C++14

Bernhard Manfred Gruber

Resources

- C++ Weekly - Ep 178 - The Important Parts of C++14 In 9 Minutes
  - https://www.youtube.com/watch?v=mXxNvaEdNHI
- ISO C++ FAQ
  - https://isocpp.org/wiki/faq/cpp14-language
- Compiler support
  - https://en.cppreference.com/w/cpp/compiler_support/14
  - gcc 5
  - clang 3.4
  - MSVC 19.10 (Visual Studio 2017)

std::make_unique<T>(…)

- Factory function for `std::unique_ptr<T>`
- Like `std::make_shared<T>(…)` for `std::shared_ptr<T>`
- Array version: `std::make_unique<T[]>(size)`
- Forwards arguments to T's constructor
  - If T does not have a constructor, T is value-initialized (~ zero-ed)
    - Allocating a large array will initialize it!
- C++20: `std::make_unique_for_overwrite<T>(…)`
  - T is only default constructed
    - If T is trivially default constructible, T will be left uninitialized

std::make_unique<T>(…)

- Avoids spelling T twice:

```
auto p = std::unique_ptr<T>{new T{arg1, arg2}};
auto p = std::make_unique<T>(arg1, arg2);
```

- Can prevent a bug:

```
f(std::unique_ptr<T>{new T}, std::unique_ptr<T>{new T});
f(std::make_unique<T>(), std::make_unique<T>());
```

  - If the two news are evaluated and the 2nd throws, before the result of the first new is wrapped in a `unique_ptr`, we leek memory.
  - Fixed in C++17 by constraining the order of evaluation

Generalized capture expressions in lambdas

- In C++11, lambdas can capture local variables from enclosing scopes by value or by reference

```
int val, ref;
auto l = [val, &ref] { use(val, ref); };
```

- In C++14, we can additionally define new variables which live inside the lambda closure object

```
auto l = [val, &ref, newVal = 42, &newRef = ref] {
    use(newVal, newRef);
};
```

Generalized capture expressions in lambdas

```
std::vector<int> v(10);
int i = 0;
std::generate(v.begin(), v.end(), [&i] { // C++11
    return i++;
});
std::generate(v.begin(), v.end(), [j = 0]() mutable {// C++14
    return j++;
});
auto p = std::make_unique<T>();
std::thread t{[p = std::move(p)] {
    p->use();
}};
```

cppinsights.io

Polymorphic lambdas

```
[](auto t1, const auto& t2, auto&& t3) { … }
```

- Accepts parameters of any type
  - operator() becomes a template
  - cppinsights.io
- Can save some typing/redundancy
- Invaluable in meta-programming
- C++20: adds template syntax: `[]<typename T>(T t) { … }`

Polymorphic lambdas

```
std::vector<ReallyLooooooooooooongTypeName> v;
std::sort(v.begin(), v.end(),
    [](const auto& a, const auto& b) {
        return a.member < b.member;
    });
```

Polymorphic lambdas

```
using namespace boost::mp11;
std::tuple<int, float, double> t{ 1, 2.0f, 3.0 };
tuple_for_each(t, [](auto& e) {
    e *= 5;
});
// t contains {1, 4.0f, 9.0}

template <int i> void f();
mp_for_each<mp_iota_c<10>>([](auto ic) {
    constexpr int i = decltype(ic)::value;
    f<i>();
});
```

Deduced return types for functions

- In C++11, the return type of lambda functions can be deduced
- In C++14, this is allowed on all functions

```
auto f() { return 42; } // deduces -> int
```

- If a return type is omitted, the function body must be visible before the function can be called
  - If usage in multiple TUs is intended, function must be in header file
- All return statements must return the same type
- Useful in meta-programming

Deduced return types for functions

```
template <typename F>
auto timed(F f) {
    const auto start = std::chrono::system_clock::now();
    auto ret = f();
    const auto end = std::chrono::system_clock::now();
    return std::make_tuple(std::move(ret), end - start);
}
```

Relaxed restrictions on constexpr functions

- Multiple statements allowed
- Local variable definitions allowed
  - No static or thread_local
  - Must be initialized
  - Of literal type
    - Trivial destructor, aggregate or constexpr constructor, …
- Objects inside constexpr evaluation may be mutated
- constexpr member functions are no longer const
- if, switch, for, while, do allowed

Variable templates

- Allows to template (and specialize) a variable
- Useful to define constants and type traits
- Compile time cache (templates are memoized)

```
template <typename T>
constexpr bool fitsInRegister = sizeof(T) <= 8;
template <>
constexpr bool fitsInRegister<long double> = true;

static_assert(fitsInRegister<T>);
```

Variable templates

```
constexpr std::size_t fib1(std::size_t i) {
    if (i < 2) return 1;
    return fib1(i - 1) + fib1(i - 2);
}
template <int i>
constexpr std::size_t fib2 = fib2<i - 1> + fib2<i - 2>;

template <>
constexpr std::size_t fib2<1> = 1;

template <>
constexpr std::size_t fib2<0> = 1;

constexpr auto r1 = fib1(35); // 36 exceeds 33554432 expansions
constexpr auto r2 = fib2<901>; // 902 exceeds 900 rec. tmpl. Inst.
```

Aggregates with default member initializer

- An aggregate is an array or class type, with:
  - only public data members (no static ones)
  - no constructors (the matter is bit more complicated)
  - only public base classes, if any
  - no virtual member functions
  - no default member initializers (until C++14)
- Aggregates allow easy/flexible initialization
- C++20: adds designated initializer

Aggregates with default member initializer

```
struct RGB { unsigned char r, g, b; };
struct Style {
    RGB textColor = {};
    int fontSize = 12;
    RGB bgColor = {255, 255, 255};
};
void setStyle(Style) { … }

setStyle({}); // black, 12, white
setStyle({ {255} }); // red, 12, white
setStyle({ {0, 0, 255}, 16 }); // blue, 16, white
setStyle({ {}, 16, {0, 255, 0 }}); // black, 16, green
setStyle({.fontSize = 16}); // C++20, black, 16, white
```

[[deprecated]]

- Marks an entity as deprecated
  - Class/struct/union/enum, typedef/using, variable, data member, function, namespace, enumerator, template specialization
- No semantic consequences
- Compilers typically warn when such an entity is used
- Allows optional message: [[deprecated(“string literal”)]]
- Useful to warn library users about a future removal of a feature
  - Compilation error if users turn warnings into errors

[[deprecated]]

```
struct [[deprecated("Please use NewThing instead")]] OldThing {};
OldThing ot;

enum class ReportFormat {
    Json,
    Yaml,
    Xml [[deprecated]]
};
void printReport(ReportFormat rf) {}

printReport(ReportFormat::Json);
printReport(ReportFormat::Xml);
```

```
<source>:123:18: warning: 'OldThing' is deprecated: Please use NewThing instead [-
Wdeprecated-declarations]
  123 |         OldThing ot;
      |                  ^~
<source>:112:54: note: declared here
  112 | struct [[deprecated("Please use NewThing instead")]] OldThing {};
      |                                                      ^~~~~~~~
<source>:125:35: warning: 'ReportFormat::Xml' is deprecated [-Wdeprecated-declarations]
  125 |         printReport(ReportFormat::Xml);
      |                                   ^~~
<source>:117:9: note: declared here
  117 |         Xml [[deprecated]]
      |         ^~~
```

decltype(auto)

- Deduces a type, like `auto`, but with `decltype` semantic
  - `auto` uses the semantic of template type deduction
- Useful when you need to forward a value/reference
  - As a function return type
  - As the type of a variable receiving a return value from such a function

decltype(auto)

```
T value();
T& reference();

template <typename F>
auto wrap_auto(F f) { return f(); }

template <typename F>
decltype(auto) wrap_decltype_auto(F f) { return f(); }

wrap_auto(value);              // returns T
wrap_auto(reference);          // returns T
wrap_decltype_auto(value);     // returns T
wrap_decltype_auto(reference); // returns T&
         auto  r = wrap_decltype_auto(reference); // r is T
decltype(auto) r = wrap_decltype_auto(reference); // r is T&
```

String, chrono and complex literal operators

```
using namespace std::literals;

auto s1 = "asdf";  // const char*
auto s2 = "asdf"s; // std::string

long long durationNs = ((((2ll * 60 + 7) * 60 + 12) * 1000
    + 400) * 1000 + 10) * 1000 + 740;
auto duration = 2h + 7min + 12s + 400ms + 10us + 740ns;
// decltype(duration) = chrono::duration<..., nano>

auto c1 = std::complex<double>{1.1, 2.2};
auto c2 = 1.1 + 2.2i;
```

std::exchange

- Replaces an object's value with a new one, and returns the old one

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

- Can be useful when implementing move constructor/assignment
- Sometimes a better alternative than std::swap

```
std::vector<Event> events;
void processPendingEvents() {
    for (auto e : std::exchange(events, {}))
        process(e);
}
```

std::quoted

- An IO manipulator to wrap/unwrap and escape/unescaped a string
  - By default, wraps in quotes and escapes with backslash

```
const auto s = std::string{R"(abc\def"ghi")"};
std::stringstream ss;
ss << std::quoted(s);
std::string s2;
ss >> s2; // s2 == R"("abc\\def\"ghi\"")“
ss.seekg(0);
ss >> std::quoted(s2); // s2 == s
```

std::shared_timed_mutex, std::shared_lock

- Contrary to std::[timed_]mutex, std::shared_timed_mutex allows multiple threads to access a resource in addition to exclusive access
  - A reader/writer lock is a popular scenario
- Also offers (as std::timed_mutex):
  - try_lock_for(std::chrono::duration)
  - try_lock_until(std::chrono::time_point)
- Scope guards for locking
  - std::unique_lock/std::lock_guard for exclusive access
  - std::shared_lock for shared access
- std::shared_mutex in C++17

std::shared_timed_mutex, std::shared_lock

```
std::string s;
std::shared_timed_mutex m;
```

Reading threads

```
{
  std::shared_lock<decltype(m)>
    lock{m};
  auto copy = s;
}
```

Writing threads

```
{
  std::unique_lock<decltype(m)>
    lock{m};
  s = "new content";
}
```

std::integer_sequence

- Represents a compile-time sequence of integers

```
auto is1 = std::integer_sequence<std::size_t, 0, 1, 2, 3>{};
auto is2 = std::index_sequence<0, 1, 2, 3>{}
auto is3 = std::make_integer_sequence<std::size_t, 4>{}
auto is4 = std::make_index_sequence<4>{};
```

- Useful for template metaprogramming

std::integer_sequence

```
template <typename Tuple, typename Func, std::size_t... Is>
void tuple_for_each_impl(Tuple t, Func f, std::index_sequence<Is...>){
    using A = int[];
    A{ 0, (f(std::get<Is>(t)), 0)... };
    (f(std::get<Is>(t)),...); // C++17
}

template <typename Tuple, typename Func>
void tuple_for_each(Tuple t, Func f){
    tuple_for_each_impl(t, f,
        std::make_index_sequence<std::tuple_size<Tuple>::value>{});
}

void integer_sequence(){
    std::tuple<int, float, double> t{1, 2.0f, 3.0};
    tuple_for_each(t, [](auto& e) { e++; });
}
```

std::equal, std::is_permutation, std::mismatch

- All these algorithms are called with 2 ranges, passed as:

```
std::equal(a.begin(), a.end(), b.begin());
```

- Problematic if ranges have different length
- C++14: adds overloads with the end of the 2nd range as 4th argument:

```
std::equal(a.begin(), a.end(), b.begin(), b.end());
```

  - Ranges of different lengths are not equal

```
std::mismatch(a.begin(), a.end(), b.begin(), b.end());
```

  - Additional elements of the longer range are no mismatches

```
std::is_permutation(a.begin(), a.end(), b.begin(), b.end());
```

  - Ranges of different lengths are not permutations of each other

Further new language features

- Binary literals

```
int bin = 0b101010;
```

  - Suffixes like u, ul, ull can be added
- Digit separators

```
auto num = 1'562'453'752'541ul;
```

- New/delete omission/fusion (compiler optimization)
  - aka. heap elision
- Sized deallocation

```
void operator delete(void*, std::size_t) noexcept { … }
```

  - Useful to implement global memory allocators (e.g. TCMalloc)

Further new library features

- Type trait aliases
  - `std::remove_const<T>::type` becomes `std::remove_const_t<T>`
  - `C++17: std::is_const<T>::value` becomes `std::is_const_v<T>`
- constexpr for complex, chrono, array, initializer_list, utility, tuple
- Operator function objects without value types (std::plus<>{})
- std::result_of is SFINAE friendly (removed in C++20)
- std::integral_constant<T, val>::operator()
- Value initialized iterators are valid
- Heterogeneous associative container lookup
- std::get<T> for std::pair