Future of C++

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Spring 2019

Outline

- new stuff in C++17 that we haven't seen yet
- proposals for C++20

https://isocpp.org/std/status
https://en.wikipedia.org/wiki/C%2B%2B20

C++ status in compilers/libraries

- https://en.cppreference.com/w/cpp/compiler_support

clang

- compiler status: https://clang.llvm.org/cxx_status.html
- libc++ status: http://libcxx.llvm.org/ under *Current status*

gcc

- compiler status: https://gcc.gnu.org/projects/cxx-status.html
- libstdc++ status: https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html

- library features depend on both the compiler and the library implementation
- clang uses libstdc++ by default on Linux, but can use libc++ instead
  - `-stdlib=libc++`

Already Seen New Features in C++17

Language

- `if constexpr`
- fold expressions
- `auto` parameters in templates
- class template argument deduction
- structured bindings `auto [ a, b ] = get_tuple();`
- lambdas can capture `*this` (copy of current object)
- guaranteed cases of copy elision

Library

- `std::optional`, `std::variant`, `std::any`
- `std::string_view`
- filesystem support
- uninitialized memory helper functions
- `std::scoped_lock`

New Features in C++17

`if`/`switch` with initializer

- allows to declare a variable and then evaluate an expression
- `if ( auto x = get(); x.valid() ) { ... }`
- the scope of `x` is the whole `if ... else` statement
- in fact, both parts can be expressions and both parts can be declarations

New Features in C++17

Attributes

- in the language since C++11
- in C++14: `[[noreturn]]`, `[[deprecated]]`

`[[fallthrough]]`

- explicit annotation for fallthrough `case`s
- better diagnostics (warnings) with `-Wimplicit-fallthrough` (clang)

`[[nodiscard]]`

- issue a compiler warning if a value is discarded
- not yet used in the standard library, proposal for C++: `std::async`, `empty` methods of containers, etc.

`[[maybe_unused]]`

- suppress warnings on unused entitites (parameters, variables, . . .)

New in C++17: More Guarantees About Evaluation Order

What is stored in `s` at the end of the following program?

```
std::string s = "but I have heard it works even if you "
                "don't believe in it";
s.replace(0, 4, "")
 .replace(s.find("even"), 4, "only")
 .replace(s.find(" don't"), 6, "");
```

New in C++17: More Guarantees About Evaluation Order

What is stored in `s` at the end of the following program?

```
std::string s = "but I have heard it works even if you "
                "don't believe in it";
s.replace(0, 4, "")
 .replace(s.find("even"), 4, "only")
 .replace(s.find(" don't"), 6, "");
```

- it is not defined before C++17!

New in C++17: More Guarantees About Evaluation Order

C++17 guarantees that `a` is evaluated before `b` in these cases:

- `a.b`, `a->b`, `a->*b`
- `a(b1, b2, b3)` (arguments can still be evaluated in arbitrary order)
- `b = a`, `b @=a` (any compound assignment)
- `a[b]`
- `a << b`, `a >> b`
- **this evaluation order is preserved even for overloaded operators**
  - e.g. stream operators

New Features in C++17

Miscellaneous language features

- fixed behaviour of `auto x{ y }`
  - now `x` and `y` have the same type; in C++14 `x` is `initializer_list`
- nested namespace declaration `namespace foo::bar { ... }`
- `static_assert` without message
- inline variables
  - can be safely defined in header files
- UTF-8 literals `u8"something"`
- lambdas implicitly `constexpr`
- range `for` generalisation: `end` and `begin` can have different types
- `__has_include` preprocessor helper

New Features in C++17: Library

`std::invoke`

- allows uniform invocation of functions, function objects, function pointers, members functions pointers, and member data pointers
- for member pointers the first argument must be object on which the member functions should be invoked
- for member data pointers the only argument should be the instance from which the data should be extracted
- this is what happens inside `std::function`, `std::bind`, `std::thread` constructor, now it is available in the library

New Features in C++17: Library

`std::invoke`

- allows uniform invocation of functions, function objects, function pointers, members functions pointers, and member data pointers
- for member pointers the first argument must be object on which the member functions should be invoked
- for member data pointers the only argument should be the instance from which the data should be extracted
- this is what happens inside `std::function`, `std::bind`, `std::thread` constructor, now it is available in the library

`std::apply`

- invokes given callable object with arguments from a tuple
- essentially unwraps the tuple into `std::invoke`
- works on anything that has `std::get` and `std::tuple_size`

New Features in C++17: Library

`std::invoke`

- allows uniform invocation of functions, function objects, function pointers, members functions pointers, and member data pointers
- for member pointers the first argument must be object on which the member functions should be invoked
- for member data pointers the only argument should be the instance from which the data should be extracted
- this is what happens inside `std::function`, `std::bind`, `std::thread` constructor, now it is available in the library

`std::apply`

- invokes given callable object with arguments from a tuple
- essentially unwraps the tuple into `std::invoke`
- works on anything that has `std::get` and `std::tuple_size`

`std::make_from_tuple`

- like `std::apply`, but invokes a constructor

New Features in C++17: Library

`map` and `set` extensions

- support for moving nodes between instances of the container (avoiding copy/move constructors of contained values)
- merging of containers
- `map` only: `insert_or_assign`, `try_emplace` (takes a key and arguments to construct value from)

`std::shared_mutex`

- reader-writer mutex
- multiple readers can share the mutex
  - `lock_shared` method
  - better: use `std::shared_lock`
- writer access is exclusive
  - `lock` method
  - `std::unique_lock` or `std::lock_guard` or `std::scoped_lock`

New Features in C++17: Library

parallel algorithms

- overloads of standard algorithms
- first parameter: execution policy
  - `std::execution::par` – run in parallel
- not yet supported by either clang or gcc
- currently only supported by Intel C++

Technical Specifications

large topics considered for standardisation are first processed in form of technical specifications (TS)

- they are refined, implemented in compiler/library as experimental features
  - e.g. `<experimental/*>` headers for library extensions
- later might be merged into a standard

Technical Specifications

large topics considered for standardisation are first processed in form of technical specifications (TS)

- they are refined, implemented in compiler/library as experimental features
  - e.g. `<experimental/*>` headers for library extensions
- later might be merged into a standard
- some of the current TS are:
  - concepts – templates with requirements on the substituted arguments
  - ranges – concept-based range versions of STL iterators and algorithms
  - networking – small set of network-related libraries, ASIO inspired
  - modules – support for modules as an alternative to headers
  - coroutines – support for generators working similar to C# `yield`
  - concurrency – extended futures and promises, barriers, ...
  - and more: `http://en.cppreference.com/w/cpp/experimental`

C++20

Features currently voted into C++20

- designated initializers
- lambdas with templates
- initialization in range-based `for`
- comparison operator <=>
- concepts
- contracts
- ranges
- coroutines
- modules
- ...

Designated Initializers

- already in C since C99
- restricted (need to keep ordering, not for arrays)

```
struct A {
    int x = 0;
    int y = 0;
    double z;
};

A a { .x = 5, .z = 3.14 };
```

Lambdas with Templates

- this is in fact templated:

```
[](auto x, auto y) { return x + y; }
```

- C++20: allow for explicit template specification

```
[]<typename T>(T x, T y) { return x + y; }

[]<typename It>(It iter, typename It::difference_type diff) {
    /* ... */
}
```

Initialization in Range-Based `for`

- allow initialization part before range

```
for (auto thing = get_thing();
     const auto& item : thing.items()) {
    // do something
}
```

- `for (const auto& item : get_thing().items())` might not be well-defined, why?

Initialization in Range-Based `for`

- allow initialization part before range

```
for (auto thing = get_thing();
     const auto& item : thing.items()) {
    // do something
}
```

- `for (const auto& item : get_thing().items())` might not be well-defined, why?
  - what if `get_thing` returns a temporary object?

Three-way Comparison Operator

`operator <=>` (aka *spaceship*)

- implementing it automatically generates <, >, <=, >=, ==, !=
- can be defaulted – compiler automatically implements it
- different return type for different semantics:
  - `strong_ordering`
  - `weak_ordering`
  - `partial_ordering`
  - `strong_equality`
  - `weak_equality`

Concepts

Motivation

```
struct A { /* ... */ };
std::map< A, std::string > map;
```

what happens if `A` does not define `operator<`?

Concepts

Motivation

```
struct A { /* ... */ };
std::map< A, std::string > map;
```

what happens if `A` does not define `operator<`?

- a nasty type error somewhere deep inside `std::map` implementation

Concepts

Motivation

```
struct A { /* ... */ };
std::map< A, std::string > map;
```

what happens if `A` does not define `operator<`?

- a nasty type error somewhere deep inside `std::map` implementation

what if we could say that the key of `std::map` has to support `operator<`?

- this is what concepts do
- allows better error messages

Concepts

Motivation

```
struct A { /* ... */ };
std::map< A, std::string > map;
```

what happens if `A` does not define `operator<`?

- a nasty type error somewhere deep inside `std::map` implementation

what if we could say that the key of `std::map` has to support `operator<`?

- this is what concepts do
- allows better error messages

the standard already mentions concepts, but as an abstract description of requirements, not something checkable or usable by the compiler

- e.g. ForwardIterator concept

Concepts

- concept declaration

```
template <typename T>
concept bool LessComparable = requires(T a, T b) {
    { a < b } -> bool;
};
```

- concept constraints

```
template <typename K, typename V>
    requires LessComparable<K>
struct map { ... };

template <typename Container>
void sort(Container container)
    requires LessComparable<typename Container::value_type>
{ ... }
```

- currently supported by `gcc -fconcepts`

Concepts

- shortcuts

```
template <LessThanComparable K, typename V>
struct map { ... };

// this is a templated function!
void f(auto param) { ... };

// this is a templated function with a constraint
void sort(Sortable auto& container) { ... };

// can also be written as
void sort(Sortable& container) { ... };
```

Contracts

- allow to specify function pre- and post-conditions

```
int f(int i)
    [[expects: i > 0]]
    [[ensures x: x < 1]];
```

- assert attribute

```
int f(int i) {
    int x = i * i;
    [[assert: x >= 0]];
    ...
}
```

- code analysis tools / optimizers may use them
- violation may be reported at run time

Ranges

- a complete rewrite of the algorithms part of the standard library
- we have seen this earlier
- implementation: https://github.com/CaseyCarter/cmcstl2

```
namespace view = std::experimental::ranges::view;

std::vector ints { 1, 2, 3, 4, 5 };
auto even = [](int i) { return i % 2 == 0; };
auto square = [](int i) { return i * i; };
for (int i : ints | view::filter(even)
                  | view::transform(square)) {
    std::cout << i << ' ';
}
```

Coroutines

- functions that can suspend their execution and be resumed later
- coroutines in Python: `yield`

`co_await`

- suspend execution until a promise is fulfilled

`co_yield`

- suspend execution and return a value to caller

`co_return`

- finish execution and return a value to caller

Modules

includes are not a good way to declare functions from libraries

- they are just textual substitution
- risk of macro collisions
- processed every time they are used, slow compilation

Modules

includes are not a good way to declare functions from libraries

- they are just textual substitution
- risk of macro collisions
- processed every time they are used, slow compilation

there is need for something better, modules

- better isolated, faster
- should allow gradual and backward-compatible move to modules
- but there will be no standardised format of compiled modules
  - modules represented in a compiler-specific way
  - in a way more advanced precompiled headers

Modules

```
import std.io; // make names from std.io available
export module M; // declare module M
export import std.random; // import and export names from
                          // std.random
export struct Point { // define and export Point
    int x;
    int y;
};

export template< typename T >
T foo( const T &x ) { return x; }

#define MACROS_ARE_NOT_EXPORTED "Yay!"
```

- initial implementation in Visual Studio and clang (`-fmodules-ts`)