C++ 11 and C++ 14

New Language And Library Features That Will Make Your Code Better

Kate Gregory
www.gregcons.com/kateblog
@gregcons

C++ Standard

- For a long time, there was no standard at all
  - Multiple compilers, mostly agreed with what Stroustrup wrote
- C++ 98
  - Slightly tweaked in 2003
  - Some people say C++ 98/03
- TR1 – technical report 1
  - The parts of "C++ 0x" everyone could agree on
  - Released in 2005
  - Compilers started to implement parts they liked
- C++ 11
  - What C++0x turned out to be
- C++ 14
  - Settled Feb 15th 2014 at Issaquah meeting
  - Completes C++ 11

- Language
  - Keywords, punctuation, syntax, parsing

- Library
  - std::

Visual C++

- Microsoft C++ 1 was Microsoft C 7.0, in 1992
  - Over 20 years ago!
- VC1 was C++ 2
- … there was no VC3 (version # syncing) …
- VC9 was Visual C++ 2008, Visual Studio 2008
  - VC9 SP1 implemented some TR1 features
- VC10 is Visual C++ 2010, Visual Studio 2010
  - Lots of C++11 features are included
- VC11 is Visual C++ 11, Visual Studio 2012
  - ALL library features
  - Some/most language features
- VC12 is Visual C++ 12, Visual Studio 2013
  - More language features (variadic templates!)
  - Some C++ 14 features

The Big Deals

- auto
  - Productivity, readability
  - Maintenance
  - Needed for lambdas
- Lambdas
  - Make standard algorithms usable
  - Concurrency
  - Functional style
- Range-based for
- Uniform Initialization
  - {} everywhere
- shared_ptr, unique_ptr
  - Don't delete stuff!
  - Also, new stuff less
  - Stack semantics (RAII) is your friend
- Variadic templates

auto

- If you know C# var, you know auto
- Still strongly typed – just not by you
- 3 major strengths:
  - Annoying iterator declarations
  - Unspeakable types
  - Dependent types (again, iterators) in volatile code
- Most of what you don't like about standard containers and standard algorithms goes away with auto

Tiny Functions

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

void print_square(int i)
{
        cout << i*i << endl;
}

int main()
{
        vector<int> v;
        // vector gets filled
        for_each(v.begin(), v.end(), print_square);
}
```

Why Does It Need a Name?

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  vector<int> v;
  // vector gets filled
  for_each(v.begin(), v.end(),
    [](int i) { cout << i*i << endl; } );
}
```

Lambdas

- Three parts
  - [] – "Hi, I'm a lambda" aka capture clause
  - () – parameters (imposed by the caller)
  - {} – body
- Capture clause is non-optional but can be empty
  - [x]
  - [&x]
  - [=]
  - [&]
  - Can also mix and match
- May need to specify return type
  - [](int x) -> int {/* stuff */}

Lambdas and Concurrency

- Parallel Patterns Library (ppl.h)
  - concurrency::parallel_for
  - concurrency::parallel_for_each
- C++ AMP (amp.h)
  - concurrency::parallel_for_each
- Both take a lambda as a parameter
  - Represents the work being spread across cores

Range for

- Most of the for_each you write are for the whole container
  - begin(v), end(v)
  - v.begin(),v.end()
- Neater:

```
for(int elem: v)
    {/*loop body*/}
```

- Note:
  - Language keyword, not library function in std::
  - auto works here too – try const auto& to avoid copies

Initialization

- Many ways to initialize built in types like int
  - int a = 2;
  - int b(2);
- Initializing C-style arrays could be done with {}
  - But who uses C-style arrays now?
- To initialize an object, use a constructor
  - Foo f = 3;
  - Employee newHire(John, today + 1, salary);
  - Employee CEO();
  - Employee someone;
- Lots of different ways means confusion
  - Especially for newcomers to the language

Uniform initialization

- Braces are always ok
  - `int a{2};`
  - `Employee CEO{};`
  - `Employee newHire {John,today+1,salary};`
  - `vector<int> v {1,2,3,4};`
  - `vector<Employee> staff {CEO,newHire};`
- Consistent and easy to remember
- Can nest
  - ```
    vector<Employee> company { CEO,
                               newHire,
                               {Mary, today+1, salary}
                             };
    ```
    ```

shared_ptr and unique_ptr

- Stop managing memory yourself
  - Member variables or local objects you're just using for a calculation
  - Raw pointers are the wrong choice if lifetime is to be managed
    - Fine for observation/reaching eg parent->Invalidate();
- Best choice: solid objects, stack semantics
  - Even when passing to / returning from functions
  - RVO, move semantics
- Lowest overhead smart pointer: unique_ptr
  - Noncopyable, but movable
  - Plays well with collections (move it in, move it out)
- OK with ref counting overhead: shared_ptr
  - make_shared lowers overhead somewhat

Variadic

- Taking an unspecified number and type of arguments
- Function
  - printf
- Macro
  - Logging
- Templates
  - make_shared, make_unique

```
auto sp1 = make_shared<int>(2);
auto sp2 = make_shared<Employee>(John, today+1,
                                 salary);
```

History

- Variadic templates are in C++ 11
  - Needed for many valuable library features
  - Including make_shared
- Parts of C++ 11 appeared in VS 2010
  - And some parts slightly earlier in a feature pack for 2008
- Variadic templates were not actually implemented in Visual Studio until Visual Studio 2013
  - But features relying on them were implemented earlier
- Before that the library implementation faked them with macros
- Infinity was actually 10
  - And for performance reasons infinity was later lowered to 5
- Now that VC++ has variadic templates, your builds will be faster

std::tuple

- Like a std::pair, but any number of elements
- Saves writing little class or struct just to hold a clump of values
- Create with uniform initialization

```
std::tuple<int, std::string, double>
                        entry { 1, "Kate", 100.0 };
```

- Or use `std::make_tuple`
  - Makes `auto` possible
- To access or set values, use `std::get<position>(tupleinstance)`
- Has comparison operators etc already implemented

C++ Renaissance?

- Some of us never left
- Some great tech coming from Microsoft:
  - Writing Windows 8 store apps in C++/CX
  - Leveraging the GPU without learning another language using C++ AMP
  - More parity with managed languages
- Ask people what they don't like about C++
  - Almost every answer gets "that's different w/ C++ 11"
  - No denying there's a lot of punctuation, though
- What should you do next?
  - Get Visual Studio 2013
  - Try some C++ 11 and 14 features
  - Try writing a Windows Store app
  - Try using C++ AMP
  - www.gregcons.com/kateblog