C++ Technical Specifications

A panel discussion

Library Fundamentals TS

Marshall Clow

Library Fundamentals V1 (published in May 2015)

- Calling a function with a tuple of arguments.
- _v shortcut for <type_traits>
- Allocator support for std::experimental::function.
- optional, any, string_view
- New searching algorithms: Boyer-Moore and Boyer-Moore-Horspool Searching
- New sampling algorithms

Library Fundamentals V2 (being published now)

- ostream_joiner
- propagate_const
- GCD and LCM
- Source location information
- Uniform Container erasure

Concurrency TS

Gor Nishanov

future.then

1

N4538 Working Draft, Technical Specification for C++ Extensions for Concurrency

```
future<int> tcp_reader(int64_t total) {
    struct State {
        char buf[4 * 1024];
        int64_t total;
        Tcp::Connection conn;
        explicit State(int64_t total) : total(total) {}
    };
    auto state = make_shared<State>(total);
    return Tcp::Connect("127.0.0.1", 1337).then(
        [state](future<Tcp::Connection> conn) {
            state->conn = std::move(conn.get());
            return do_while([state]()->future<bool> {
                if (state->total <= 0) return make_ready_future(false);
                return state->conn.read(state->buf, sizeof(state->buf)).then(
                    [state](future<int> nBytesFut) {
                        auto nBytes = nBytesFut.get()
                        if (nBytes == 0) return make_ready_future(false);
                        state->total -= nBytes;
                        return make_ready_future(true);
                    });
            });
    });
}
```

N4538 Working Draft, Technical Specification for C++ Extensions for Concurrency

```
future<void> do_while(function<future<bool>()> body) {
    return body().then([=](future<bool> notDone) {
        return notDone.get() ? do_while(body) :
make_ready_future(); });
}
```

future.then

```
auto tcp_reader(int total) -> future<int>
{
    char buf[4 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.Read(buf, sizeof
(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

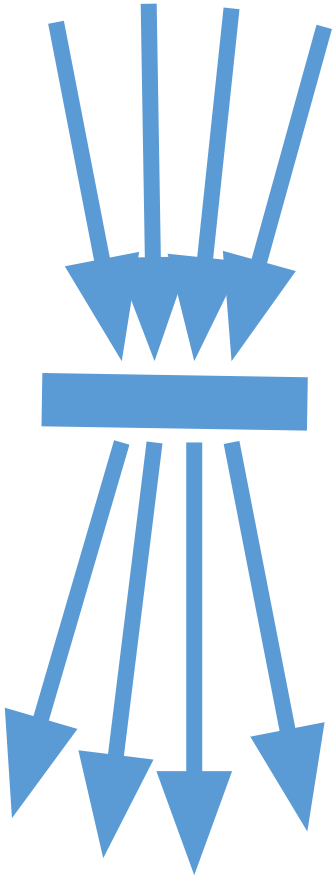
N4538 Working Draft, Technical Specification for C++ Extensions for Concurrency

latch, barrier, flex_barrier

N4538 Working Draft, Technical Specification for C++ Extensions for Concurrency

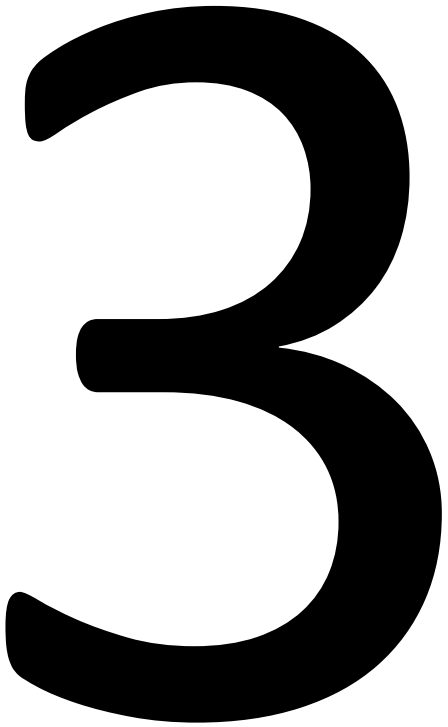
2

count_down_and_wait()
count_down()



arrive_and_wait()
arrive_and_drop()

```
template <class T> struct atomic_shared_ptr;
template <class T> struct atomic_weak_ptr;
```



is_lock_free()
store
load
exchange
compare_exchange_weak
compare_exchange_strong

N4538 Working Draft, Technical Specification for C++ Extensions for Concurrency

https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html

Parallelism TS

Gabriel Dos Reis

ISO/IEC TS 19570:2015

- Execution policy: sequential, parallel, parallel+vector
  - Global policy objects: seq, par, par_vec

```
std::vector<int> v = acquire_data();

// C++14
std::sort(v.begin(), v.end());                  // standard
sequential sort

// C++14 + Parallelism TS
using namespace std::experimental::parallel;
sort(seq, v.begin(), v.end());                  // explicitly
sequential sort
sort(par, v.begin(), v.end());                  // permitting parallel
execution
sort(par_vec, v.begin(), v.end());              // permitting
vectorization as well

// Dynamic selection of execution policy
execution_policy policy = seq;
if (v.size() > threshold)
   policy = par;
sort(with_policy, v.begin(), v.end());
```

New Reduction Algorithms

`std::reduce`

like **`std::accumulate`**

requires associativity & commutativity

`std::exclusive_scan` `std::inclusive_scan`

like **`std::partial_sum`**

requires associativity

Generalized sum:

`op(x1, …, xn) == op(op(y1,…,yk), op(ym,…, yn))` with m = k+1

Generalized nonconmmutative sum:

`op(x1, …, xn) == op(op(x1,…, xk), op(xm, …, xn))`

with m = k+1

More Algorithms

`std::for_each`

returns an iterator instead of a functor avoids discarding information when implementing higher-level algorithms

`std::for_each_n`

implements **`std::generate_n`**, **`std::copy_n`**, etc.

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

Transactional Memory TS

Michael Wong

Why do we need a TM language?

TM requires language support

Hardware here and now

Multiple projects extend C++ with TM constructs

Adoption requires common TM language extensions

Draft specification of transactional language constructs for C++

- 2008: Discussions by Intel, Sun, IBM started in July
- 2009: Version 1.0 released in August
- 2011: Version 1.1 fixes problems in 1.0, exceptions
- 2012: Brought proposal to C++Std SG1; became SG5
- 2013: close to wording for a C++ Technical Specification
- 2015: TM for C++ approved; starting TM 2 for C++ and TM 1 for C

2015: SG5 TM TS Language in a nutshell

1 construct for transactions

1. Compound Statements

2 Keywords for different types of TX

atomic_noexcept | atomic_commit | atomic_cancel {<compound-statement> }

synchronized {<compound-statement> }

2 Function/function pointer keyword

transaction_safe

transaction_safe_dynamic

-must be a keyword because it conveys necessary semantics on type

2 Function/function pointer attribute

[[transaction_unsafe]]

-provides static checking and performance hints, so it can be an attribute

[[optimized_for_synchronized]]

-provides a speculative version for synchronized blocks for the common case when no unsafe functions are called

SNEAK PEAK: 2015: TM TS For C

1 construct for transactions

1. Compound Statements

2 Keywords for different types of TX

_Atomic {<compound-statement> }

_Synchronized {<compound-statement> }

2 Function/function pointer keyword

_Transaction_safe

_Transaction_unsafe

-provides static checking and performance hints, so it can be an attribute

_Optimized_for_synchronized

-provides a speculative version for synchronized blocks for the common case when no unsafe functions are called

Networking TS

Michael Caisse

Doc : N4478

Author : Christopher Kohloff

Boost.Asio

reference implementation

Standalone Asio

`http://think-async.com/`

TS Addresses

- Networking using TCP and UDP, including support for multicast.
- Client and server applications.
- Scalability to handle many concurrent connections.
- Protocol independence between IPv4 and IPv6.
- Name resolution (i.e. DNS).
- Timers.

```
class async_connection  : public std::enable_shared_from_this<async_connection>
{
public:
  async_connection(tcp::socket socket)  : socket_(std::move(socket))
  {}

  void start() { do_read(); }

private:
  void do_read()
  {
    auto self(shared_from_this());
    socket_.async_read_some(buffer(buffer_space_),
        [this, self](std::error_code ec, std::size_t length)
        {
          if (!ec)
          {
            uppercase(buffer_space_.begin(), buffer_space_.begin() + length);
            do_write(length);
          }
        });
  }

  void do_write(std::size_t length)
  {
    auto self(shared_from_this());
    async_write(socket_, buffer(buffer_space_, length),
        [this, self](std::error_code ec, std::size_t /*length*/)
        {
          if (!ec)
          {
            do_read();
          }
        });
  }
  tcp::socket socket_;
  std::vector<char> buffer_space_{1024};
};
```

Asynchronous Models

Callbacks

```
auto self(shared_from_this());
socket_.async_read_some(buffer(buffer_space_),
     [this, self](std::error_code ec, std::size_t length)
     {
       if (!ec){ /* do work ... */ }
     });
```

Futures

```
std::future<std::size_t> fut =
      socket.async_read_some(buffer(buf_space_), use_future);

// ...
std::size_t length = fut.get();
```

Coroutines / resumable functions

```
try {
  std::vector<char> buf_space(1024);
  for (;;)
  {
    std::size_t len = socket.async_read_some(buffer(buf_space_),
                                             yield);
    // do work ...
    async_write(socket, buffer(buf_space_, len), yield);
  }
}
catch (std::system_error& e) { ... }
```

Filesystem TS

Beman Dawes

Concepts TS

Eric Niebler

Please ask questions!