# The C++14 Standard Library

**Jonathan Wakely**
**Red Hat**

# Overview

20. General Utilities (+10)

21. Strings (+1)

22. Localization (+1)

23. Containers (+1)

24. Iterators (+1)

25. Algorithms (+1 / -1)

26. Numerics (+1)

27. Input/output (+1/ -1)

30. Thread support (+1)

C++11 FDIS was N3290, the C++14 draft is N3936

# General Utilities

`exchange<>()`

`get<>()` tuple elements by type

`integer_sequence<>`

`allocator<>::propagate_on_container_move_assignment`

`make_unique<>()`

`greater<>`, `less<>` etc.

Improved `integral_constant<>`

SFINAE-friendly `result_of<>`

Transformation Traits aliases

Suffixes for `duration<>` literals

# General Utilities: `exchange`

The non-atomic equivalent of `atomic_exchange`

Assigns a new value to an object and returns the old value

Example:

```
void Thing::stuff()
{
  if ( !std::exchange(m_initialized, true) )
    do_initialization();
  // do other stuff
}
```

# General Utilities: `exchange`

The non-atomic equivalent of `atomic_exchange`

Assigns a new value to an object and returns the old value

Another example:

```
void unique_ptr<T>::reset(pointer new_ptr)
{
  if ( auto old_ptr = std::exchange(m_ptr, new_ptr) )
    m_deleter(old_ptr);
}
```

# General Utilities: exchange

```
template <class T, class U = T>
  T exchange(T& obj, U&& new_val);
```

*Effects:* Equivalent to:

```
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
```

# General Utilities: `get`

C++11 provides the `get<N>()` function for retrieving a tuple element by its index in the tuple

C++14 adds new overloads to retrieve an element by type:

```
tuple<int, double, string> t{ 1, 2.0, "three" };

auto& d = std::get<double>(t);

assert( &d == &std::get<1>(t) );
```

The tuple must have exactly one element of that type

Overloaded for `pair` too

# Utilities: `integer_sequence`

```
template<class T, T...> struct integer_sequence;

template<class T, T N> using make_integer_sequence
    = integer_sequence<T, 0, 1, 2, 3, ... N-1>;

template<size_t... I> using index_sequence
  = integer_sequence<size_t, I...>;

template<size_t N> using make_index_sequence
    = make_integer_sequence<size_t, N>;

template<class... T> using index_sequence_for
    = make_index_sequence<sizeof...(T)>;
```

# Utilities: `integer_sequence`

Used inside variadic templates to expand elements of a tuple (or tuple-like object) into a list of arguments

```
template<class F, class Tuple>
 decltype(auto)
 apply(F f, Tuple&& t)
 {
   return f( std::get< ??? >(t) );
 }
```

We want a pack expansion like `std::get<N>(t)...` where `N` is a parameter pack of indices

# Utilities: `integer_sequence`

```
template<class F, class Tuple>
decltype(auto)
apply(F&& f, Tuple&& t)
{
  using TupleSize = tuple_size<decay_t<Tuple>>;
  using Indices = make_index_sequence<TupleSize::value>;

  return apply_impl(f, t, Indices{});
}

template<class F, class Tuple, size_t... I>
decltype(auto)
apply_impl(F&& f, Tuple&& t, index_sequence<I...>)
{
  return f(std::get<I>(t)...);
}
```

# Utilities: `allocator::pocma`

```
template<class T>
class allocator
{
public:
  using propagate_on_container_move_assignment
    = true_type;
  // ...
};
```

# Utilities: `make_unique`

In C++11 we have `shared_ptr` and `make_shared`:

```
func( make_shared<X>(aargo, bargo, cargo),
      make_shared<Y>(dargo, eargo, fargo) );
```

But there is no equivalent for `unique_ptr`

```
func( unique_ptr<X>{new X{aargo, bargo, cargo}},
      unique_ptr<Y>{new Y{dargo, eargo, fargo}} );
```

The `unique_ptr` version is not exception-safe

# Utilities: `make_unique`

For creating a single object:

```
unique_ptr<X> p = make_unique<X>(argo, bargo, cargo);
```

For creating arrays of objects:

```
unique_ptr<Y[]> p = make_unique<Y[]>(n);  // new Y[n]()
```

# Utilities: diamond operators

C++ has always provided functors for the built-in operators

`less<>` is widely-used as the default comparison function for sorted containers

Other operators such as `plus`, `not_equal_to`, `bit_xor` also available

They are all class templates taking a single argument that must be provided:

```
std::sort(begin, end, std::greater<ValueType>{});
```

# Utilities: diamond operators

C++14 gives them a default template argument, so that `greater<>` means `greater<void>`, and defines specializations for `void` which accept any argument types, deduced as needed:

```
template<> class greater<void> {
  template<class T, class U> decltype(auto)
  operator()(T&& t, U&& u) const
  { return std::forward<T>(t) > std::forward<U>(u); }

  using is_transparent = unspecified;
};

std::sort(begin, end, std::greater<>{});
```

# Utilities: `integral_constant`

```
template <class T, T v>
struct integral_constant {

  static constexpr T value= v;

  using value_type = T;

  using type = integral_constant<T,v>;

  constexpr operator value_type() { return value; }


  constexpr value_type operator()() { return value; }

};
```

# Utilities: `result_of` ♥ SFINAE

In C++11 you got a compile-time error if you tried to instantiate `result_of` with invalid arguments, so `decltype` was often preferred

C++14 says that `result_of<F(A...)>::type` is only defined when the type `F` is callable with arguments of type `A...`

```
template<typename F, typename... Args>
typename std::result_of<F(Args...)>::type
call(F f, Args&&... args)
{ return f( std::forward<Args>(args)... ); }
```

# Utilities: Traits Aliases

The C++11 type transformation traits are very useful, but the syntax is verbose and ugly:

```
typename remove_const<T>::type
```

C++14 adds alias templates for all the traits with a nested `type` member:

```
template<typename T>
  using remove_const_t = typename remove_const<T>::type;
```

So now you can just use `remove_const_t<T>`

# Utilities: Duration literals

C++11 added User-Defined Literals to the language, which are now used in three places in the standard library

The first set of suffixes are defined in the `<chrono>` header to simplify creating durations, so that:

```
cv.wait_for( chrono::milliseconds(150) );
```

becomes:

```
cv.wait_for( 150ms );
```

# Utilities: Duration literals

The available suffixes are:

```
h   // chrono::hours
min // chrono::minutes
s   // chrono::seconds
ms  // chrono::milliseconds
us  // chrono::microseconds
ns  // chrono::nanoseconds
```

# Utilities: Duration literals

```
constexpr std::chrono::microseconds
operator ""_µs (unsigned long long usec)
{ return std::chrono::microseconds{ usec }; }

constexpr std::chrono::duration<long double, std::micro>
operator ""_µs(long double usec)
{
  using D
    = std::chrono::duration<long double, std::micro>;
  return D{ usec };
}

using namespace std::literals;
assert( 5_µs == 5us );
assert( 3.14_µs == 3.14us );
```

# Utilities: Aside

I wish new extensions had been added to a new header rather than just adding more and more to `<utility>`

We could have another header containing the 2nd set of general utilities

```
#include <utilitu>
```

This is just a silly pun on "utility two" that's also a palindrome!

# Utilities: Further Aside

I also wish we had a header of utilities for working with files

```
#include <futility>
```

# Strings: string literals

There are literal suffixes for creating the standard `basic_string` specializations:

```
using namespace std::literals;
auto s   = "narrow"s; // std::string
auto s16 = "utf-16"s; // std::u16string;
auto s32 = "utf-32"s; // std::u32string;
auto ws  =  L"wide"s; // std::wstring;
```

The `s` suffix for strings is not ambiguous with the suffix for creating `chrono::seconds` values

# Localization: `isblank`

C99 added the `isblank` character classification function, for testing whether a character is one of a locale-specific whitespace characters (in the `"C"` locale that is `' '` or `'\t'`). C++14 adds:

```
int isblank(int);

template<typename charT>
  bool blank<charT>(charT, const locale&);
```

and a new bitmask element:

```
ctype_base::blank
```

# Containers: lookup

The standard associative containers now support "heteregeneous lookup"

```
map<string, int> m{ {"a", 1}, {"b", 2} };
auto it = m.find("b");
```

Which is a fancy way of saying you can search using a key of a different type to the keys stored in the container

The `find` call above calls

```
  map::find(const key_type&)
```

which requires constructing a temporary `string`

# Containers: lookup

C++14 adds new templated overloads of `find` (and `count`, `lower_bound`, `upper_bound` and `equal_range`) which are function templates:

```
template<typename K>
  iterator find(const K& x);
template<typename K>
  const_iterator find(const K& x) const;
```

*But* these new overloads are only present when the container's comparison function has a nested `is_transparent` type

# Containers: lookup

The new lookup functions are disabled by default to avoid causing performance regressions in existing code, so you have to opt-in by explicitly using a custom comparison functor that tells the container to enable the new functions

The "diamond operators" such as `less<>` all define the `is_transparent` typedef, so they can be used to enable the new lookup functions

```
map<string, int, less<>> m{ {"a", 1}, {"b", 2} };
auto it = m.find("b");

template<typename Key, typename Val>
  using Map = std::map<Key, Value, std::less<>>;
```

# Iterators: Forward Iterators

C++14 gives a slightly stronger new guarantee for forward iterators:

> The domain of == for forward iterators is that of iterators over the same underlying sequence. **However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type.** [*Note:* value initialized iterators behave as if they refer past the end of the same empty sequence — *end note*]

# Algorithms: moar robust!

In previous versions of C++ the non-modifying sequence algorithms `equal`, `mismatch` and `is_permutation` take a pair of iterators denoting the first range, and a second iterator denoting the *start* of the second range

It is the caller's responsibility to ensure that the second range is at least as long as the first

If the second range is shorter, the library will walk off the end of that second range

If the second range is longer, the end of the range won't be checked

# Algorithms: moar robust!

C++14 adds new overloads of `equal`, `mismatch` and `is_permutation` which take two iterators for the second range

These overloads are more explicit about the length of the two ranges that the user *wants* to compare, and will stop checking at the end of the shorter range

For non-random access iterators it is more efficient for the algorithm to do the range checking on-the-fly, rather than for the user to have to find the length of the shorter range

# Algorithms: less `rand`!

C and C++ provide the `rand()` function to produce pseudo-random numbers, but the implementation is completely unspecified and so the quality of the pseudo-random numbers it produces varies between platforms and is unreliable.

A common use of `rand()` is to simply apply the modulo operator to obtain a number in a desired range: `rand() % count` — even if the C library's `rand()` implementation is excellent and produces good random numbers **this does not produce a uniform distribution!**

# Algorithms: less `rand`!

C++11 provided several new random number facilities in `<random>`, with clearly defined, portable semantics and tools for correctly obtaining uniform distributions for a given range, and these should be preferred to `rand()`

C++14 strengthens a note discouraging the use of `rand()`:

> [Note: The random number generation (26.5) facilities in this standard are often preferable to `rand`, **because randâ€™s underlying algorithm is unspecïﬁed. Use of** `rand` **therefore continues to be nonportable, with unpredictable and oft-questionable quality and performance.** â€” end note]

# Algorithms: less `rand`!

C++14 also deprecates the `random_shuffle` algorithms, because the three-argument overload uses an unspecified PRNG, and at least one implementation simply uses `rand() % count` (oops!)

The C++11 `shuffle` algorithm, which uses the new `<random>` facilities, should be used instead.

# Numerics: `complex` literals

There are literal suffixes for creating complex numbers:

```
using namespace std::literals;
auto a = 1i;      // complex<double>(0, 1.0)
auto b = 3.14if; // complex<float>(0, 3.14f)
auto c = -99il;  // complex<long double>(0, -99.L)
```

# Input/output: `quoted`

A common C++ gotcha is trying to read a string and getting incomplete input, because `operator>>` is a formatted input function, which means it stops reading on whitespace.

```
string orig = "Hello, world!";
stringstream buf;
buf << orig;
string s;
buf >> s;
assert( s == orig );    //  (â•¯Â°â–¡Â°ï¼‰â•¯ï¸µ â”»â”â”»)
```

# Input/output: `quoted`

C++14 adds a new IO manipulator for writing out, and reading back in, delimited strings, using the `quoted` manipulator.

```
string orig = "Hello, world!";
stringstream buf;
buf << std::quoted(orig);
string s;
buf >> std::quoted(s);
assert( s == orig );    //  \o/ yay!
```

# Input/output: `gets`

C99 deprecated the `gets()` function and C11 removed it entirely, because it is unsafe and error-prone, but because the C++ standard library is still based on the C90 library, we still defined `gets()` by reference.

Defining an unsafe function in C++ that even C doesn't define was embarrassing, so it has been removed from C++14 - yay!

# Threads: `shared_timed_mutex`

Early versions of the C++0x thread library proposals included a `shared_mutex` type (and associated `shared_lock`) which didn't make it into the C++11 standard.

That type has now been added to C++14, but renamed to `shared_timed_mutex` for consistency with the other Mutex types.

It is a read/write mutex, allowing multiple threads to take a "shared" lock which can then be upgraded to an exclusive lock when a single thread wants to own the mutex.

# Technical Specifications

- Filesystem
- Library Fundamentals (optional, any, polymorphic allocators, string_view, shared_ptr for arrays, variable templates for type traits, boyer-moore search algorithms, sample algorithm and more!)
- Array extensions (dynarray)
- Parallelism
- Concurrency

These slides are available at https://gitorious.org/wakelyaccu/accu2014

You can download and view the HTML version of these slides with commands like:

```
git clone https://git.gitorious.org/wakelyaccu/accu2014.git
firefox accu2014/cxx14lib.html
```