO_USE_BOOST_THREAD &&
BOOST_AFIO_USE_BOOST_FILESYSTEM && !ASIO_STANDALONE
```

Boost.APIBind - Multi ABI 7/8

5. Remove header guards in your header file:

```
~~#ifndef SOME_GUARD_MACRO_HPP~~
~~#define SOME_GUARD_MACRO_HPP~~
...
~~#endif~~
```

Boost.APIBind - Multi ABI 8/8

6. Replace with new header guards:

```
#include "config.hpp"
#ifdef BOOST_AFIO_NEED_DEFINE
BOOST_AFIO_V1_NAMESPACE_BEGIN
...
BOOST_AFIO_V1_NAMESPACE_END
#endif
```

Ugh that’s dirty! Recommending non-standard header guards in all future Boost libraries? I don’t like it! (P.S. Neither do I!)

But how is it for users to use these libraries?

```
// test_all_multiabi.cpp in the AFIO unit tests

// A copy of AFIO + unit tests completely standalone apart from Boost.Filesystem
#define BOOST_AFIO_USE_BOOST_THREAD 0
#define BOOST_AFIO_USE_BOOST_FILESYSTEM 1
#define ASIO_STANDALONE 1
#include "test_all.cpp"
#undef BOOST_AFIO_USE_BOOST_THREAD
#undef BOOST_AFIO_USE_BOOST_FILESYSTEM
#undef ASIO_STANDALONE

// A copy of AFIO + unit tests using Boost.Thread, Boost.Filesystem and Boost.ASIO
#define BOOST_AFIO_USE_BOOST_THREAD 1
#define BOOST_AFIO_USE_BOOST_FILESYSTEM 1
// ASIO_STANDALONE undefined
#include "test_all.cpp"
#undef BOOST_AFIO_USE_BOOST_THREAD
#undef BOOST_AFIO_USE_BOOST_FILESYSTEM
```

Boost.APIBind - Quick Summary

- What I just explained looks dirty, messy and brittle, but it actually is fairly trouble free in practice and it works
  - The config.hpp preprocessor boilerplate is easily templated and is fire and forget installable
  - Only real sore point is it's too easy to break multiabi, but unit testing catches that immediately and that isn't the fault of the APIBind technique
  - And this solution is FAR simpler than ASIO's method

How do binds modularise and dependency inject a legacy monolithic namespace like std or boost?

Symbolic linking between C++ namespaces

Boost.APIBind - quick reminder

- Incomplete STL11 binds for array, atomic, chrono, condition_variable, functional, future, mutex, random, ratio, regex, system_error, thread, tuple, type_traits, typeindex
  - Can bind to either the STL or Boost
- Incomplete STL1z binds for filesystem, networking
  - Can bind to the STL (Dinkumware), Boost, or standalone ASIO the Networking TS reference impl

Boost.APIBind - how binds work 1/5

1. EXPORT: Feed your interface (header) file to the libclang tool and it spits out a bind for each of the following matching a regex:
   - Types (struct, class) and functions.
   - Template types (including template templates), and template functions.
   - enums (scoped and C form).
   - Currently missing: default template args, variables and template variables.

In <ratio> header file:

```
namespace std { template <intmax_t N, intmax_t D = 1>
class ratio; }
```

We invoke:

./genmap bind/stl11/std/ratio BOOST_STL11_RATIO_MAP_ "std::([^_][^:]*)" ratio "boost::([^_][^:]*)" boost/ratio.hpp

Bind generated by tool:

```
BOOST_STL11_RATIO_MAP_NAMESPACE_BEGIN
#ifdef BOOST_STL11_RATIO_MAP_NO_RATIO
#undef BOOST_STL11_RATIO_MAP_NO_RATIO
#else
template<intmax_t _0, intmax_t _1> using ratio = ::std::
ratio<_0, _1>;
#endif
BOOST_STL11_RATIO_MAP_NAMESPACE_END
```

2. IMPORT: Into your config.hpp file add this:

```
#define BOOST_STL11_RATIO_MAP_NAMESPACE_BEGIN namespace mylib
{
#define BOOST_STL11_RATIO_MAP_NAMESPACE_END }
// Bind std::ratio into namespace mylib
#include BOOST_BINDLIB_INCLUDE_STL11(bindlib, std, ratio)
// OR Bind boost::ratio into namespace mylib
#include BOOST_BINDLIB_INCLUDE_STL11(bindlib, boost, ratio)
```

Expands into:

```
#include "bindlib/bind/stl11/std/ratio"
```

Equals the effect of symbolically linking std::ratio into mylib namespace:

```
namespace mylib { template<intmax_t _0, intmax_t _1> using
ratio = ::std::ratio<_0, _1>; }
```

Boost.APIBind - how binds work 4/5

3. USE: No longer qualify use of ratio<Num, Den> when in namespace mylib

```
namespace mylib {
  // From auto generated bind #include file in config.hpp
  template<intmax_t _0, intmax_t _1> using ratio = ::std::ratio<_0, _1>;
  // In rest of codebase
  std::ratio<1, 2> foo;  // Use naked!
}
```

Boost.APIBind - how binds work 5/5

What have we just achieved?

*"C preprocessor controlled dependency injection of **part** of a monolithic legacy C++ namespace A into client namespace B"*

- The C preprocessor can now select what the mylib::ratio<Num, Den> symbolic link points to:
  - mylib::ratio<N, D> => std::ratio<N, D>

Notes on porting a Boost library to Boost.APIBind

My experiences porting Boost.AFIO to APIBind

Boost.APIBind - Porting a Boost library 1/5

- Despite the minimal Boost emulations provided by APIBind, this IS the same effort as porting your Boost library to a whole new platform
- Regular expression find & replace in files + regular git commits is going to be your best friend … but a slog!
- PLAN whether you'll make use of a STL11 feature dependency injected:
  - BAD IDEA: shared_ptr<>, is_constructible<> etc
  - GOOD IDEA: thread, filesystem, networking

Boost.APIBind - Porting a Boost library 2/5

- Remember you don’t HAVE to make Boost STL vs STL11 dependency injected; don’t HAVE to implement multi-abi; don’t HAVE to … etc - all this reduces the porting effort needed
  - AND you can do a port incrementally!
- Here are my experiences porting Boost.AFIO to BindLib ...

Boost.APIBind - Porting a Boost library 3/5

- AFIO is about 8k lines of library, but 18k lines including all unit testing - so small
- Approx. 60 hours to port AFIO over to BindLib - mostly tedious find & replace
- Found dozens of unexpected bugs like ABI leakage or incorrect use of STL11 or bad assumptions in unit testing

Boost.APIBind - Porting a Boost library 4/5

- Very significantly improved rigour of code quality - this was unexpected at beginning - specifying dependencies is good!
- Benefits gained is that AFIO is now very configurable and flexible for end users
  - Users download a single header only tarball or add AFIO to their git repo as a git submodule and they’re ready to go

Boost.APIBind - Porting a Boost library 5/5

- In fact most new AFIO development work exclusively uses the standalone edition - I've seen very significant productivity improvements not dealing with the slow Boost.Build et al and using a precompiled header as an AFIO "C++ module"
- Still needs Boost for docs generation
- Using APIBind as the foundation for a C++ 11 only Boost 2.0 is definitely a valid vision … (c.f. Robert's talk, also this conference)

Questions?

Boost.AFIO

Lead maintainer: Niall Douglas (me)

Boost.AFIO - what is it?

"strongly ordered portable asynchronous filesystem and file i/o extending ASIO"

Requires: Filesystem TS, Networking TS

>= Compilers*: clang 3.2, GCC 4.7, VS2013

>= OSs*: Microsoft Windows XP, Linux 2.6.32, Android, FreeBSD 10, OS X 10.5

* All these are per-commit CI tested, though not necessarily the earliest supported version. Only Microsoft Windows currently has a native asynchronous backend.

Boost.AFIO vs Boost.ASIO 1/2

- AFIO uses continuable futures instead of asio::async_result. Why deviate from ASIO?
  - For file i/o you want *strong* always-forward ordering across *all* operations, for network i/o ordering isn't as important
  - A file i/o has enormous latency variance, far more than networking
  - A file i/o takes far longer than a network i/o so overheads of a future-promise aren't so important
  - Monadic control flow and error handling makes a

Boost.AFIO vs Boost.ASIO 2/2

- The use case for async file i/o is VERY different to network i/o
  - Async file i/o is usually *slower* (warm cache ~15%) than sync file i/o
    - You choose async for *control* not performance
  - Predictability is much more important for file i/o
    - Avoiding file system race conditions
    - Identical semantics on Windows and POSIX
    - Coping well with networked file systems
    - NOT LOSING DATA!

Parts

Boost.AFIO - parts 1/3

- A thread source capable of executing closures
  - defaults to an ASIO io_service with eight kernel threads
  - each closure consumes and returns a future file handle (futures can also return exceptions of course)
  - once used **`std::packaged_task`**, now uses **`enqueued_task`** which is an intrusive

Boost.AFIO - parts 2/3

- A path class thinly wrapping **`std::filesystem::path`**
  - Main difference is on Windows where it converts to a NT kernel path, not Win32 path
- Universal **`stat_t`** and **`statfs_t`**
  - Thanks to using NT kernel API directly achieves a very close equivalence to POSIX
- A file handle, **`async_io_handle`**

Boost.AFIO - parts 3/3

- A dispatcher, **`async_io_dispatcher`**
  - takes in some thread source
  - applies default flags to operations (e.g. always fsync on file close) to save doing that per op
  - accepts batches of operations to schedule
  - issues operations according to dependency chain
  - only serialises execution twice per operation
  - just eight mallocs and frees per operation
  - Average 15 µs latency ± 0.15 µs @ 95% C.I.

Boost.AFIO - supported operations

- **Filing system race guarantees per operation**
- Batch async create/open/delete dirs, files and symlinks
- Batch async fsync, truncate, scatter/gather read/write
- Batch async enumeration of directories, filing systems and file extents
- Batch async hole punching/deallocation of file storage
- Per-handle current path retrieval, metadata retrieval, hard linking, unlinking and atomic relinking
- Portable lock files (and soon portable file locking)

Importance to C++

Boost.AFIO - what C++ 11/14 does it use?

Not much due to supporting COM/C bindings:

- C++ 11 up to what VS2013 implements
  - Makes use of some C++ 14 internally if available
- Rvalue reference support (absolutely essential due to the batch API)
- Variadic templates (since v1.3)
- Template aliasing (since v1.3, for APIBind only)

Boost.AFIO - WG21 relevance

- Intended to extend the Networking TS with async file i/o for the C++ standard library
- Intended to superset the Filesystem TS with race guaranteed filesystem operations
- Intended to seamlessly integrate with resumable functions in a future STL
  - i.e. you write C++ synchronously, but it executes asynchronously

Questions?

Boost.Fiber

Lead maintainer: Oliver Kowalke

Boost.Fiber - what is it?

“a framework for micro-/userland-threads (fibers) scheduled cooperatively”

Requires: Boost.Context, Boost.Config

>= Compilers: clang 3.4, GCC 4.9 (C++ 14)

>= OSs: Microsoft Windows (Mingw), POSIX

>= CPUs: ARM, x86, MIPS, PPC, SPARC, x64

Boost.Fiber - what is it?

Its main use case is simplification of async implementation logic for end users:

- With futures you might monadically do:

```
write(buffer1).
then([](auto f) {
   return f ? write(buffer2) : f; }).then(read
(buffer3)).get();
```

- With Fibers:

```
if(write(buffer1).get())
   write(buffer2).get();
read(buffer3).get();
```

Parts

Boost.Fiber - parts

- A fiber is a user space cooperatively scheduled thread
  - Stackful context switching (Boost.Context) + a scheduler
  - Currently about 10x faster than kernel threads
- Fibers execute *thread locally*
  - If one Fiber blocks on something e.g. a fiber:: future, only other fibers in the same thread may execute during the wait

Boost.Fiber - parts

- A fiber based replica of the STL threading primitives

  std::thread => fiber::fiber

  std::this_thread => fiber::this_fiber

  std::mutex => fiber::mutex

  std::condition_variable => fiber::condition_variable

  std::future<T> => fiber::future<T>

Importance to C++

Boost.Fiber - what C++ 11/14 does it use?

- C++ 14 only mainly due to use of execution_context in Boost.Context and deferred parameter pack expansion
  - That in turns makes use of move capturing lambdas and std::integer_sequence
  - Requiring 14 saves a lot of workaround work though
- *Probably* could support VS2015 without too much work - VS2015 only lacks generalised constexpr

Boost.Fiber - WG21 relevance

- A debate currently exists in WG21 between compiler-generated resumable functions (N4134) and “barebones only” minimal standardised coroutine support for library frameworks to build upon
  - Debate summarised in N4232 (Stackful Coroutines and Stackless Resumable Functions)
- I am personally torn between the two approaches
  - I dislike the viral keyword markup “island” in N4134
  - But I accept only the compiler can automate many optimisations e.g. fixedstack call tree leaf slicing

Questions?

Boost.DI

Lead maintainer: Krzysztof Jusiak

Boost.DI - what is it?

“provides compile time, macro free constructor dependency injection”

Requires: No Boost dependencies

>= Compilers: clang 3.4, GCC 4.9, VS2015 (C++ 14)

>= OSs: Microsoft Windows, POSIX

Boost.DI - what is it?

What is dependency injection?

- Well known design pattern in Web 2.0 service design and .NET
- As implied above, generally associated with dynamic language runtimes, not static ones like that of C++
- Yet C++ actually has a very close analogue to DI in template metaprogramming ...

Boost.DI - what is it?

```
template <typename OutputPolicy, typename LanguagePolicy>
class HelloWorld : private OutputPolicy, private LanguagePolicy
{ …

typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish>
   HelloWorldEnglish;
HelloWorldEnglish hello_world;
hello_world.run(); // prints "Hello, World!"

typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyGerman>
   HelloWorldGerman;
HelloWorldGerman hello_world2;
hello_world2.run(); // prints "Hallo Welt!"
```

Source: https://en.wikipedia.org/wiki/Policy-based_design

Boost.DI - what is it?

DI is exactly the same design pattern as template policy mix-in instantiation:

- Except it’s done at runtime, not during compile-time
- This effectively makes Boost.DI an inverted plugin or modular framework
- Extremely useful for mocking, even for comprehensive std::bad_alloc testing
- It is surprising Boost doesn’t have a library implementing this (the Strategy Pattern) before now

Parts

Boost.DI - parts

- Metaprogramming framework to assemble the appropriate make_unique<T> and make_shared<T> instances for some runtime specified dependency graph
- A registry of components and their relations
- Ecosystem of related useful utilities e.g. parser for XML config files which configure a dependency injection

Importance to C++

Boost.DI - what C++ 11/14 does it use?

- Makes heavy use of variadic templates, generic lambdas, constexpr, concept checking
- Nevertheless I can imagine this library written in 03 without drastic API changes
  - The biggest pain would be that client code would need to use Boost.Preprocessor to emulate variadic template overrides

Boost.DI - WG21 relevance

None that I am aware of

Questions?

Big Picture Answers

Why do these libraries require C++ 11 or 14? From an API perspective

Practical in C++ 03:

1. Fiber only recently was an 03 library
2. AFIO's design really only needs rvalue refs
   - Without those it would look more C array-like to implement a batch API
3. DI's design really only need variadic templates for its API to be clean
4. Http could just as easily be 03 only

Why do these libraries require C++ 11 or 14? From an API perspective

Impractical in C++ 03:

1. Hana simply isn't possible without C++ 14
   - Pushes C++ 14 implementations to their limits
2. Expected makes no sense without C++11
   - Monadic idiom makes no sense in C++ without rvalue refs, unrestricted unions and constexpr
3. Tick and Fit wouldn't have much utility without C++ 11/14
   - Concepts and traits are hard without constexpr

Is there a common theme of the most popular C++ 11/14 features used?

Universally used:

- Rvalue refs, lambdas, type inference, variadic templates, static_assert, range for, long long, defaulted and deleted functions, nullptr, STL11, generic lambdas

Somewhat used:

- Initializer lists, uniform initialisation, constexpr, class enums, overrides & final

Is there a common theme of the most popular C++ 11/14 features used?

Not common:

- Template aliases, unrestricted unions, new literals, alignment, variable templates, member initializers, inline namespaces

Never seen used not even once:

- Extern templates

Is there a common theme in choice of library design and use of third party libraries?

- Everybody avoids Boost.Test
  - assert()/static_assert() is surprisingly common
- Most avoid Boost.Build in favour of cmake
  - Usually header only with tests using only cmake, or also cmake
- Almost everybody tries to use as little Boost as possible
  - To the point of no Boost dependencies at all

Do these new libraries take notice of one another and integrate themselves well with other libraries, or are they ivory towers?

- Only AFIO presently provides a large number of build config options
  - And all of those are STL selection options
- All the libraries reviewed are ivory towers
  - Best traditions of early DIY pre-Boost!
- Indeed, **all** the libraries in the front matrix were also ivory towers
  - Good rationale for a Boost 2.0 common library

How many of these forthcoming libraries explicitly seek to contribute to future C++ standardization?

Yes:

- Fiber (async)
- AFIO (async)
- Hana (functional)
- Expected (functional)
- Range (functional)

No:

- DI
- Http
- APIBind
- Tick
- Fit

Are there techniques used in one library which would make a lot of sense to be used in another library, but for some reason are not?

- Overwhelmingly yes!
  - Best Practice of C++ 11/14 is highly uneven across libraries
  - So much so I went ahead and wrote up a Handbook of Examples of Best Practices in C++ 11/14 libraries based on the ten libraries I reviewed for this talk
- This Handbook of Example implementations is now online (link at end of these slides) and its table of contents is ...

Best C++ 11/14 Practices Handbook

1. Strongly consider using git and GitHub to host a copy of your library and its documentation
2. Strongly consider versioning your library's namespace using inline namespaces and requesting users to alias a versioned namespace instead of using it directly
3. Strongly consider trying your library on Microsoft Visual Studio 2015
4. Strongly consider using free CI per-commit testing, even if you have a private CI
5. Strongly consider per-commit compiling your code with static analysis tools
6. Strongly consider running a per-commit pass of your unit tests under both valgrind and the runtime sanitisers

Best C++ 11/14 Practices Handbook

7. Strongly consider a nightly or weekly input fuzz automated test if your library is able to accept untrusted input
8. (Strongly) consider using constexpr semantic wrapper transport types to return states from functions
9. Consider making it possible to use an XML outputting unit testing framework, even if not enabled by default
10. Consider breaking up your testing into per-commit CI testing, 24 hour soak testing, and parameter fuzz testing
11. Consider not doing compiler feature detection yourself
12. Consider having Travis send your unit test code coverage results to Coveralls.io

Best C++ 11/14 Practices Handbook

13. Consider creating a status dashboard for your library with everything you need to know shown in one place
14. Consider making (more) use of ADL C++ namespace composure as a design pattern
15. Consider defaulting to header only, but actively manage facilities for reducing build times
16. Consider allowing your library users to dependency inject your dependencies on other libraries
17. Consider being C++ resumable function ready
18. Essay about wisdom of defaulting to standalone capable (Boost) C++ 11/14 libraries with no external dependencies
19. Essay about wisdom of dependency package managers in C++ 11/14

Anything else?

- Boost which was sickly only a few years ago is now in rude health
- Seventeen new libraries since 2013 (9 passed review)
- C++ 11/14 libs look like pre-Boost libs did
  - Obvious rationale for a C++ 11 only Boost 2.0 to repeat the Boost success, but with C++ 11/14
    - I personally think that makes a C++ 11 only fully modular Boost 2.0 **highly wise**

Thank you

And let the discussions begin!

Link to Best C++ 11/14 Practices Handbook:
https://svn.boost.org/trac/boost/wiki/BestPracticeHandbook