C++14 (Preview)

Alisdair Meredith, Library Working Group Chair

A Quick Tour of the Sausage Factory

- A Committee convened under ISO/IEC so multiple companies can co-operate and define the language
- Google ‘wg21’ to find the official site (usually top hit)
- Community site for publicizing activity, sharing the latest information, and encouraging wider involvement
  - isocpp.org

A Quick Tour of the Sausage Factory

- 1998 first standard (7 years, 21 meetings)
- 2003 TC1 (5 years, 10 meetings)
- 2011 second standard (8 years, 21 meetings)
- 2013 ongoing work ... (2 years, 4 meetings so far)

A Quick Tour of the Sausage Factory

- August 2011 : Bloomington, Indiana
  - No new work in advance, while standard under ballot
  - A chance to reflect and consider future goals
  - Start work on the freshly mounting issues lists...
  - Look more actively at new TR work (now called TS)

A Quick Tour of the Sausage Factory

- February 2012 : Kona, Hawaii
  - Significant new membership
  - Plan schedule for next project
    - New standards in (minor) 2017 and (major) 2022
  - Create first 5 study groups
  - Resolve first defect reports

A Quick Tour of the Sausage Factory

- November 2012 : Portland, Oregon
  - Study groups convene (and double)
  - Library and Core want to publish defect reports before 2017, but TC asks ‘fix or new feature’
    - Plan 2014 standard with tiny extensions
    - Requires all ballots compete in 2013
- Ballot ISO to create Filesystem TS

A Quick Tour of the Sausage Factory

- April 2013 : Bristol, UK
  - Library grows a separate evolution working group
  - Significant new features start to arrive
  - Send new standard to ISO for ballot
  - Ballot to create a ‘Concepts Lite’ TS
  - Ballot to create an initial ‘Networking’ TS

Study Groups

- SG1 Concurrency
- SG2 Modules
- SG3 Filesystem
- SG4 Networking
- SG5 Transactional memory
- SG6 Numerics
- SG7 Reflection
- SG8 Concepts
- SG9 Ranges
- SG10 Portability
- SG11 Database access
- SG12 Undefined behavior

What Failed To Make The Cut?

- Concepts lite
  - moving to a separate TS
- digit separators
  - most awkward bikeshed to paint yet
- Splicing `map`s and `set`s
- Literals for `std::complex`
- Remove deprecated `operator++` for `bool`

Key New Language Features

- Polymorphic lambda expressions
- Runtime-length arrays on the stack
- Variable templates
- Deduced return type for functions
- Binary integer literals

Key Language Fixes/Tweaks

- Generalized constexpr
  - constexpr members no longer implicitly const
- Enhanced lambda capture
- Contextual conversion in more contexts
- Member initializers for aggregates

New Library Components

- `dynarray`
- `optional`
- shared locks
- quoted strings
- `integer_sequence`

Small Library Extensions

- `exchange`
- `make_unique`
- `cbegin/rbegin` as free functions
- literals for `string` and `chrono` (but not `complex`)
- ‘diamond’ operator functors
- traits aliases

Small Library Extensions

- index a `tuple` by type
- robust algorithms with two ranges
- heterogeneous look for `map` and `set`
- `hash` for `enum`s
- `result_of` is SFINAE friendly
- null iterators are well defined values

Completing the set...

- 187 Core defects resolved
- 76 Library defects resolved
- 3 new TS documents under way
  - Filesystem library
  - Networking library (phase 1)
  - Concepts lite

New Features in Detail

- Lambdas
- Runtime arrays
- Constexpr
- Deduced return type for functions
- literals
- std::optional

Lambda

- Currently must name the type of each argument, no way to deduce
- Captures are either a copy or a reference, no ability to capture a move-only type like `unique_ptr`

Lambda

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

Lambda : C++11

```
template <typename Container, typename Pred>
void reverse_sort(Container& c, Pred fn) {
    using Param = typename
                  std::iterator_traits<Container::iterator>::value_type;

    std::sort(c.begin(), c.end(),
              [fn](Param const &a, Param const &b)
                  { return fn(b, a); }
             );
}
```

Lambda : prefer bind?

```
template <typename Container, typename Pred>
void reverse_sort(Container& c, Pred fn) {

    std::sort(c.begin(), c.end(), std::bind(fn, _2, _1));


}
```

Polymorphic Lambda

```
template <typename Container, typename Pred>
void reverse_sort(Container& c, Pred fn) {

    std::sort(c.begin(), c.end(),
              [fn](auto const &a, auto const &b)
                  { return fn(b, a); }
             );
}
```

Lambda Capture

- Example from the standard:

```
int x = 4;

auto y = [&r = x, x = x+1]()->int {
            r += 2;
            return x+2;
          }();
```

Lambda Capture

- Example from the standard:

```
int x = 4;

auto y = [&r = x, x = x+1]()->int {
            r += 2;
            return x+2;
         }();
```

- Updates ::x to 6, and initializes y to 7

Lambda Capture

```
std::mutex mtx;
std::unique_lock<std::mutex> lock{mtx};

std::sort(container.begin(), container.end(),
          [lock = move(lock)](auto &a, auto &b)
          { return a < b; }
         );
```

Runtime Arrays

- language and library facilities for the same purpose
  - language facility guaranteed on stack
  - library optimistic, relies on unspecified compiler optimizations kicking in
- language facility subject to several unique restrictions, library facility is a regular class template
- only the library facility “knows” its size

Runtime Arrays : language

- Like an array with restrictions
- Cannot copy the array by value
- Cannot find the type, e.g., `declype` or `typeid`
- Cannot find the size of the object, e.g., `sizeof`
- Cannot bind a reference to such an array
- Cannot take the address of such an object
- Cannot be used as data member of a class

Runtime Arrays : library

- Like `std::array` with size given at construction
- Allocate on stack only if the `dynarray` is a local variable on the stack
- Class is neither copyable nor movable, otherwise use like a regular object
- Can be used as data member in other structures and captured by lambda expressions
- Supports `begin/end` and `for` loops

Runtime Arrays : example

```
template <typename Iterator>
auto median(Iterator first, Iterator last)
  -> decltype(*first)
{
   assert(first != last);

   auto length = std::distance(first, last);
   decltype(*first) data[length];
   auto ptr = std::addressof(data[0]);

   std::copy(first, last, ptr);
   auto result = ptr + length/2;
   std::nth_element(ptr, result, ptr + length);
   return *result;
}
```

Runtime Arrays : example

```
template <typename Iterator>
auto median(Iterator first, Iterator last)
  -> decltype(*first)
{
   assert(first != last);

   auto length = std::distance(first, last);
   std::dynarray<decltype(*first)> data{length};
   auto ptr = begin(data);

   std::copy(first, last, ptr);
   auto result = ptr + length/2;
   std::nth_element(ptr, result, end(data));
   return *result;
}
```

Constexpr

```
auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}
    return cursor - str;
}
```

Constexpr

```
constexpr auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}

    return cursor - str;
}
```

Constexpr

```
constexpr auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}

    return cursor - str;
}
```

Constexpr

```
constexpr auto strlen(char const *str) -> size_t {
    return !str or !*str
           ? 0 : 1 + strlen(str + 1);
}
```

- inline function
- function body must be a single return statement
- may not modify values or have other side effects

Constexpr

```
auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}
    return cursor - str;
}

constexpr auto strlen_c(char const *str) -> size_t
{
    return !str or !*str
           ? 0 : 1 + strlen_c(str + 1);
}
```

Constexpr : C++11

```
constexpr auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}

    return cursor - str;
}
```

Constexpr : C++14

```
constexpr auto strlen(char const *str) -> size_t {
    if (!str or !*str) return 0;

    auto cursor = str;
    while (*++cursor) {}

    return cursor - str;
}
```

Deduced Return Type

- C++11: Lambda expressions can deduce return type
- C++14: Regular functions can also deduce return type
- **Not** restricted to a single ‘return’ statement
- Multiple returns must have the same returned type
- May be forward-declared
- result type is determined after opportunity for SFINAE

Deduced Return Type

```
constexpr auto strlen(char const *str) -> unsigned int {
    if (!str) return 0;

    unsigned int result{0};
    while (*str) {
        ++result;
        ++str;
    }
    return result;
}
```

Deduced Return Type

```
constexpr auto strlen(char const *str) -> unsigned int {
    if (!str) return 0u;  // consistent return types

    unsigned int result{0};
    while (*str) {
        ++result;
        ++str;
    }
    return result;
}
```

Deduced Return Type

```
constexpr auto strlen(char const *str) {
    if (!str) return 0u;

    unsigned int result{0};
    while (*str) {
        ++result;
        ++str;
    }
    return result;
}
```

Deduced Return Type

```
template <typename T>
auto identity(T const & value) {
    return value;
}
```

Result type is `T`, returning a copy of the argument

Deduced Return Type

```
template <typename T>
auto identity(T const & value) -> auto {
    return value;
}
```

Result type is `T`, returning a copy of the argument
Equivalent to previous slide

Deduced Return Type

```
template <typename T>
auto identity(T const & value) -> decltype(auto){
    return value;
}
```

Result type is `T const &`

Literals

- Binary integer literals are implemented by the compiler
  - indicated by a prefix, like octal and hex
- Library literals are functions in namespaces
  - must `include` appropriate header
  - enabled by a `using` directive
  - ‘s’ can mean seconds or strings
    - disambiguated by the argument

Literals

```
#include <chrono>
#include <string>
using namespace std::literals;
auto a = 0b101010;  // int 42
auto b = 0b001101L; // long 13
auto c = "Hello"s;  // string
auto d = L"World"s; // wstring
auto e = 42s;       // seconds
auto f = 42ms;      // milliseconds
auto g = 42us;      // microseconds
auto h = 42ns;      // nanoseconds
auto i = 42min;     // minutes
auto j = 42h;       // hours
```

Literals

- inline namespaces give finer-grained control
  - using namespace std::literals::string_literals;
  - using namespace std::literals;
  - using namespace std;
- Choose how much of the standard library you want to import through the ‘using’ directive

std::optional

- A nullable type with value semantics
  - Nullable in C#
  - Modelled as a smart pointer
    - pointer is to internal data, no dynamic allocation
  - Usage example : retrieving a record from a database

std::optional : controversy

- what is the semantic of `optional<T &>` ???
  - `optional<reference_wrapper<T>>`
  - `T * const` with implicit dereference
    - references are never optional, just as no pointers-to-references
- comparison operators
  - delegate to `optional` **`operator`**`==` and **`operator`**`<`
  - delegate to `std::less`, `std::greater` etc?

Streaming example

```
auto in = "Hello world!"s;

stringstream ss;
ss << in;

decltype(in) out;
ss >> out;

cout << out << endl; // ???
```

Streaming example

```
auto in = "Hello world!"s;

stringstream ss;
ss << in;

decltype(in) out;
ss >> out;

cout << out << endl; // ???
```

Streaming example

```
auto in = "Hello world!"s;

stringstream ss;
ss << in;

decltype(in) out;
ss >> out;

cout << out << endl; // Hello
```

Quoted strings example

```
auto in = "Hello world!"s;

stringstream ss;
ss << quoted(in);

decltype(in) out;
ss >> quoted(out);

cout << out << endl; // Hello world!
```

Quoted strings

- **Not** designed for displaying to users
- Allows round-trip writing and reading of text containing whitespace to streams
- Allows for customizing the escape characters
- Default is to use quotes, escaped by a backslash

New Language Features

- Variable templates

```
template <typename T> T const pi;
template <float> float const pi = 3.14159f;
template <double> double const pi = 3.14159268;
auto value = pi<double>;
```

- Contextual conversion in more contexts

```
switch (myIntValue) { ...
```

- Member initializers for aggregates

```
struct Point {
    int x = -99;
    int y = -99;
};
```

Traits Extensions

- `result_of` is SFINAE friendly

```
template <typename Fn, typename Arg>
auto call(Fn f, Arg && x) -> typename result_of<Fn(Arg)>::type;
```

- traits aliases

```
template <typename T>
using add_const_t = typename add_const<T>::type;
```

- integer_sequence

```
template <typename Fn, typename ... Args>
auto call(Fn && fn, tuple<Args...> const & args)  // ???
```

Traits Extensions

- `result_of` is SFINAE friendly

```
template <typename Fn, typename Arg>
auto call(Fn f, Arg && x) -> typename result_of<Fn(Arg)>::type;
```

- traits aliases

```
template <typename T>
using add_const_t = typename add_const<T>::type;
```

- integer_sequence

```
template <typename Fn, typename ... Args, size_t ... Index>
auto call_impl(Fn && fn, tuple<Args...> const & args)
{
   return fn(get<Index>(args)...);
}

template <typename Fn, typename ... Args>
auto call(Fn && fn, tuple<Args...> const & args)
```

Traits Extensions

- `result_of` is SFINAE friendly

```
template <typename Fn, typename Arg>
auto call(Fn f, Arg && x) -> typename result_of<Fn(Arg)>::type;
```

- traits aliases

```
template <typename T>
using add_const_t = typename add_const<T>::type;
```

- integer_sequence

```
template <typename Fn, typename ... Args, size_t ... Index>
auto call_impl(Fn && fn, tuple<Args...> const & args, index_sequence<Index...>)
{
   return fn(get<Index>(args)...);
}

template <typename Fn, typename ... Args>
auto call(Fn && fn, tuple<Args...> const & args)
{
   return call_impl(fn, args, make_index_sequence<sizeof...(Args)>{});
}
```

Library Extensions

- `exchange`

```
int x = 13;
int y = exchange(x, 42);
assert(42 == x);
assert(13 == y);
```

- `make_unique`

```
auto ptr = make_unique<complex<double>>(3.14, 2.78);
```

- `cbegin/rbegin` as free functions
- `hash` specialized for enums

Library Extensions

- ‘diamond’ operator functors

```
template <typename T = void>
struct less;

template <>
struct less<void> {
    template <typename T, typename U>
    bool operator()(T && t, U && u) const {
        return forward<T>(t) < forward<U>(u);
    }
};

map<Key, Value, less<>> m;
```

Library Extensions

- index a `tuple` by type
```
get<int>(make_tuple(3.14, 42, "string"s));
```
- robust algorithms with two ranges
```
std::equal(first1, last1, first2, last2);
```
- heterogeneous look for `map` and `set`
```
map<string, int>{}.find("literal");
```
- default constructed iterators are well defined values, past-the-end of the same empty range
- shared locks // examples for another day ...

Availability

- gcc 4.8 : functions with deduced returns
- clang 3.3 : binary literals
- clang 3.3 : member initializers for aggregates

Availability : gcc 4.9 trunk

- functions with deduced returns
- runtime arrays
- extended lambda capture
- binary literals
- contextual conversions
- polymorphic lambda
- variable templates

Availability : clang 3.4 trunk

- binary literals
- member initializers for aggregates
- functions with deduced returns
- contextual conversions
- runtime arrays
- polymorphic lambda
- generalized constexpr