C++11 in the Real World

Anthony Williams

Just Software Solutions Ltd
`http://www.justsoftwaresolutions.co.uk`

11th April 2013

Features that give "bang for the buck"

- Comparative C++11/C++03 code
- Key benefits
- Quick tutorial

Automatic Type Deduction

Automatic Type Deduction (I)

C++11:

```
auto x=find_answer();
```

C++03:

```
answer_type x=find_answer();
```

Automatic Type Deduction (II): Benefits

Using auto:

- Saves typing
- Eases maintenance
- Avoids conversions
- Allows instances of unnameable types

Automatic Type Deduction (III): Unnameable Types (I)

Lambdas:

```
auto x=[&](int i){
    some_local.do_stuff(i);
};
```

Automatic Type Deduction (IV): Unnameable Types (II)

Private members:

```
class X{
  class Y{};
public:
  static Y foo();
};
auto y=X::foo();
```

Automatic Type Deduction (V): Unnameable Types (III)

Dependent types:

```
template<typename T>
void foo(T const& t){
  auto x=t.bar();
}
```

Automatic Type Deduction (VI): Usage

`auto` works *almost* like a template type parameter in a function template argument declaration

Only "simple declarations" are supported

```
auto const* p=get_pointer();
auto&& t=make_temporary();
auto& r=some_lvalue;
```

Lambda Functions

Lambda Functions (I)

C++11:

```
auto it=std::find_if(
  c.begin(),c.end(),
  [&](entry const&x)
  {return x.data<some_local;});
```

Lambda Functions (II)

C++03:

```
struct DataLessThan{
  X t;
  DataLessThan(X t_): t(t_){}
bool operator()(entry const&x)
{return x.data<t;}
};
```

Lambda Functions (III)

```
// ...
IteratorType it=std::find_if(
  c.begin(),c.end(),
  DataLessThan(some_local));
```

Using Lambda Functions:

- Allows predicates to be defined at the point of use
- Can replace `std::bind` in many cases
- Simplifies code

Lambda Functions (V): Anatomy of a Lambda

Lambda Introducer

Parameter List

Return Type

Mutable Specifier

```
[=](int x,double y)->double mutable
{
    double temp=y*y;
    temp+=y*scale;
    return x*temp;
}
```

Lambda Body

Lambda Functions (VI): Lambda Introducers

- `[]` — no captures
- `[x]` — `x` captured by copy
- `[&x]` — `x` captured by reference
- `[this]` — `this` captured by copy
- `[=]` — default capture by copy
- `[&]` — default capture by reference

Lambda Functions (VII): Lambda Introducers

`[&,y,z]` — `y` and `z` captured by copy, others captured by reference

`[=,&y,&z]` — `y` and `z` captured by reference, others captured by copy

`[&x,y]` — `x` captured by reference, `y` captured by copy, no other captures

Lambda Functions (VIII): Mutable Lambdas

- Normally variables captured by copy are `const`
- In a `mutable` lambda they are not `const` and may be modified
- Changes are thus preserved between calls

Lambda Functions (IX): Lambda tricks (I)

```
std::function<int()>
  make_counter(
    int start,int step=1){
    return [=]() mutable{
        return start+=step;
    };
}
```

Lambda Functions (X): Lambda tricks (II)

```
void foo(){
  auto c1=make_counter(5);
  auto c2=make_counter(26,4);
  std::cout<<c1()<<"\n";
  std::cout<<c1()<<"\n";
  std::cout<<c2()<<"\n";
  std::cout<<c2()<<"\n";
}
```

Collection-based `for`

Collection-based `for` (I)

C++11:

```
for(auto x: get_data()){
  do_stuff(x);
}
```

Collection-based `for` (II)

C++03:

```
std::vector<my_type> v=get_data();
for(
  std::vector<my_type>::iterator
  it=v.begin(), end=v.end();
  it!=end;++it){
  do_stuff(*it);
}
```

Collection-based `for` (III): Benefits

Using Collection-based `for`:

- Eliminates boiler-plate
- Works with `break` and `continue`
- With `auto`, can avoid spelling out the types

Collection-based `for` (IV): Usage

```
for(item-type var-name :
      collection)
      body
```

- *item-type* can include `auto`
- The *collection* can be any expression: rvalue or lvalue

Collection-based `for` (V): Usage

- The compiler makes an iterator range [`x.begin()`,`x.end()`) or [`begin(x)`,`end(x)`)
- Each iteration, `var-name` is initialized with `*iter`

Multithreading Support

Multithreading Support (I)

C++11:

```
std::future<X> f=std::async(
  makeX,42,"hello");
std::cout<<f.get()<<std::endl;
```

C++03:

```
???
```

Multithreading Support (II)

- Prior to C++11 all multithreaded code relied on non-standard extensions
- C++11 provided a portable baseline, founded in the memory model

Multithreading Support (III)

- The memory model provides the low level guarantees
- Code that uses high level synchronization correctly should not ever have to worry about the details
- Data Races are Evil

Multithreading Support (IV)

- Use `std::async` and `std::future` by preference
- Use `std::thread`, `std::mutex` and `std::condition_variable` elsewhere
- Leave `std::atomic` to library implementors

Move Support

Move Support (I)

C++11:

```
std::vector<X> make_data();
std::vector<X> v;
void foo(){
  v=make_data();
}
```

Move Support (II)

C++03:

```
std::vector<X> make_data();
std::vector<X> v;
void foo(){
  v=make_data();
}
```

Move Support (III)

C++11:

```
std::vector<X> make_data();
std::vector<X> v;
void foo(){
  v=make_data(); // moves
}
```

Move Support (IV)

C++03:

```
std::vector<X> make_data();
std::vector<X> v;
void foo(){
  v=make_data(); // copies
}
```

Move Support (V)

C++03:

```
std::vector<X> make_data();
std::vector<X> v;
void foo(){
  std::vector<X> t=make_data();
  v.swap(t);
}
```

Move Support (VI)

The key to move support lies in **r-value references**.

```
X& X::operator=(X&& y){
  destroy_member_data();
  steal_member_data(y);
  return *this;
}
```

Move Support (VII): Details (I)

- A reference declared as *some-type* && is an rvalue reference

  Provided *some-type* is not an lvalue reference

- An rvalue reference binds **only** to rvalues

  But is itself an lvalue

- Rvalues are temporaries, literals and anything cast to an rvalue

Move Support (VIII): Details (II)

- `std::move(x)` is just `static_cast<TypeOfX&&>(x)`
- A constructor for type `X` that takes an `X&&` is a **move constructor**
- An assignment operator that takes an `X&&` is a **move-assignment operator**

Move Support (IX): Details (III)

- The compiler may generate move constructors and move-assignment operators (But it may not do what you need)
- Transfer ownership of all resources from the rvalue to `*this`
- Leave the rvalue "valid"

Move Support (X): Details (IV)

Generic copy-and-move-assignment:

```
X& operator=(X other){
  swap(other);
  return *this;
}
```

Leave the actual **moving** to the move constructor.

Move Support (XI): Perfect Forwarding (I)

- In a function `template`, a parameter of `T&&` is a "universal reference".
- If an lvalue of type `X` is passed, `T` is deduced to be `X&`
- If an rvalue of type `X` is passed, `T` is deduced to be `X`

Move Support (XII): Perfect Forwarding (II)

- `std::forward<T>(param)` returns `param` cast to `T&&`
- This works with reference collapsing rules to mean that rvalues passed to our function template are forwarded as rvalues, and lvalues forwarded as lvalues

Move Support (XIII): Perfect Forwarding (III)

```
template<typename T>
void foo(T&& t){
    bar(std::forward<T>(t));
}
```

Scoped Enumerations

Scoped Enumerations (I)

C++11:

```
enum class future_status{
  ready,timeout,deferred};
```

C++03:

```
namespace future_status{
enum Type{
  ready,timeout,deferred};
}
```

Scoped Enumerations (II): Key features

- Names are scoped: `std::future_status::ready`
- No implicit conversions ⇒ Only defined operations possible
- Can specify underlying type ⇒ Forward declarations possible

Scoped Enumerations (III)

- Great for bitfields: e.g. `std::launch`
- Cannot accidentally use as bitfields `std::errc::bad_address | std::errc::file_exists` is a compile error

Scoped Enumerations (IV): Syntax

`enum class` *name* : *underlying-type*

{

enumerators

};

- Can use `struct` instead of `class`
- The *underlying-type* is optional

Other great features

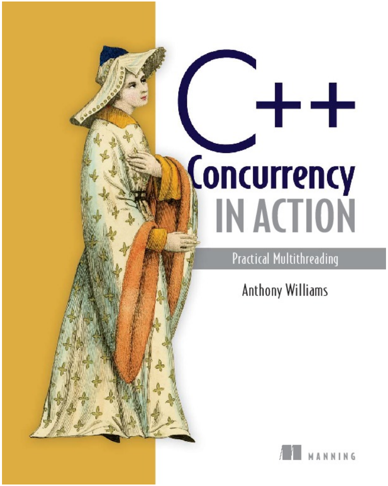
- `constexpr`
- `std::tuple`
- Variadic templates
- `std::shared_ptr`
- Template aliases
- Inline namespaces
- Regular expressions
- `std::chrono` timing facilities

Just::Thread

`just::thread` provides a complete implementation of the C++11 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.

`Just::Thread` **Pro: Actors was released this week**, with support for actors, single-object synchronization, message queues and concurrent hash maps. See `http://www.stdthread.co.uk/pro`

My Book



C++ Concurrency in Action: Practical Multithreading with the new C++ Standard.

`http://stdthread.com/book`

Image Credits

The images used in this presentation are all from Flickr. The names below are linked to the source image.

The automatic gear picture (CC-Attribution): Paulo Ordoveza
The lambda picture (CC-Attribution-Share-Alike): Matthew W. Jackson
The Star Wars figure collection (CC-Attribution): Simon Q
The threads picture (CC-Attribution): Christopher Bulle
The monthly list picture (CC-Attribution): Jamie
The removal van picture (CC-Attribution): summer photo hobby