Remedial C++ 14

Pete Williamson
petewil00@hotmail.com

Topics

C++ 14 language changes

C++ 14 library changes

A small block of code modernized

C++ Landscape

Original C++

C++ 98

C++ 11 (previously called C++ 0X)

C++ 14

C++ 17

C++ 20 (standard is finalized)

C++ 23 (in planning)

Links

C++ 14 wikipedia

Scott Meyers' Effective Modern C++

Dr Dobbs (C++ 14)

Anthony Calandra C++ 14

https://chromium-cpp.appspot.com/ - chromium allowed features

… also check your favorite search engine

C++ 14 language changes

- Return type deduction (auto return type)
- decltype(auto) - wha?
- Relaxed constexpr restrictions - constexpr all the things!
- Templates for variables
- Aggregate member initialization
- Binary literals
- Digit separators
- Generic Lambdas
- Lambda capture expressions
- Deprecated attribute

C++ 14 library changes

- Shared mutexes and locking
- Heterogeneous lookup
- Standard user defined literals
- Tuple addressing via type
- Smaller library features

Trailing return type

Trailing return type (C++ 11)

Correction to my last talk, this really is C++11.

```
std::sort(page_infos->begin(),
          page_infos->end(),
          [](const PageInfo& a, const
             PageInfo& b) -> bool {
              return a.last_access_time >
                     b.last_access_time;
            });
```

Return type deduction (auto)

Return type deduction (auto)

What it does:

You can now use “auto” with function return values.

Return type deduction example

```
auto Correct(int i)
{
  if (i == 1)
    return i; // return type deduced int

  return Correct(i-1)+i;  // ok to call
}
```

Return type deduction usage

Like C++11, use it to save typing, not thinking - you should still know what type is being returned.

Can be used with lambdas to get template like behavior. (More on that later.)

decltype(auto) - wha?

decltype(auto) - wha?

So we know that decltype returns the type of the expression in parenthesis.

We know that auto figures out what the type should be.

So what does it mean when you put them together?

decltype(auto)

What it does:

It's like auto, but it keeps const and ref

Example:

```
const int x = 0;
auto x1 = x;                  // x1 is int
decltype(auto) x2 = x; // x2 is const int
```

decltype(auto)

When to use it:

When you would normally use auto, but keeping const or ref is important (typically in templates)

Relaxed constexpr expressions

Relaxed constexpr expressions

What it does: constexpr all the things!*

(* well, more things, C++ 17 adds even more support)

Relaxed constexpr - what is new(1)

Any declarations except:

- static or thread_local variables.
- Variable declarations without initializers.
- The conditional branching statements if and switch.
- Any looping statement, including range-based for.

Relaxed constexpr - what is new(2)

It allows expressions which change the value of an object if the lifetime of that object began within the constant expression function. This includes calls to any non-const constexpr-declared non-static member functions.

You can’t use goto in a constexpr function.

Relaxed constexpr - example

```
constexpr int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
factorial(5); // == 120
```

Relaxed constexpr - when to use

Anytime you can!

It is usually better to let the work be done at compile time instead of runtime if you can.

const data is thread safe, too.

Templates for variables

Templates for variables

What it does: Now you can template variables

Example:

```
template<class T>
constexpr T pi = T(3.14159265358979323851);
template<class T>
constexpr T e  = T(2.71828182845904523531);
```

Templates for variables - usage

This works well for constants such as e and pi.

You might use it if the header was used in some places where you wanted a double, and others where you wanted a float.

Aggregate member initialiers

Aggregate member initialization

What it does:

- It used to be that you couldn’t initialize classes with aggregate initialization if you had member initializers. Now you can.
- In C++14, some braces can be elided if clear from context

Aggregate member initializers

```
struct S {
    int x;
    struct Foo {
        int i;
        int j;
        int a[3];
    } b;
};
```

Aggregate member initializers

Example:

```
S s1 = { 1, { 2, 3, {4, 5, 6} } };
S s2 = { 1, 2, 3, 4, 5, 6};
  // same, but with brace elision
```

Aggregate member initializers

When to use:

- Same as when you used member lists to initialize classes before.

Binary literals / digit separators

Binary Literals

What it does:

Lets you write binary numbers as binary in source.

Example:

```
0b1111'1111 // == 255 (0xFF)
```

When to use it:

Whenever the number makes more sense in binary than hex

Digit Separators

What it does:

Lets us break big numbers down into groups.

Example:

```
auto one_MILLION_dollars = 1'000'000;
auto ichi_oku_yen = 1'0000'0000;
```

When to use it:

Up to your tastes. Makes big numbers readable.

Generic Lambdas

Generic Lambdas

What it does:

Lets you write lambdas with auto types.

Examples:

```
auto lambda = [](auto x, auto y) {return x + y;};
auto identity = [](auto x) { return x; };
int three = identity(3); // == 3
std::string foo = identity("foo"); // == "foo"
```

Generic Lambdas - when to use

These can be used a lot like templates.

They can be used at smaller scope than templates to avoid name collisions.

They are good for visiting syntax:

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Generic Lambdas

The are actually implemented using templates, but a lot of the type ugliness is hidden from the code.

Lambda Capture Expressions

What it does:

When a lambda does a capture, you can initialize it with "="

Example:

```
auto lambda =
    [value = 1] {return value;};
```

Lambda Capture Expressions

When to use:

This is how to get move only expressions into a lambda.

```
std::unique_ptr<int> ptr(new int(10));
auto lambda = [value = std::move(ptr)] {
    return *value;
 };
```

Deprecated attribute

Deprecated attribute

What it does:

Mark code as deprecated so we can give compile warnings

Deprecated example

```
[[deprecated]] int f();
[[deprecated("g() is thread-unsafe. Use h() instead")]]
void g( int& x );

void test()
{
  int a = f(); // warning: 'f' is deprecated
  g(a); // warning: 'g' is deprecated: g() is thread-unsafe. Use h() instead
}
```

Deprecated - when to use it

When you are moving to a new API and want to get folks to stop using the old one.

C++14 library changes

C++ 14 Library changes

- Shared mutexes and locking
- Heterogeneous lookup
- Standard user defined literals
- Tuple addressing via type
- Smaller library features

Library: Shared mutexes and locks

shared_timed_mutex in C++14

For multiple readers, single writer.

The timed part means we can put a timeout on the attempt to get the mutex.

Library - Heterogeneous lookup

If you have a map full of std::string, and you want to look up a char*, this allows you to let your container compare different types of keys together.

It works with any types the comparison operator supports. std::less and std::greater are modified to work for types that can convert to each other.

Library - Std literals with type

Std library now supports declaring types string, duration, and complex by adding a letter to specify the type:

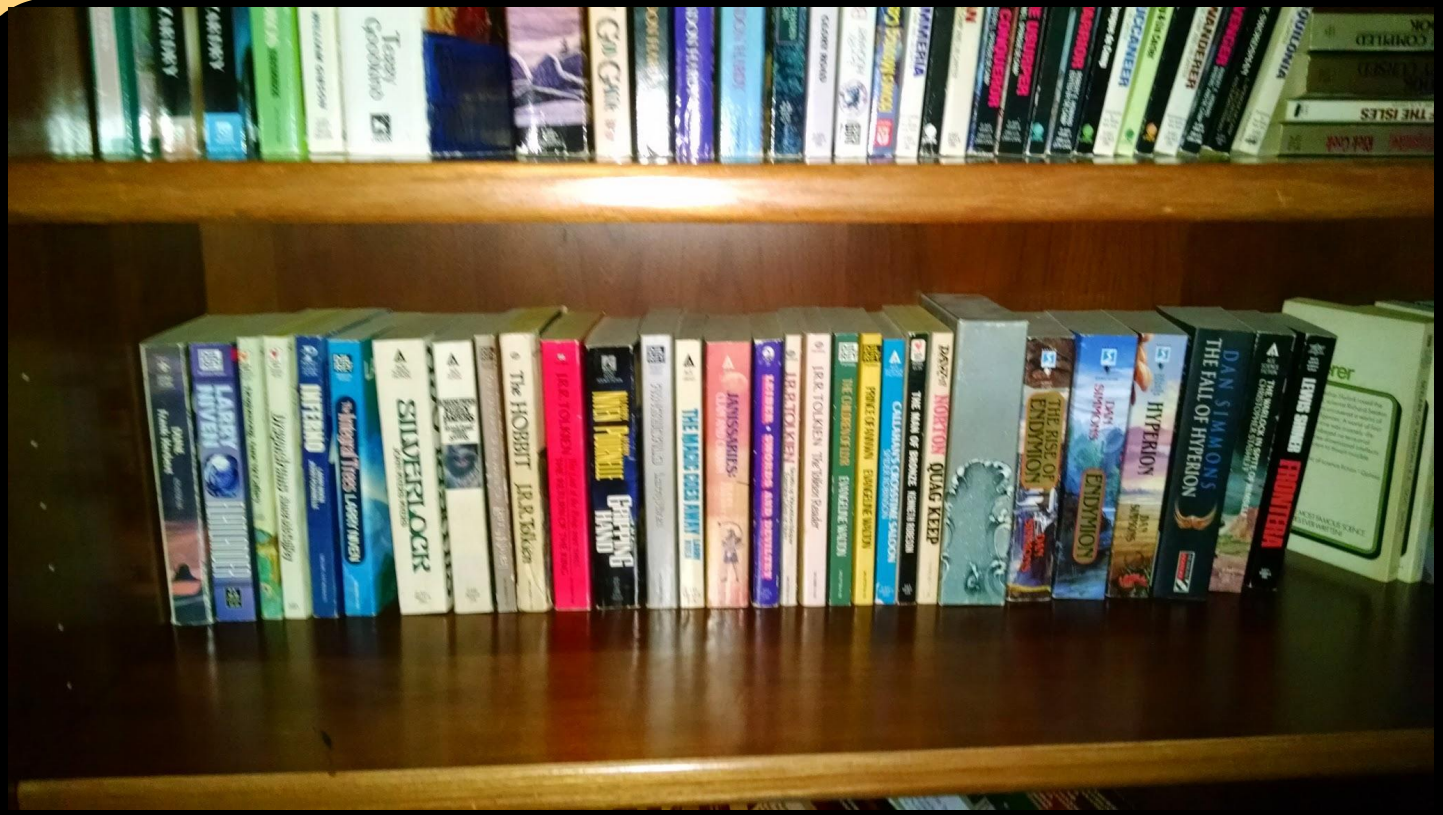
```
auto str = "hello world"s; // auto deduces string
auto dur = 60s; // auto deduces chrono::seconds
auto z   = 1i;  // auto deduces complex<double>
```

Library - tuple addressing via type

You can ask for a type from a tuple instead of having to supply the proper index, as long as it is unambiguous.

```
tuple<string, string, int> t("foo", "bar", 7);
int i = get<int>(t);          // i == 7
int j = get<2>(t);            // Same as before: j == 7
string s = get<string>(t);    // Compile-time error due
                              // to ambiguity
```

Library - smaller features



Library - smaller features

- std::make_unique<> - builds smart pointers
- std::integral_constant - new overload of operator() which returns constant types (think type traits)
- std::integer_sequence added for compile time use of sequences
- const iter support: std::cbegin/std::cend/std::crbegin/std::crend
- New overloads of std::equals and others taking a second pair of ranges - you can skip length check.

Library smaller features

- std::is_final type trait to detect if a class is final.
- std::quoted stream IO manipulator to preserve spaces in a string on output/input by quoting them.

Coding in the modern style

Coding in the modern style

auto

rvalue references and moving

for(each)

constexpr

const iter

lambda

Modern style example (11 and 14)

```
constexpr int Fibbonachi(int n) { … };
const int limit = Fibbonachi(11);  // Calculated at compile time.

std::vector<BigComplicatedVector> varlist;
std::vector<Subcomponent> matches;

for(const auto var : varlist) {
  auto result = std::find_if(std::begin(var), std::end(var),
                    [var]() { var.size() < 10 });
  If (result != var.end())
     matches.emplace_back(std::move(var.contents));
}
```

Without modern style (89)

```
const int limit = 89;  // We calculated it by hand offline

std::vector<BigComplicatedVector> varlist;
std::vector<Subcomponent> matches;

for(BigComplicatedVector.const_iterator it;
      it = std::begin(varlist); it != std::end(varlist)) {
  BigComplicatedVector.const_iterator result =
      std::find_if(std::begin(var), std::end(var), &sizeLessThanTenFunction); // lambda
  If (result != var.end())
     matches.push_back(var.contents);  // Makes a copy instead of a move
}
```

Links to learn more

Links to learn more

Wikipedia - https://en.wikipedia.org/wiki/C%2B%2B14#New_standard_library_features

Anthony Calandra - https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP14.md

CPP Reference - https://en.cppreference.com/w/

Effective Modern C++ by Scott Meyers - https://www.aristeia.com/books.html

Link to my Remedial C++ 11 talk

https://www.youtube.com/watch?v=i1zNN_U6tEQ

Spread your wings and soar

A look ahead

C++ 17 is big, maybe bigger than 11 and 14 combined. It will get it's own slide deck.

C++ 20 is shaping up to be bigger than 17 with Reflection, Coroutines, Concepts, Ranges, and more.

Some features (contracts) have been postponed to C++ 23

CppCast.com (podcast) covers the latest developments.

Thanks for listening!

Any questions?