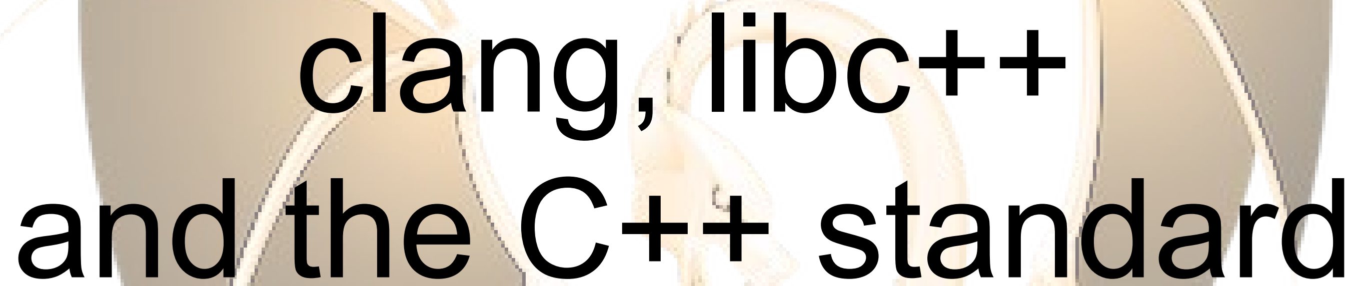
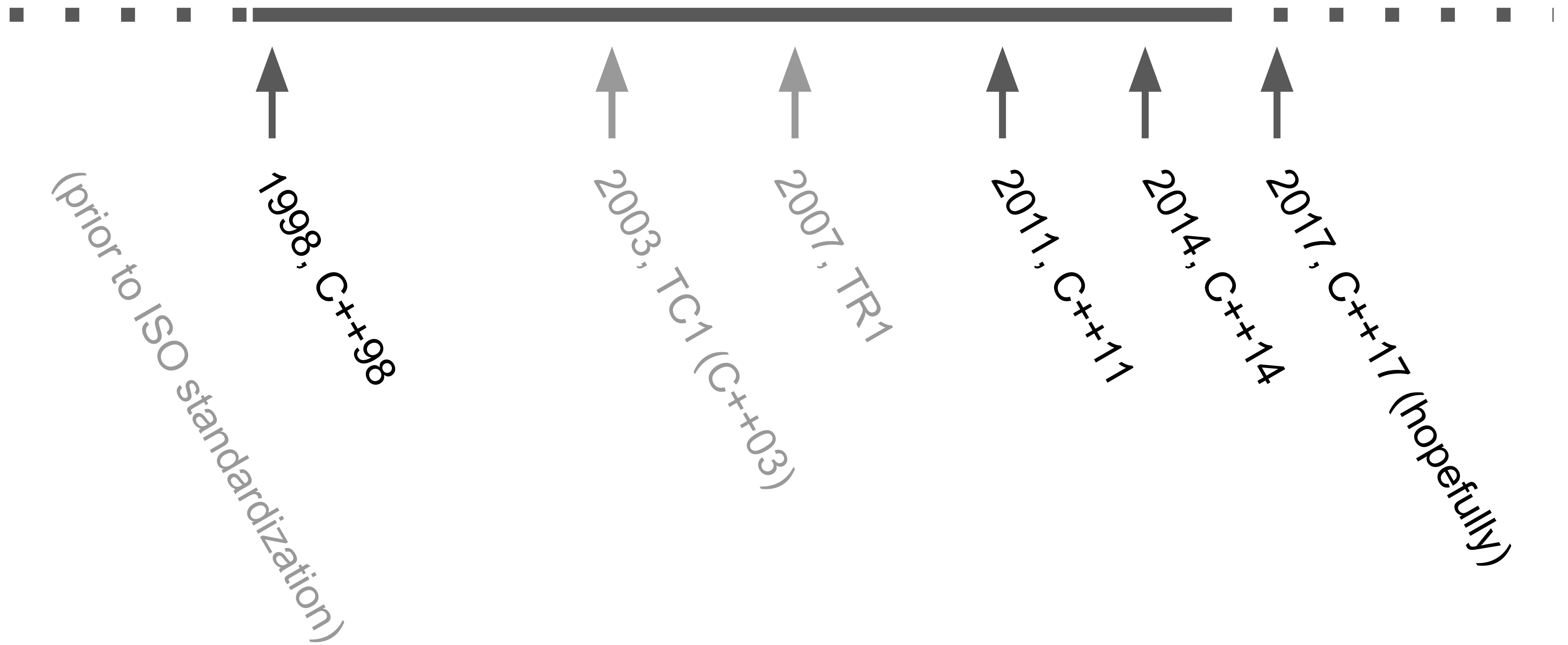
clang, libc++ and the C++ standard

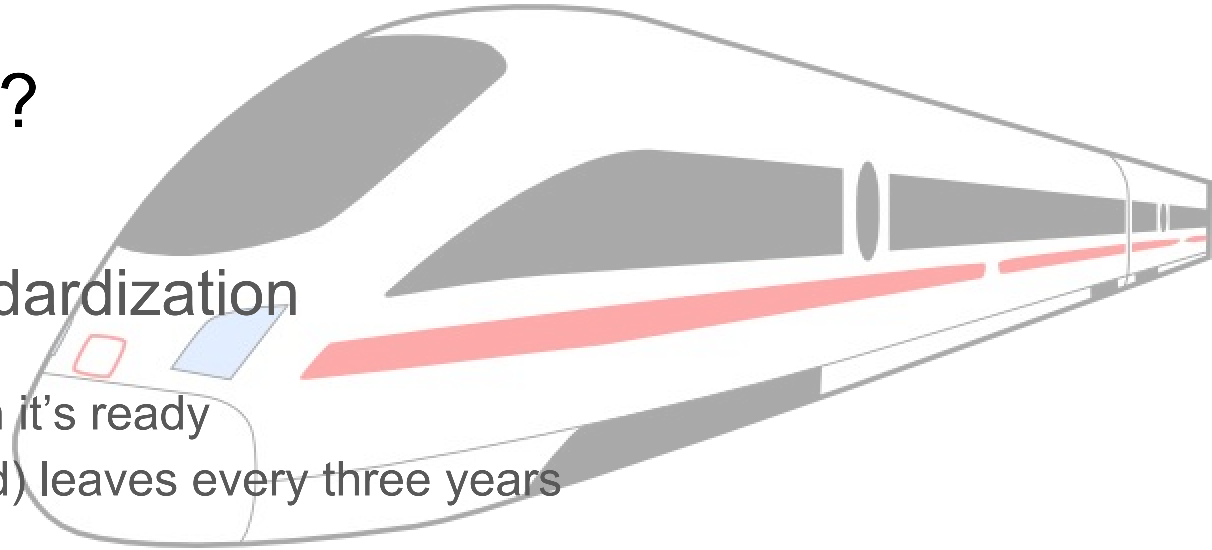


Marshall Clow
Richard Smith

A (very) brief history of ISO C++



What has changed?



“Train” model for standardization

- Ship what is ready, when it’s ready
- “The train” (new standard) leaves every three years

Standardization accelerating

- People using C++11 and C++14 and asking for more
- Many more people participating in ISO standardization
    - std-proposals@isocpp.org
    - >100 people at physical meeting
    - >100 papers per meeting
- ~10% standard growth in last meeting

Lots of exciting features here or on their way

Overview

- Rundown of new features in modern C++
  - New building blocks
  - Improvements to existing code
- Language and library impact and interactions
- Clang, libc++, LLVM implementation status

Building blocks

Better building blocks means that you can spend more time working on the parts of your program that are specific to the problem that you’re trying to solve.

Filesystem

- FileSystem was the first of the TSes.
- Based on Boost.FileSystem v3
- Boost.FileSystem has been shipping for years.
- Provides OS-independent way to iterate directory trees, get/set permissions, etc.
- Libc++ status: We have an implementation out-of tree; it will be merged into the trunk soon, and should ship as part of LLVM 3.9.

Mathematical special functions

- These are the basic building blocks of mathematical programming (so I am told :-))
- Bessel functions, Legendre polynomials, Riemann zeta, ...
- They were part of TR1, but not brought forward into C++11, but instead spun out as their own standard (N3060)
- They are being re-integrated into the C++ standard for C++17
- Libc++ status: Implementation has not yet begun

Lambda expressions

Mostly useful in concrete code, awkward in generic code

```
template<typename Container>
void reverse_sort(Container &c) {
 std::sort(c.begin(), c.end(),
           [](const ??? &a, const ??? &b) {
             return a > b;
           });
}
```

Unwieldy: `decltype(*c.begin())`

… or: `typename Container::value_type`

Lambda expressions

C++14 adds generic lambdas

```
template<typename Container>
void reverse_sort(Container &c) {
 std::sort(c.begin(), c.end(),
           [](const auto &a, const auto &b) {
             return a > b;
           });
}
```

Implicit template `operator()(const T&, const U&)`

Clang 3.4+

Lambda expressions

Lambdas can capture local variables by reference

```
void WidgetHolder::frob_all(string s) {
  widgets.for_all(
    [&s](Widget *w) { w->frob(s); });
}
```

Ideal: hold a pointer to the stack frame

Not implementable in general due to ABI issues :-(

- Clang does not yet perform this optimization in any cases
- Can use `@llvm.localescape`, `@llvm.localrecover`

Lambda expressions

Lambdas can capture local variables by copy

```
void WidgetHolder::frob_all_later(string s) {
  thread_pool.queue([=s] {
    frob_all(s);
  });
}
```

C++11: performs unnecessary copy (no way to move).

Lambda expressions

C++14: lambdas can capture local variables by move

```
void WidgetHolder::frob_all_later(string s) {
  thread_pool.queue([s = std::move(s)] {
    frob_all(s);
  });
}
```

Fully general notation: can “capture” any expression

Clang 3.4+

Lambda expressions

C++14: `this` always captured as a pointer

```
void LogMessage::log_later(string s) {
  thread_pool.queue([=] {
    log(s); // calls this->log(s)
  });       // oops, *this might be gone
}
```

C++17: explicit capture of `*this` captures a copy

```
  thread_pool.queue([=, *this] {
```

Clang patch under review

`string_view`

- "constant reference to contiguous sequence of characters"
- Grew out of use in LLVM and Google (and other places)
- Because it doesn't allocate space or copy the underlying data, creating or copying a `string_view` is much cheaper than creating a `string`.
- Almost exactly the same interface as a `const string`.
- If you do text processing, and pass `std::string`s (or `const string &`) around, you should consider `string_view` for your constant strings.
- Caveat: There's no requirement that a `string_view` be NUL-terminated.

String searching

- Part of the Library Fundamentals TS
- A general, extensible mechanism for searching text
- Shipping in boost since 2012
- The standard includes implementations of Boyer-Moore and Boyer-Moore-Horspool algorithms.
- Both can give you much better search times than the existing `std::search`, especially when looking for long patterns.
- Libc++ status: Complete in the LFTS, will be merged into namespace `std` soon.

Constant expressions

`constexpr` allows sophisticated compile-time computation

- Build lookup tables
- Static initialization of complex objects
- …

C++11 `constexpr` very limiting

- No mutable variables
- No control flow
- Awkward programming style

Constant expressions

```
constexpr bool contains(const char *str,
                        char k) {
  return !*str ? false :
         (*str == k || contains(str + 1, k));
}
```

Still Turing-complete, but just barely

- See Clang's `test/SemaCXX/constexpr-turing.cpp`
- Also `test/SemaCXX/constexpr-nqueens.cpp`
  - Only Clang can handle both!

Constant expressions

C++14 removes most restrictions

```
constexpr bool contains(const char *str,
                        char k) {
  for (; *str; ++str)
    if (*str == k) return true;
  return false;
}
```

Almost full compile-time function evaluation

Clang 3.4+

Constant expressions

Still not general enough

```
constexpr bool contains(string_view s,
                        char k) {
  return std::any_of(
    s.begin(), s.end(),
    [=](char x) { return x == k; });
}
```

- `std::any_of` not `constexpr`
  - … but more and more of the standard library is
- Lambda expression not allowed in constant expression

Constant expressions

C++17: lambdas are now allowed too

```
constexpr bool contains(string_view s,
                        char k) {
  return my_any_of(
    s.begin(), s.end(),
    [=](char x) { return x == k; });
}
```

Clang patch under review

`any`

- A “container” that can hold a single element of any type
- You can get it out using `any_cast<DestType>`, which will throw if the contained value cannot be converted to `DestType`.
- You can also get a `type_info` representing the type of the currently contained value using `type()`.
- One of the first contributions to boost (from 2001)
- Libc++ status: Complete in the LFTS, will be merged into namespace `std` soon.

`variant`

- A better union. Holds one of a set of types (which are specified at compile time).
- Another long time Boost library (from 2003)
- Is not part of Library Fundamentals
- Is not (currently) part of C++17, but there is a lot of interest in adding it to either LFTS 3 or possibly C++17
- Libc++ status: Implementation work has begun.

`optional`

- A type that can hold a single variable of a particular type, or nothing.
- One way to think of it is a container that can hold exactly zero or one items.
- Useful when you have a process that returns results, or can fail.
- Another long-time Boost library (since 2003)
- Libc++ status: Complete in the LFTS, will be merged into namespace `std` soon.

Fold expressions

C++11 added variadic templates

- Provides real type safety compared to C varargs
- Hard to use in some cases

```
template<typename T> void print(T &&t)
 { cout << t; }
template<typename T, typename ...Ts>
void print(T &&t, Ts &&...ts) {
  cout << t << " ";
  print(std::forward<Ts>(ts)...);
}
```

Fold expressions

C++17 adds ability to “fold” pack together using binary op

```
template<typename T, typename ...Ts>
void print(T &&t, Ts &&...ts) {
  // fold using operator <<
  ((cout << t) << ... << ts); // no " " :(
  // -> cout << t << t0 << t1 << ...;

  // fold using operator ,
  cout << t; ((cout << " " << ts), ...);
} // -> cout << t; (cout << " " << t0,
  //                cout << " " << t1, ...);
```

Clang 3.6+

Folding with type traits

Conjunction and disjunction traits

For example, you can use these for `any_of` or `none_of` for the types in a parameter pack:

```
template <typename... T>
struct all_arith : std::bool_constant
<std::conjunction<std::is_arithmetic<T>...>>
{};

static_assert( all_arith<int, float, short>::value,"");
static_assert(!all_arith<int, void,  short>::value,"");
```

Constexpr if

C++17 allows selective instantiation of part of a template

```
template<typename T, typename ...Ts>
void print(T &&t, Ts &&...ts) {
  std::cout << t;
  if constexpr (sizeof...(Ts) != 0) {
    std::cout << " ";
    print(std::forward<Ts>(ts)...);
  }
}
```

Unselected branch is not instantiated

Clang implementation not started

Constexpr if

Also useful for type traits

```
template<typename It>
void munge(It it, It end) {
  if constexpr (is_random_access_v<It>) {
    // algorithm 1
  } else {
    // algorithm 2
  }
}
```

Library type traits

Type traits are used in generic code to discover properties of types that are passed in.

For example, `std::vector::push_back` does different things when reallocating depending on the type in the vector.

- If the type `is_trivially_copyable`, then it will use memcpy, rather than a copy/move constructor.
- If the type `is_nothrow_move_constructible`, then it will move the elements.
- Otherwise it will copy the elements.

Library type traits - value versions

For every is_foo<T> type trait, also provide is_foo_v<T>

- Shorthand for is_foo<T>::value
- Uses new C++14 language feature: variable templates
  - Obvious syntax and meaning:

```
template<typename T> const T pi
  = T(3.141592…);
```

Template argument deduction for constructors

```
tuple<int, double, string> t =
     {1, 2.5, to_string(1e100)};
```

Redundant and error prone

C++17 will (probably) allow

```
  tuple t = {1, 2.5, to_string(1e100)};
```

Template argument deduction for constructors

Explicit deduction guides

```
template<typename ...T>
  tuple(T ...) -> tuple<T...>;
```

Guides may also be inferred from constructors

- Though there are some problematic cases

Approved by Evolution, not voted into standard yet

Prototyped in Clang

Existing code gets better

- Newer versions of C++ bring improvements even for unmodified code
  - C++11: move semantics
  - C++11: implicit move for return value
  - C++14: optimizable allocation
  - C++14: sized deletion
  - C++17: guaranteed copy elision / RVO

Existing code gets better: optimizable allocation

- C++14 allows allocation and deallocation of the form
  `p = new Foo` and `delete p`
  to be merged or eliminated
- LLVM has been able to eliminate such allocations (as a non-conforming extension) for years
  - Now only does so when C++ standard allows
  - `declare void @_ZdlPv(i8*)` **`nobuiltin`**
  - `delete p` => `call @_ZdlPv(%p)` **`builtin`**
  - `::operator delete(p)` => `call @_ZdlPv(%p)`

Clang 3.4+

Existing code gets better: optimizable allocation

What about `std::allocator`?

- Add `__builtin_operator_new`, `__builtin_operator_delete` to Clang
- Use them from `std::allocator`

Existing code gets better: optimizable allocation

- C++14 calls a sized form of operator delete when the size is statically known (it usually is!)
- Allows more efficient deallocation

Existing code gets better: optimizable allocation

Library adds functions:

```
void operator delete(void *p, size_t size);
void operator delete[](void *p, size_t size);
```

Downsides:

- ABI break
- libc++ implementation can't easily provide speedup (no standard malloc interface for sized free)

Disabled by default in Clang, use `-fsized-deallocation`
Will be re-enabled once updated ABI libraries are common

Existing code gets better: guaranteed copy elision

```
string f(string a, string b) {
  return a + b;
}
string x = f("foo", "bar");
```

How many temporary objects here?

Existing code gets better: guaranteed copy elision

```
string f(string a, string b) {
  return a + b;
}
string x = f("foo", "bar");
```

Old (C++98 - C++14) model:

- Temporary objects created immediately
- Compiler permitted to elide copy from temporary object
  - C++11 onwards: *move* from temporary object
    - Still not free
  - Elided by Clang, GCC, … but not portable

Existing code gets better: guaranteed copy elision

```
string f(string a, string b) {
  return a + b;
}
string x = f("foo", "bar");
```

New (C++17) model:

- Wait and see how the expression is used
- If a “temporary expression” is used to initialize a variable, do not create the temporary
- Clang implementation requires *deleting* code

What’s *not* in C++17

Train model: features that are not ready yet don’t board

Big upcoming features not in C++17:

- Concepts and ranges (new standard library)
- Coroutines
- Modules
- Networking

Each is in, or becoming, a Technical Specification

Concepts

- Concepts defined as predicates over types:

```
template<typename T> concept bool Iterator =
  requires (T t) { ++t; *t; t != t; };
```

- Allow `template<Iterator T>`, where `Iterator` is a concept (type theory: a kind, or “type of type”)
  - Shorthand for

```
template<typename T> requires Iterator<T>
```

- Goal: improved diagnostics:

```
error: T (aka 'int') is not an Iterator
note: expression *t is not valid
```

- Goal: improved expressivity

Concepts

Example:

```
template<typename T>
requires LessThanComparable<T>
const T& min(const T& x, const T& y) {
    return (y < x) ? y : x;
}
```

where `LessThanComparable` is a concept defined elsewhere.

Concepts

Not quite ready for C++17:

- Syntax too experimental
    - 5 new syntaxes for declaring a function template
    - no clean syntax for declaring a concept
- Some semantic problems
    - Exponential-time algorithm for ordering templates
- More “bake time” desired by users

Concepts likely for C++20

Clang implementation in progress

Ranges

The basic abstraction in the STL is the iterator. This allows us to write N algorithms that work with M containers, instead of N * M algorithms (one version for each container)

Ranges are a different abstraction. Rather than passing around a [begin, end) pair of iterators, you work in terms of a range object. Ranges are designed to be composable.

The Ranges proposal depends on Concepts.

See Eric Niebler’s C++Now / CppCon talk for an example of using ranges in practice.

Coroutines

First-class support for function suspension and resumption

```
generator<int> make_squares() {
  for (int n = 0; n <= sqrt(INT_MAX); ++n)
    co_yield n * n;
}

future<string> read_from_network() {
  auto socket = co_await
      http::connect("http://llvm.org", 80);
  auto val = co_await socket.read();
  co_return val;
}
```

Coroutines

Model:

- Function signature (return type) in control of behavior
- Allocate separate storage for coroutine state
    - Elidable allocation
- Parts of stack frame live in coroutine state instead
- Opaque handle type allows resumption of coroutine

See http://wg21.link/p0057r2 for complete specification

Coroutines

Implementation:

- Clang supports parsing syntax, `-fcoroutines-ts`
- Prototype LLVM implementation due to Gor Nishanov
  - Approach: “fork”-style intrinsic spawns coroutine
    - Other intrinsics provide support for resumption, and for eliding allocation, copies, ...
  - Modeled as single function for most optimizations
  - Late lowering into separate launch / resume / destroy functions and a switch
- Proposal for IR change heading to llvm-dev@ soon

Modules

`#include` is a bad way to expose the interface of a library

- Brittle
- Slow

Idea: compile library interface separately, perform semantic import when desired

Modules TS

C++ modules proposal:

```
module X;

import module Y;

export void f();
export struct S { /* ... */ };
```

Implementation can be in module interface or a separate file

Modules in Clang

“Transitional” model

- No explicit syntax
- “Module map” file specifies which header files form which modules
- `#include` implicitly mapped to module import
- See Doug Gregor’s 2012 LLVM DevMtg talk
- Compiles large (XXX MLoC) codebase, 50% compile time reduction from modularizing key ~10% of libraries

Coming to Clang soon:

- Real syntax from Modules TS

Networking

Based on Boost.ASIO

A generalized framework for handling asynchronous IO, with a set of portable networking primitives.

This will be an ongoing project; there’s lots of work here.

Libc++ implementation has begun

Clang and libc++ support for C++17

Clang: http://clang.llvm.org/cxx_status.html

Libc++: http://libcxx.llvm.org/cxx1z_status.html

Much still to do; integrating the TSes is a lot of work.

Lots already done, despite features being added 2 weeks ago

Q&A