C++14

Contents

[1. C++14 Language Extensions 2](#_Toc392944440)

[a. Binary literals 2](#_Toc392944441)

[b. Generalized return type deduction 2](#_Toc392944442)

[c. decltype(auto) 3](#_Toc392944443)

[d. Generalized lambda captures 4](#_Toc392944444)

[e. Generic lambdas 5](#_Toc392944445)

[f. Variable templates 6](#_Toc392944446)

[g. Extended constexpr 6](#_Toc392944447)

[h. The [ [deprecated] ] attribute 7](#_Toc392944448)

[i. Digit separators 7](#_Toc392944449)

[2. C++14 Library Extensions 7](#_Toc392944450)

[a. Shared locking 7](#_Toc392944451)

[b. User-defined literals for std:: types 7](#_Toc392944452)

[c. make\_unique 8](#_Toc392944453)

[d. Type transformation \_t aliases 8](#_Toc392944454)

# C++14 Language Extensions

## Binary literals

* C++ now supports binary literals:

// The answer to life, the universe, etc. in...

auto a1 = 42; // ... decimal

auto a2 = 0x2A;  // ... hexadecimal

auto a3 = 0b101010;  // ... binary

* This works well in combination with the new ' digit separators, for example to separate nybbles or bytes:

auto a = 0b100'0001; // ASCII 'A'

## Generalized return type deduction

* C++11 permitted automatically deducing the return type of a lambda function whose body consisted of only a **single** return **statement**:

// C++11

[=]() -> some\_type  { return foo() \* 42; } // OK

[=]  { return foo() \* 42; } // OK, deduces "-> some\_type"

* This has been expanded in two ways.
  1. First, it now works even with more complex function bodies containing **more than one** return statement, as long as all return statements return the same type:

// C++14

[=] { // ok, deduces "-> some\_type"

while (something())

{

if (expr)

{

return foo() \* 42;  // with arbitrary control flow

}

}

return bar.baz(84);  // & multiple returns

} // (types must be the same)

* 1. Second, it now works with all functions, not just **lambdas**:

**// C++11, explicitly named return type**

some\_type f() { return foo() \* 42; } // OK

auto f() -> some\_type { return foo() \* 42; }// OK

**// C++14**

auto f() { return foo() \* 42; }// OK, deduces "-> some\_type"

auto g() // OK, deduces "-> some\_type"

{

while (something())

{

if (expr)

{

return foo() \* 42; // with arbitrary control flow

}

}

return bar.baz(84); // & multiple returns

}

## decltype(auto)

* Given these functions:

string lookup1();

string& lookup2();

* In C++11 we could write the following wrapper functions which remember to preserve the reference-ness of the return type:

string look\_up\_a\_string\_1() { return lookup1(); }

string& look\_up\_a\_string\_2() { return lookup2(); }

* In C++14, we can automate that:

decltype(auto) look\_up\_a\_string\_1() { return lookup1(); }

decltype(auto) look\_up\_a\_string\_2() { return lookup2(); }

* Note: decltype(auto) is primarily useful for deducing the return type of forwarding functions and similar wrappers, as shown above, where you want the type to exactly “track” some expression you’re invoking.
* However, decltype(auto) is not intended to be a widely used feature beyond that. In particular, although it can be used to declare local variables, doing that is probably just an antipattern since a local variable’s reference-ness should not depend on the initialization expression. Also, it is sensitive to how you write the return statement. These two functions have different return types:

decltype(auto) look\_up\_a\_string\_1() { auto str = lookup1(); return str; }

decltype(auto) look\_up\_a\_string\_2() { auto str = lookup1(); return(str); }

## Generalized lambda captures

* In C++11, lambdas could not (easily) capture by move.
* In C++14, we have generalized lambda capture that solves not only that problem, but allows you to define arbitrary new local variables in the lambda object.

// a unique\_ptr is move-only

auto u = make\_unique<some\_type>(some, parameters);

// move the unique\_ptr into the lambda

go.run([u = move(u)] { do\_something\_with(u); });

* In the above example, we kept the name of the variable u the same inside the lambda. But we’re not limited to that… we can rename variables:

go.run([u2 = move(u)] { do\_something\_with(u2); }); //capture as "u2"

* And we can add arbitrary new state to the lambda object, because each capture creates a new type-deduced local variable inside the lambda:

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. This generates a lambda type with a templated operator() so that the same lambda object can be invoked with any suitable type and a type-safe function with the right parameter type will be automatically generated.
* In C++11, we had to explicitly state the type of a lambda parameter, which was often fine but sometimes annoying:

// C++11: have to state the parameter type

for\_each(begin(v), end(v), [](const decltype(\*begin(v))& x) { cout << x; });

sort( begin(w),

end(w),

[](const shared\_ptr<some\_type>& a, const shared\_ptr<some\_type>& b)

{

return \*a < \*b;

});

auto size = [](const unordered\_map<wstring, vector<string>>& m)

{

return m.size();

};

* In C++14, we can get type deduction for the same functions we could write in C++11:

// C++14: just deduce the type

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;

});

## Variable templates

* Needs further reading…

## Extended constexpr

* In C++11, to make a function constexpr can mean rewriting it. For example, let’s say we have this constexpr function:

constexpr int my\_charcmp(char c1, char c2)

{

return (c1 == c2) ? 0 : (c1 < c2) ? : -1 : 1;

}

* That’s fine and useful for characters, so why not extend it to strings? That would require iteration over the characters of the string, which **C++11 did not allow** in constexpr functions, so the C++11 version that supports strings would have to be recursive instead (and a little more complicated).
* C++14 now allows more things inside the body of constexpr functions, notably:
  + local variable declarations (not static or thread\_local, and no uninitialized variables)
  + mutating objects whose lifetime began with the constant expression evaluation
  + if, switch, for, while, do-while (not goto)
* So in C++14, the above function generalized to strings can stay idiomatic, and use a normal loop directly:

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;

}

* C++14 also removes the C++11 rule that constexpr member functions are implicitly const.

## The [ [deprecated] ] attribute

* The deprecated attribute allows marking an entity deprecated, which makes it still legal to use but puts users on notice that use is discouraged and may cause a warning message to be printed during compilation.
* 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.

## Digit separators

* The single-quote character ' can now be used anywhere within a numeric literal for aesthetic readability. It does not affect the numeric value.

auto million = 1'000'000;

auto pi = 3.14159'26535'89793;

# C++14 Library Extensions

## Shared locking

* Needs further reading…

## User-defined literals for std:: types

* C++11 added user-defined literals, but didn’t use them in the standard library. Now some very useful and popular ones work:

auto a\_string = "hello there"s; // type std::string

auto a\_minute = 60s; // type std::chrono::duration = 60 seconds

auto a\_day = 24h; // type std::chrono::duration = 24 hours

* Note s means “string” when used on a string literal, and “seconds” when used on an integer literal, without ambiguity.

## make\_unique

* This fixes an omission:

auto p1 = make\_shared<widget>(); // C++11, type is shared\_ptr

auto p2 = make\_unique<widget>(); // new in C++14, type is unique\_ptr

* Unlike make\_shared, using make\_unique doesn’t add any optimization. What it does do is enable us to tell people “don’t write naked new any more to allocate an object on the heap” (except only perhaps in the internals of low-level data structures).

## Type transformation \_t aliases

* Needs further reading…