C++0x Support in Visual Studio 2010

Or: The Ampersand Recovery && Reinvestment Act of 2010

Stephan T. Lavavej
Visual C++ Libraries Developer
stl@microsoft.com

C++0x Core Language Features In VS 2010 (VC10)

- Lambdas
  - Including `mutable` lambdas
- `auto`
  - Including multi-declarator `auto`
- `static_assert`
- Rvalue references
  - NOT including `*this`
- `decltype`
  - Including trailing return types

Lambdas: Motivation

- Function objects are very powerful
  - Encapsulating both behavior and state
- Function objects are obnoxious to define
  - Entire classes are inconveniently verbose
  - Worse, they're inconveniently nonlocal
- Libraries are insufficiently helpful
  - Syntactic contortions
  - Horrible compiler errors
  - Performance problems
- We need a core language feature for unnamed function objects

Lambdas: Hello, World!
Or: Mary Had A Little Lambda

- Lambda expressions implicitly define and construct unnamed function objects
  - Define an unnamed class
  - Construct an unnamed object
- Lambda expression:

```
for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
```

- Equivalent handwritten function object:

```
struct LambdaFunctor {
    void operator()(int n) const {
        cout << n << " ";
    }
};
for_each(v.begin(), v.end(), LambdaFunctor());
```

Lambdas: Multiple Statements

- Lambdas can contain arbitrary amounts of code:

```
for_each(v.begin(), v.end(), [](int n) {
    cout << n;

    if (n % 2 == 0) {
        cout << " even ";
    } else {
        cout << " odd ";
    }
});
```

Lambdas: Automatic Return Types

- Lambdas return `void` by default
- Lambdas of the form `{ return` *`expression`*`; }` get automatic return types
- Example:

```
transform(v.begin(), v.end(), front_inserter(d),
    [](int n) { return n * n * n; });
```

- Another example:

```
sort(v.begin(), v.end(),
    [](const string& l, const string& r) {
        return l.size() < r.size();
    }
);
```

Lambdas: Explicit Return Types

- Explicit return types go on the right:

```
transform(v.begin(), v.end(), front_inserter(d),
    [](int n) -> double {
        if (n % 2 == 0) {
            return n * n * n;
        } else {
            return n / 2.0;
        }
    });
```

- If you forget, the compiler will complain:

```
error C3499: a lambda that has been specified to have a
   void return type cannot return a value
```

Lambdas: Stateless Versus Stateful

- The empty *lambda-introducer* [ ] says "I am a stateless lambda", i.e. "I capture nothing"
- Within the *lambda-introducer*, you can specify a *capture-list*:

```
v.erase(remove_if(v.begin(), v.end(),
    [x, y](int n) { return x < n && n < y; }
    ), v.end());
```

- If you forget, the compiler will complain:

```
error C3493: 'x' cannot be implicitly captured as no
   default capture mode has been specified
```

- Lexical scope and conceptual scope are different

Lambdas: Stateful Versus Handwritten

- Lambda expression:

```
v.erase(remove_if(v.begin(), v.end(),
    [x, y](int n) { return x < n && n < y; }), v.end());
```

- Equivalent handwritten function object:

```
class LambdaFunctor {
public:
    LambdaFunctor(int a, int b) : m_a(a), m_b(b) { }
    bool operator()(int n) const {
        return m_a < n && n < m_b; }
private:
    int m_a;
    int m_b;
};
v.erase(remove_if(v.begin(), v.end(),
    LambdaFunctor(x, y)), v.end());
```

Lambda Value Captures: Good News, Bad News

- Good news
  - The function object can outlive the local variables that were captured to create it
- Potentially bad news
  - Some objects are expensive to copy
  - The captured copies can't be modified within the lambda
    - Because the function call operator is `const` by default
    - Solution: `mutable` lambdas, see next slide, but even so…
  - Updates to the captured copies won't be reflected in the local variables
  - Updates to the local variables won't be reflected in the captured copies

mutable Lambdas: C++ox Loves Recycling Keywords

```
int x = 1;
int y = 1;

for_each(v.begin(), v.end(),
    [x, y](int& r) mutable {
        const int old = r;
        r *= x * y;
        x = y;
        y = old;
    }
);
```

Lambda Reference Captures: I Promised There'd Be Ampersands

```
int x = 1;
int y = 1;

for_each(v.begin(), v.end(),
    [&x, &y](int& r) {
        const int old = r;
        r *= x * y;
        x = y;
        y = old;
    }
);
```

Lambdas: Reference Captures Versus Handwritten

```
#pragma warning(push)          // assignment operator
#pragma warning(disable: 4512) // could not be generated
class LambdaFunctor {
public:
    LambdaFunctor(int& a, int& b) : m_a(a), m_b(b) { }
    void operator()(int& r) const {
        const int old = r;
        r *= m_a * m_b;
        m_a = m_b;
        m_b = old;
    }
private:
    int& m_a;
    int& m_b;
};
#pragma warning(pop)
int x = 1;
int y = 1;
for_each(v.begin(), v.end(), LambdaFunctor(x, y));
```

Lambdas: Default Captures

- These lambdas are equivalent:

```
[x, y](int n) { return x < n && n < y; }
[=](int n) { return x < n && n < y; }
```

- So are these:

```
[&x, &y](int& r) {
    const int old = r; r *= x * y; x = y; y = old; }
[&](int& r) {
    const int old = r; r *= x * y; x = y; y = old; }
```

Lambdas: Mixing Value And Reference Captures

```
int sum = 0;
int product = 1;
int x = 1;
int y = 1;
for_each(v.begin(), v.end(),
    [=, &sum, &product](int& r) mutable {
        sum += r;
        if (r != 0) { product *= r; }
        const int old = r;
        r *= x * y;
        x = y;
        y = old;
    });
// Equivalent: [&, x, y]
// Equivalent: [&sum, &product, x, y]
```

Lambdas And function

- Lambdas produce ordinary function objects (with unspecified types) and can be stored in boost/tr1/std::function

```
void meow(const vector<int>& v,
const function<void (int)>& f) {
    for_each(v.begin(), v.end(), f);
    cout << endl;
}
meow(v, [](int n) { cout << n << " "; });
meow(v, [](int n) { cout << n * n << " "; });
function<void (int)> g = [](int n) {
    cout << n * n * n << " "; };
meow(v, g);
```

Lambdas And Standard Library Member Functions

- What's wrong with `mem_fn(&string::size)`?
- It's nonconformant to take the address of any Standard Library non-virtual member function
- C++03 17.4.4.4/2 permits additional overloads, making the address-of operator ambiguous
- `static_cast` can't disambiguate: C++03 17.4.4.4/2 also permits additional arguments with default values, which change the signature
- Lambdas for the win!
  - `[](const string& s) { return s.size(); }`

auto

```
map<string, string> m;
const regex r("(\\w+) (\\w+)");
for (string s; getline(cin, s); ) {
    smatch results;
    if (regex_match(s, results, r)) {
        m[results[1]] = results[2];
    }
}
for (auto i = m.begin(); i != m.end(); ++i) {
    cout << i->second << " are " << i->first << endl;
}
// Instead of map<string, string>::iterator
```

auto: How It Works And Why To Use It

- `auto` is powered by the template argument deduction rules
  - `const auto * p = foo` and `const auto& r = bar` work
- `auto`…
  - Reduces verbosity, allowing important code to stand out
  - Avoids type mismatches and the resulting truncation, etc. errors
  - Increases genericity, by allowing templates to be written that care less about the types of intermediate expressions
  - Deals with undocumented or unspeakable types, like lambdas

auto Lambdas

- auto can be used to name lambdas, allowing them to be reused
  - Doesn't defeat the point of lambdas; they're still local

```
template <typename T, typename Predicate>
void keep_if(vector<T>& v, Predicate pred) {
    auto notpred = [&](const T& t) { return !pred(t); };
    v.erase(remove_if(v.begin(), v.end(), notpred), v.end());
}
auto prime = [](const int n) -> bool {
    if (n < 2) { return false; }
    for (int i = 2; i <= n / i; ++i) {
        if (n % i == 0) { return false; }
    }
    return true;
};
keep_if(a, prime);
keep_if(b, prime);
```

static_assert (1/2)

```
C:\Temp>type static.cpp
#include <type_traits>
using namespace std;

template <typename T> struct Kitty {
  static_assert(is_integral<T>::value,
    "Kitty<T> requires T to be an integral type.");
};

int main() {
  Kitty<int> peppermint;

  Kitty<double> jazz;
}
```

static_assert (2/2)

```
C:\Temp>cl /EHsc /nologo /W4 static.cpp
static.cpp
static.cpp(6) : error C2338: Kitty<T> requires T to be
   an integral type.
        static.cpp(12) : see reference to class template
   instantiation 'Kitty<T>' being compiled
        with
        [
            T=double
        ]
```

Rvalue References

- Rvalue references enable two different things
  - Move semantics: for performance
  - Perfect forwarding: for genericity
- The move semantics and perfect forwarding patterns are easy to follow
- But they're powered by new initialization, overload resolution, template argument deduction, and "reference collapsing" rules
- It's C++'s usual deal: complex rules let you write beautifully simple and powerful code
- Take an hour to read this: http://blogs.msdn.com/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx

Rvalue References: Move Semantics, Part I

```
class remote_integer {
public: // ... copy ctor, copy assign, etc. omitted ...

    remote_integer(remote_integer&& other) {
        m_p = other.m_p;
        other.m_p = NULL;
    }

    remote_integer& operator=(remote_integer&& other) {
        if (this != &other) {
            delete m_p;
            m_p = other.m_p;
            other.m_p = NULL;
        }
        return *this;
    }

private:
    int * m_p;
};
```

Rvalue References: Move Semantics, Part II

```
#include <utility>

class remote_point {
public: // implicitly defined copy ctor and copy assign are okay

    remote_point(remote_point&& other)
        : m_x(std::move(other.m_x)),
          m_y(std::move(other.m_y)) { }

    remote_point& operator=(remote_point&& other) {
        m_x = std::move(other.m_x);
        m_y = std::move(other.m_y);
        return *this;
    }

private:
    remote_integer m_x;
    remote_integer m_y;
};
```

Rvalue References: Perfect Forwarding

```
#include <utility>

void inner(int&, int&) {
    cout << "inner(int&, int&)" << endl;
}
void inner(int&, const int&) {
    cout << "inner(int&, const int&)" << endl;
}
void inner(const int&, int&) {
    cout << "inner(const int&, int&)" << endl;
}
void inner(const int&, const int&) {
    cout << "inner(const int&, const int&)" << endl;
}

template <typename T1, typename T2> void outer(T1&& t1, T2&& t2) {
    inner(std::forward<T1>(t1), std::forward<T2>(t2));
}
```

decltype: Perfect Forwarding's Best Friend Forever

```
struct Plus {
    template <typename T, typename U>
    auto operator()(T&& t, U&& u) const
    -> decltype(forward<T>(t) + forward<U>(u)) {
        return forward<T>(t) + forward<U>(u);
    }
};
vector<int> i, j, k;
transform(i.begin(), i.end(), j.begin(),
    back_inserter(k), Plus());
vector<string> s, t, u;
transform(s.begin(), s.end(), t.begin(),
    back_inserter(u), Plus());
```

C++0x Standard Library Features In VS 2010 (VC10)

- Rvalue references
  - Supply-side: `vector`, etc. gains move ctors and move assign ops
  - Demand-side: `vector` reallocation, etc. exploits move semantics
  - Perfect forwarding: `make_shared<T>()`, etc.
  - Goodbye `auto_ptr`: Hello `unique_ptr`
- New member functions: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- New algorithms: `copy_if()`, `is_sorted()`, etc.
- Singly linked lists: `forward_list`
- Code conversions: `<codecvt>`
- Exception propagation: `exception_ptr`
- `<system_error>`
- Non-Standard Dinkum Allocators Library `<allocators>`
- Tons of `<random>` updates
- And more!

make_shared<T>()

- VS 2008 SP1 (VC9 SP1):
  - `shared_ptr<T> sp(new T(args));`
  - `shared_ptr<T> sp(new T(args), del, alloc);`
- VS 2010 (VC10):
  - `auto sp = make_shared<T>(args);`
  - `auto sp = allocate_shared<T>(alloc, args);`
- Convenient!
  - Mentions the type T once instead of twice
- Robust!
  - Avoids the Classic Unnamed `shared_ptr` Leak
- Efficient!
  - Performs one dynamic memory allocation instead of two

cbegin(), cend(), crbegin(), crend()

```
vector<int> v;

for (auto i = v.begin(); i != v.end(); ++i) {
    // i is vector<int>::iterator
}

for (auto i = v.cbegin(); i != v.cend(); ++i) {
    // i is vector<int>::const_iterator
}
```

unique_ptr

- Supersedes auto_ptr, which is now deprecated
- Noncopyable but movable

```
unique_ptr<Cat> c(new Cat);
unique_ptr<Dog> d;
d.reset(new Dog);
unique_ptr<Monster> m_src(new Monster);
unique_ptr<Monster> m_dest(move(m_src));
```

- Works just fine in containers

```
vector<unique_ptr<Animal>> v;
v.push_back(move(c));
v.push_back(move(d));
v.push_back(move(m_dest));
for (auto i = v.cbegin(); i != v.cend(); ++i) {
    (*i)->make_noise();
}
```

New Algorithms

- <algorithm>
  - bool all_of/any_of/none_of(InIt, InIt, Pred)
  - InIt find_if_not(InIt, InIt, Pred)
  - OutIt copy_n(InIt, Size, OutIt)
  - OutIt copy_if(InIt, InIt, OutIt, Pred)
  - bool is_partitioned(InIt, InIt, Pred)
  - pair<Out1, Out2> partition_copy(InIt, InIt, Out1, Out2, Pred)
  - FwdIt partition_point(FwdIt, FwdIt, Pred)
  - bool is_sorted(FwdIt, FwdIt, Comp?)
  - FwdIt is_sorted_until(FwdIt, FwdIt, Comp?)
  - bool is_heap(RanIt, RanIt, Comp?)
  - RanIt is_heap_until(RanIt, RanIt, Comp?)
  - pair<FwdIt, FwdIt> minmax_element(FwdIt, FwdIt, Comp?)
- <numeric>
  - void iota(FwdIt, FwdIt, T)
- <iterator>
  - InIt next(InIt, Distance)
  - BidIt prev(BidIt, Distance)

_ITERATOR_DEBUG_LEVEL: The New World Order

- Non-Standard, but important
- VS 2005-2008 (VC8-9):
  - `_SECURE_SCL == 0, 1`
  - `_HAS_ITERATOR_DEBUGGING == 0, 1`
- VS 2010 Beta 1 (VC10 Beta 1):
  - `_ITERATOR_DEBUG_LEVEL == 0, 1, 2`
- Later:
  - `_ITERATOR_DEBUG_LEVEL` will default to `0` in release
  - `#pragma detect_mismatch` will detect mismatch at link time, preventing ODR violations from causing incomprehensible crashes
    - Will NOT perform general ODR validation

Questions?

- C++ Standardization Committee
  - N2857: March 2009 C++0x Working Draft
  - N2869: March 2009 C++0x Core Language Features
  - N2870: March 2009 C++0x Standard Library Features
- Dinkumware
  - Dinkum Allocators Library
  - Dinkum Conversions Library
- VCBlog
  - C++0x Features in VC10
    - Part 1: Lambdas, auto, and static_assert
    - Part 2: Rvalue References
    - Part 3: decltype
- stl@microsoft.com

Bonus Slides!

Nullary Lambdas

- Empty parentheses can be elided from lambdas taking no arguments:

```
vector<int> v;
int i = 0;
generate_n(back_inserter(v), 10, [&] { return i++; });
```

- As opposed to `[&]() { return i++; }`
- Note that due to how lambdas work syntactically, empty parentheses cannot be omitted when you have `mutable` or `-> ReturnType`

Lambdas: Capturing Data Members

- Local variables can be captured
- But data members aren't local variables
- You can explicitly capture `this` with `[this]`
  - And then use `m_foo` or `this->m_foo` as usual
- Or, you can implicitly capture `this` with `[=]`
  - Triggered by using `m_foo` or `this->m_foo`
- `this` can also be implicitly captured with `[&]`
  - But `this` is always captured by value
  - `[&this]` is forbidden

unique_ptr Example (1/5)

```
C:\Temp>type meow.cpp
#include <iostream>
#include <memory>
#include <ostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;
```

unique_ptr Example (2/5)

```
class Animal {
public:
    Animal() { }
    virtual ~Animal() { }

    void make_noise() const {
        cout << "This " << name()
            << " says " << sound() << "." << endl;
    }

private:
    Animal(const Animal&);
    Animal& operator=(const Animal&);

    virtual string name() const = 0;
    virtual string sound() const = 0;
};
```

unique_ptr Example (3/5)

```
class Cat : public Animal {
private:
    virtual string name() const { return "kitty"; }
    virtual string sound() const { return "meow"; }
};

class Dog : public Animal {
private:
    virtual string name() const { return "puppy"; }
    virtual string sound() const { return "bow-wow"; }
};

class Monster : public Animal {
private:
    virtual string name() const { return "floating eye"; }
    virtual string sound() const { return "*paralyzing gaze*"; }
};
```

unique_ptr Example (4/5)

```
int main() {
    unique_ptr<Cat> c(new Cat);

    unique_ptr<Dog> d;
    d.reset(new Dog);

    unique_ptr<Monster> m_src(new Monster);
    unique_ptr<Monster> m_dest(move(m_src));

    vector<unique_ptr<Animal>> v;

    v.push_back(move(c));
    v.push_back(move(d));
    v.push_back(move(m_dest));

    for (auto i = v.cbegin(); i != v.cend(); ++i) {
        (*i)->make_noise();
    }
}
```

unique_ptr Example (5/5)

```
C:\Temp>cl /EHsc /nologo /W4 meow.cpp
meow.cpp

C:\Temp>meow
This kitty says meow.
This puppy says bow-wow.
This floating eye says *paralyzing gaze*.
```