C++14 is coming

Kurbakov Dmytro

April 10, 2019

Overview

Removed functions

std::gets

Reads stdin into given character string until a newline character is found or end-of-file occurs.

C++11

```
char* std::gets( char* str );
```

This unsafe I/O function from the C library is no longer available.

Source: ISO/IEC CD 14882, C++ 2014, National Body Comments

std::rand

Returns a pseudo-random integral value between 0 and RAND_MAX (0 and RAND_MAX included).

C++11

```
// use current time as seed for random generator
std::srand(std::time(nullptr));
std::cout << std::rand() << '\n';
```

This low-quality random number facility from the C library is discouraged in favour of the random library.

Source: Discouraging rand() in C++14, v2

Language extentions

Generalized return type deduction

C++11 **auto** was introduced with limited area of usage.
C++11 missed **auto** for return types...

> *...due to time constraints, as the drafting didn't address various questions and concerns that the Core WG had.*

C++14 **auto** can be used as return type... but not for virtual calls

> *It would be possible to allow return type deduction for virtual functions, but that would complicate both override checking and vtable layout, so it seems preferable to prohibit this.*

Source: Return type deduction for normal functions

Generalized return type deduction

C++11

```
auto f() -> int { return foo() * 42; }
```

C++14

```
auto f() { return foo() * 42; }
auto g() {
    if( expr ) { return foo() * 42; }
    // multiple returns (types must be the same)
    return bar.baz(84);
}
```

decltype(auto)

decltype(auto) is primarily useful for deducing the return type of forwarding functions and similar wrappers and not intended to be a widely used feature beyond that.

Source: Return type deduction for normal functions

decltype(auto)

Given

```
string  lookup1();
string& lookup2();
```

C++11

```
string  wrapper_1() {return lookup1();}
string& wrapper_2() {return lookup2();}
```

C++14

```
decltype(auto) wrapper_1() {return lookup1();}
decltype(auto) wrapper_2() {return lookup2();}
```

decltype(auto)

Important: **decltype(auto)** is sensitive to how you write the return statement

Quiz: what return these two functions?

```
decltype(auto) foo() {
    auto str = lookup1(); return str;
}
decltype(auto) boo() {
    auto str = lookup1(); return(str);
}
```

Answer

decltype(auto) vs. auto

When should we use **auto** and when **decltype(auto)**?

Rules of thumb

- Use **auto** if a reference type would never be correct.
- Use **decltype(auto)** only if a reference type could be correct.

Source: C++ Type deduction and why you care. S. Mayers

Generalized lambda captures

- C++11 no support fo capturing by move.
- C++14 generalized lambda capture (capture by move, define arbitrary new local variables in the lambda object).

Source: Wording Changes for Generalized Lambda-capture

Generalized lambda captures

C++14: capture by move

```
// a unique_ptr is move-only
auto u = make_unique<Type>(arg1, arg2);
// move the unique_ptr into the lambda
go.run( [ u=move(u) ] { foo( u ); } );
```

C++14: define new variables

```
int x = 4;
int z = [&r = x, y = x+1] {
r += 2; // set x to 6; "R is for Renamed Ref"
return y+2;  // return 7 to initialize z
}(); // invoke lambda
```

Generic lambdas

- Lambda function parameters can now be auto to let the compiler deduce the type.
- Fix UB. See the example from the Appendix A.

Source: Proposal for Generic (Polymorphic) Lambda Expressions

Generic lambdas

C++14

```
for_each(begin(v), end(v), [](const auto& x) {
        cout << x;
} );
sort( begin(w), end(w),
        [](const auto& a, const auto& b) {
        return *a<*b;
} );
auto size = [](const auto& m) {
        return m.size();
};
```

Default arguments for lambdas

Lambda expressions can now have default arguments

```
auto f = [](int a = 10){};
f();
```

Source: Default arguments for lambdas

Variable templates

Idea: Simplify definitions and uses of parameterized constants, allow the definition and uses of constexpr variable templates

Source: Variable Templates (Revision 1)

Variable templates

C++14

```
// variable template
template<class T>
constexpr T pi = T(3.14159265358979323851L);
// function template
template<class T>
T circular_area(T r)
{return pi<T> * r * r;}
```

Extended constexpr

- Allow declarations within constexpr functions, other than
  - static or thread_local variables
  - uninitialized variables
- Allow if and switch statements (but not goto)
- Allow all looping statements: for (including range-based for), while, and do-while
- Allow mutation of objects whose lifetime began within the constant expression evaluation.
- Remove the rule that a constexpr non-static member function is implicitly const.

Source: Relaxing constraints on constexpr functions constexpr member functions and implicit const

Extended constexpr

C++14

```
constexpr
int my_strcmp(const char* str1,
              const char* str2 ) {
  int i = 0;
  for(;str1[i]
       && str2[i]
       && str1[i] == str2[i]; ++i)
      { }
  if( str1[i] == str2[i] ) return 0;
  if( str1[i] < str2[i] ) return -1;
  return 1;
}
```

The [[deprecated]] attribute

- The attribute-token deprecated can be used to mark names and entities whose use is still allowed, but is discouraged for some reason.
- The attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, an enumeration, or a template specialization.

Source: [[deprecated]] attribute

The [[deprecated]] attribute

C++14

```
struct [[deprecated]] S;
class Foo() {
[[deprecated]] void f();
};
using PS [[deprecated]] = S*;
union U { [[deprecated]] int n; };
namespace [[deprecated]] NS { int x; }
```

Tuple access by type

Allowing tuples to be addressed by type as well as by numerical index.

```
tuple<char, char, int> t('a', 'b', 1);
int i = get<int>(t); // i == 1
int j = get<2>(t); // j == 1
char s = get<char>(t); // Compile-time error
```

Souce: Wording for Addressing Tuples by Type: Revision 2

Constexpr member functions without const

A constexpr member function is no longer implicitly const. Only for data members does constexpr imply const now.

C++11

```
// const function
constexpr size_t size() noexcept;
```

C++14

```
// non-const function
constexpr size_t size() noexcept;
// const function
constexpr size_t size() const noexcept;
```

Source: Fixing constexpr member functions without const

Library extentions

Shared locking

The class shared_lock is a general-purpose shared mutex ownership wrapper allowing deferred locking, timed locking and transfer of lock ownership. Locking a shared_lock locks the associated shared mutex in shared mode.
As the result:

- Seven constructors to unique_lock<Mutex> were added
- A new header shared_mutex was added
  - class shared_mutex;
  - class upgrade_mutex;
  - template <class Mutex> class shared_lock;
  - template <class Mutex> class upgrade_lock;

Source: Shared locking in C++

make_unique

No comments =)

C++14

```
auto p1 = make_shared<widget>();
auto p2 = make_unique<widget>();
```

Source: JTC1/SC22/WG21 N3656

Constant, reverse and constant reverse iterators

More iterators in C++14

- Constant iterators: std::cbegin, std::cend
- Reverse iterators: std::rbegin, std::rend
- Constant reverse iterators: std::rcbegin, std::rcend

Overload for std::equal, std::missmatch, std::is_permutation

Algorithms that operate on two ranges get new overloads that take begin and end iterators for both ranges rather than requiring the second range to be sufficiently long.

Source: Making non-modifying sequence operations more robust: Revision 2

std::exchange

Replaces the value of obj with new_value and returns the old value of obj.

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

Important: T must meet the requirements of MoveConstructible. Also, it must be possible to move-assign objects of type U to objects of type T

Source: exchange utility function, revision 3

Compile-time integer sequences

The class template std::integer_sequence represents a compile-time sequence of integers. Consis of:

- std::index_sequence
- std::integer_sequence
- std::make_index_sequence
- std::make_integer_sequence

Source: Compile-time integer sequences

C++14 and NavKit. Open discussion

Language:

- Generalized return type deduction
- decltype(auto)
- Generalized lambda captures
- Generic lambdas
- Default arguments for lambdas
- Variable templates
- Extended constexpr
- The [[deprecated]] attribute
- Tuple access by type
- Constexpr member functions without const

Library

- Shared locking
- std::make_unique
- const and reverse iterators
- Overload for std::equal, std::missmatch, std::is_permutation
- std::exchange
- Compile-time integer sequences

Whant to know more?

Some useful links for C++14

- Changes between C++11 and C++14
- C++14 Language Extensions
- C++14 Library Extensions
- Effective Modern C++ by Scott Meyers, Chapter 1. Deducing Types