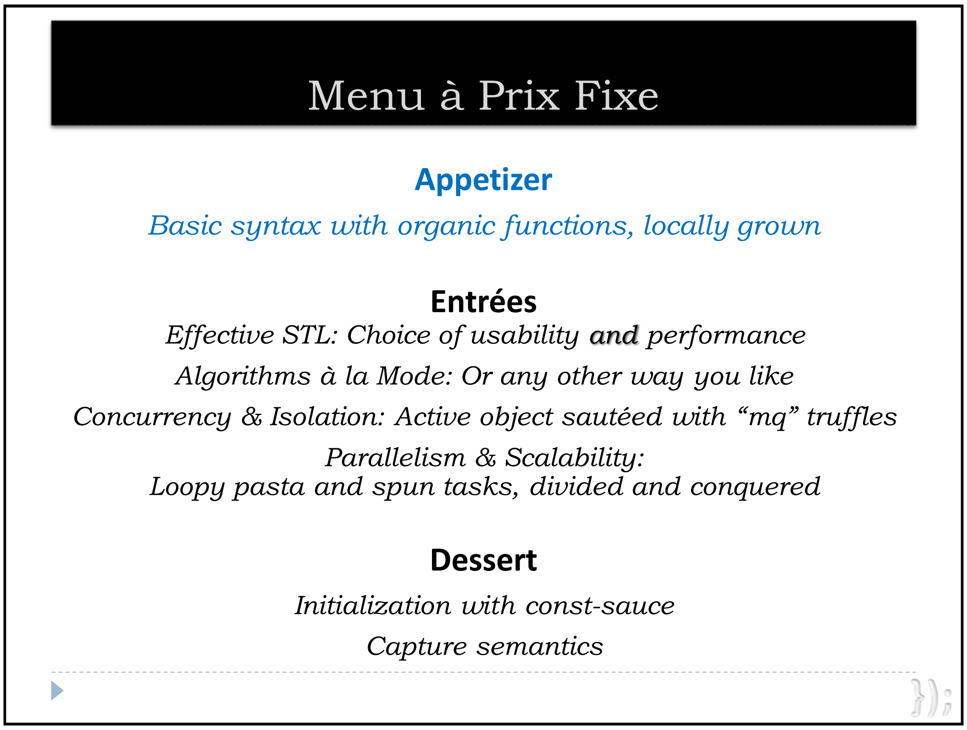
Herb Sutter
Software Development Consultant
www.gotw.ca/training



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

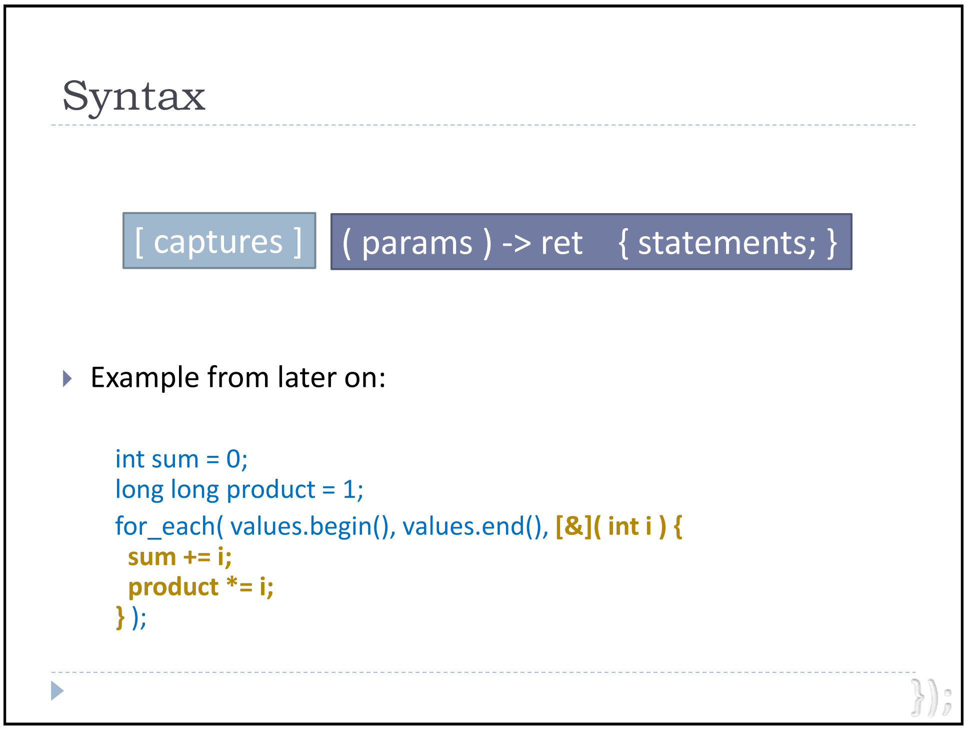
Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

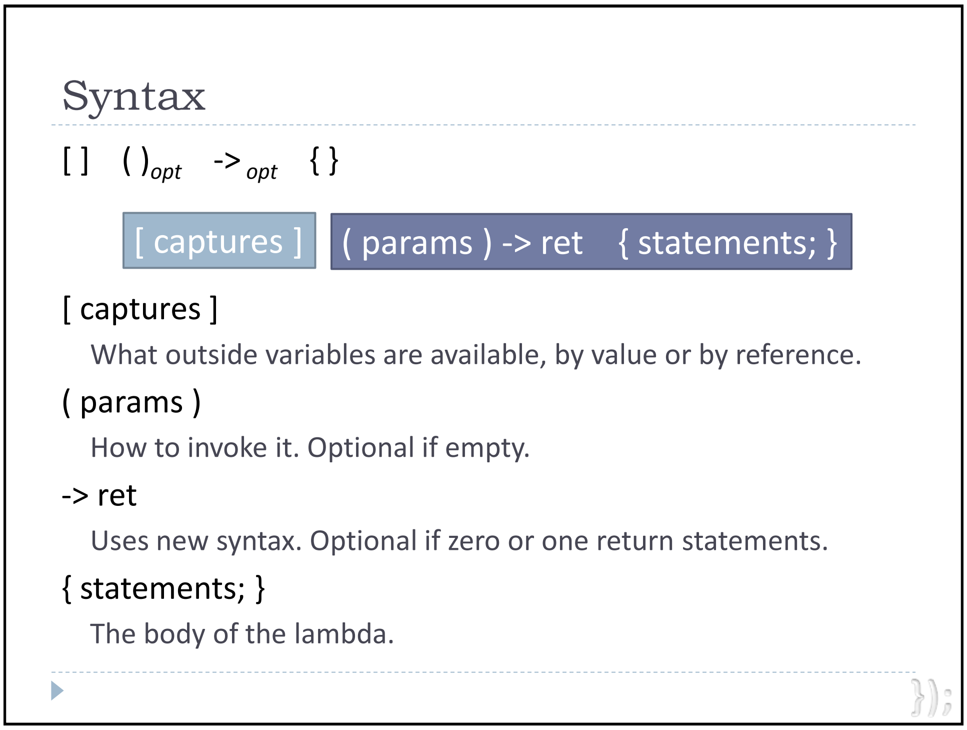


Syntax

[captures] (params) -> ret { statements; }

- Example from later on:

```
int sum = 0;
long long product = 1;
for_each( values.begin(), values.end(), [&]( int i ) {
  sum += i;
  product *= i;
} );
```



Syntax

[ ]   ( )$_{opt}$ -> $_{opt}$ { }

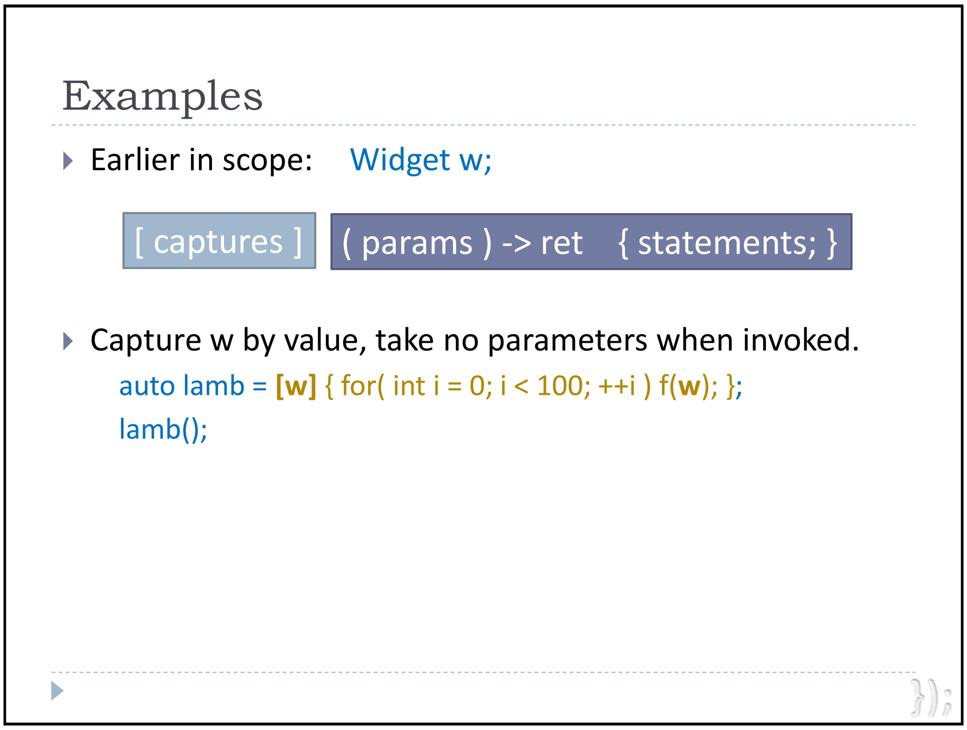
[captures] (params) -> ret { statements; }

[captures]
  What outside variables are available, by value or by reference.

(params)
  How to invoke it. Optional if empty.

-> ret
  Uses new syntax. Optional if zero or one return statements.

{ statements; }
  The body of the lambda.



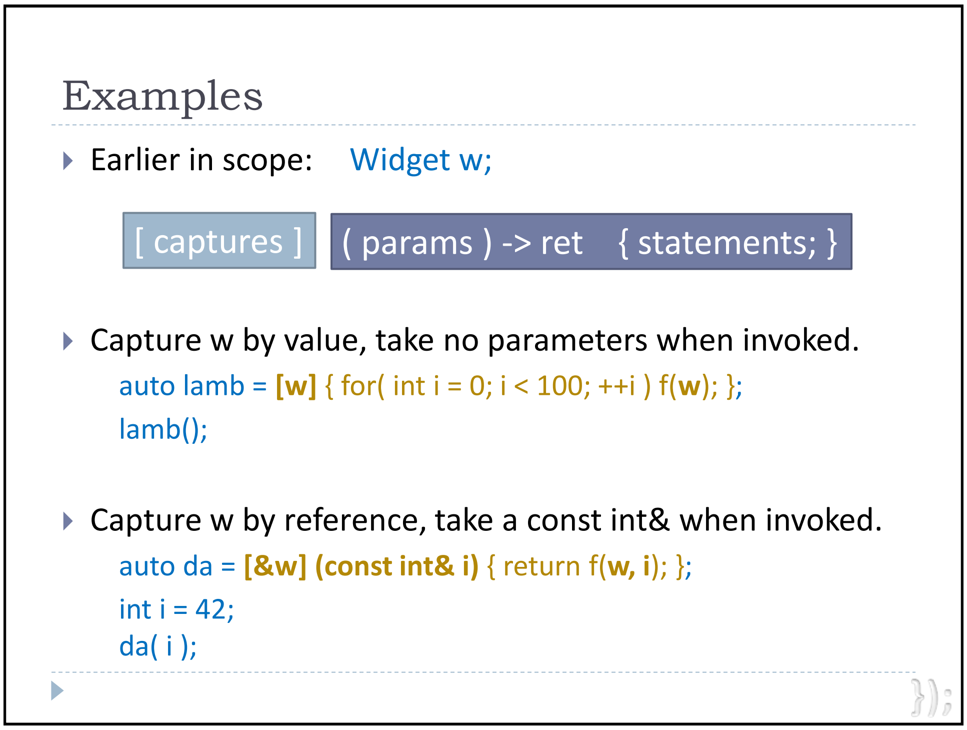
Examples

- Earlier in scope: Widget w;

[captures] (params) -> ret { statements; }

- Capture w by value, take no parameters when invoked.

```
auto lamb = [w] { for( int i = 0; i < 100; ++i ) f(w); };
lamb();
```



Examples

- Earlier in scope: Widget w;

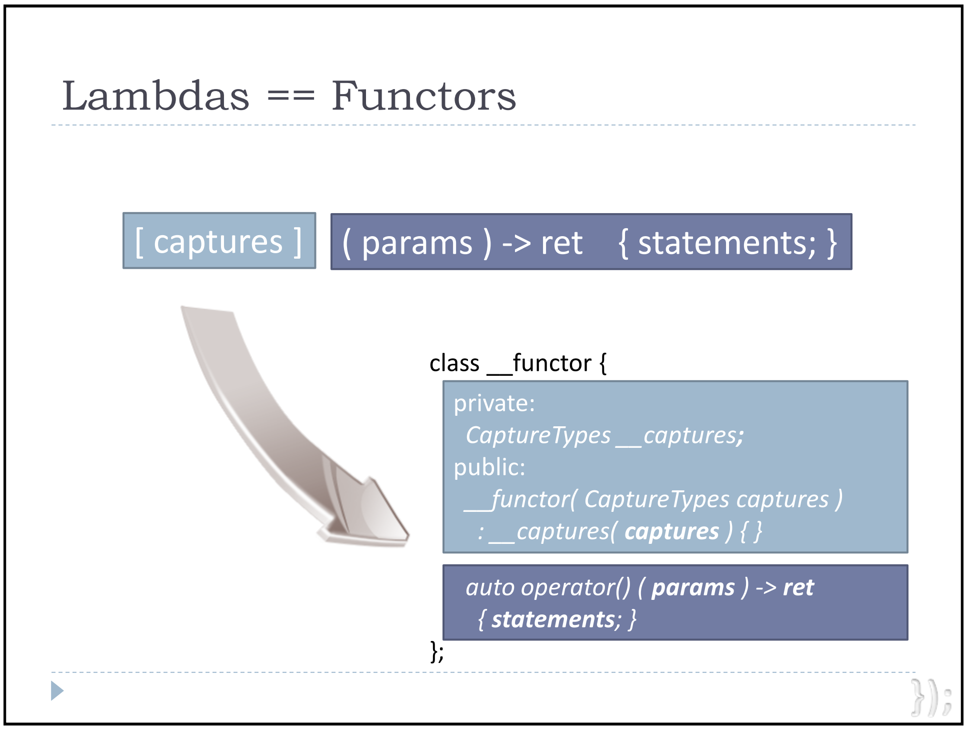
[captures] (params) -> ret { statements; }

- Capture w by value, take no parameters when invoked.

```
auto lamb = [w] { for( int i = 0; i < 100; ++i ) f(w); };
lamb();
```

- Capture w by reference, take a const int& when invoked.

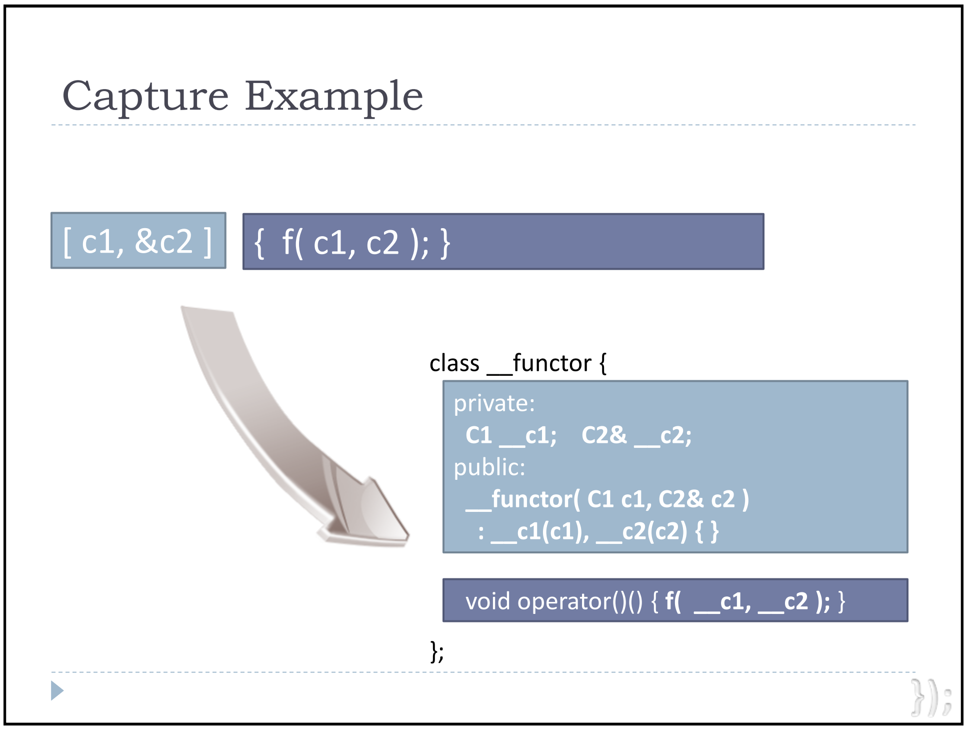
```
auto da = [&w] (const int& i) { return f(w, i); };
int i = 42;
da( i );
```



Lambdas == Functors

```
[ captures ]  ( params ) -> ret   { statements; }
```

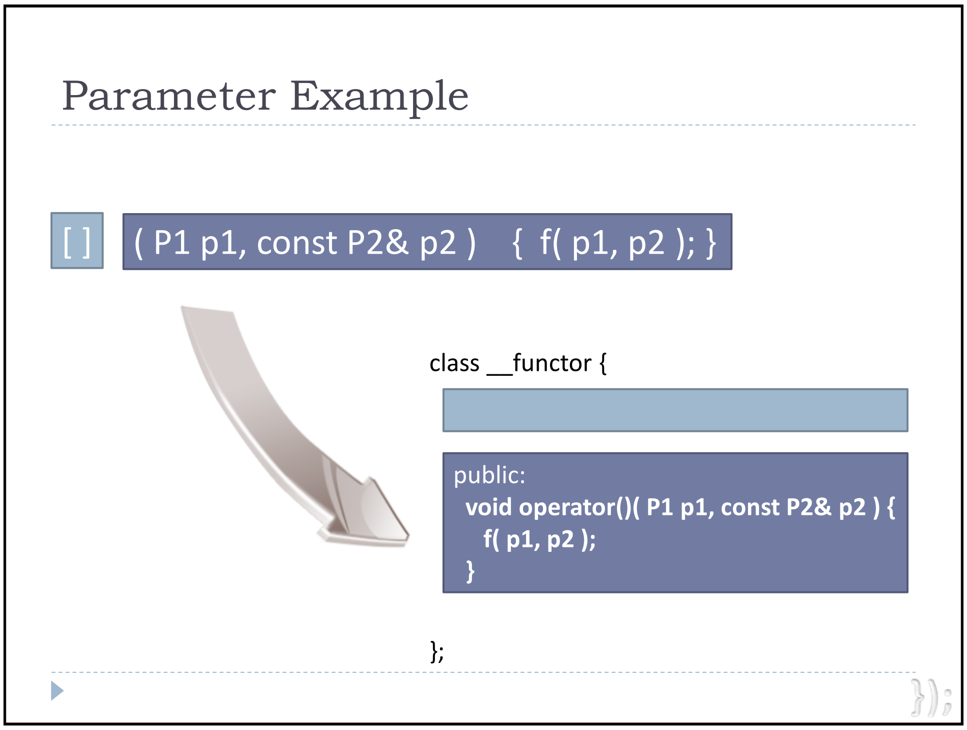
```
class __functor {
  private:
    CaptureTypes __captures;
  public:
    __functor( CaptureTypes captures )
      : __captures( captures ) { }

    auto operator() ( params ) -> ret
      { statements; }
};
```



Capture Example

```
[ c1, &c2 ]  {  f( c1, c2 ); }
```

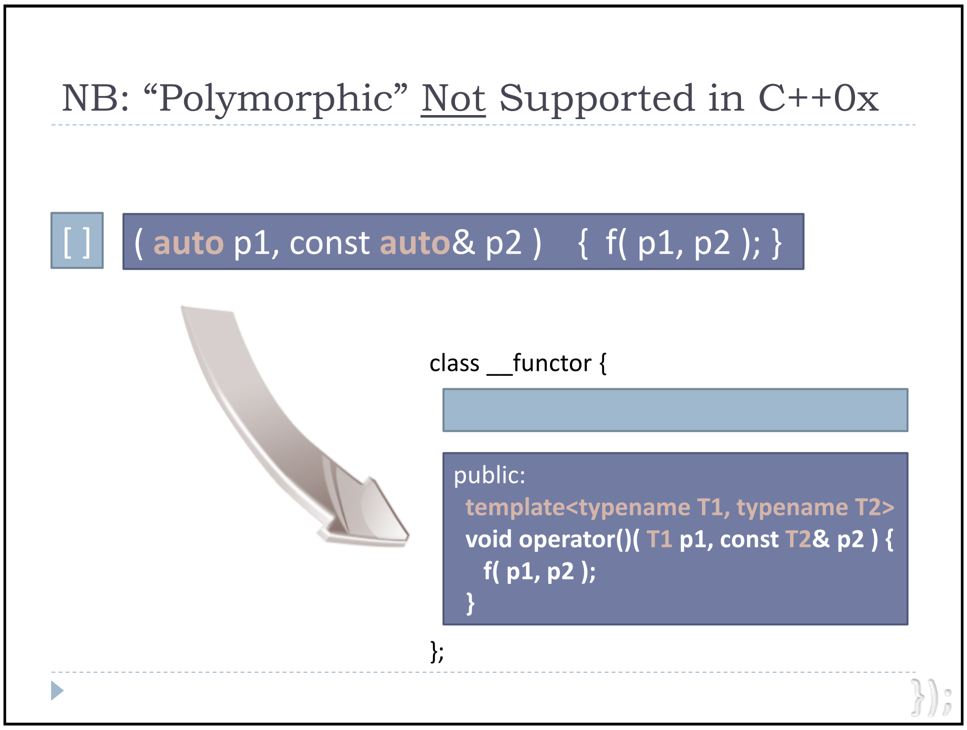
```
class __functor {
  private:
    C1 __c1;    C2& __c2;
  public:
    __functor( C1 c1, C2& c2 )
      : __c1(c1), __c2(c2) { }

    void operator()() { f( __c1, __c2 ); }
};
```



Parameter Example

```
[ ]  ( P1 p1, const P2& p2 )   { f( p1, p2 ); }
```

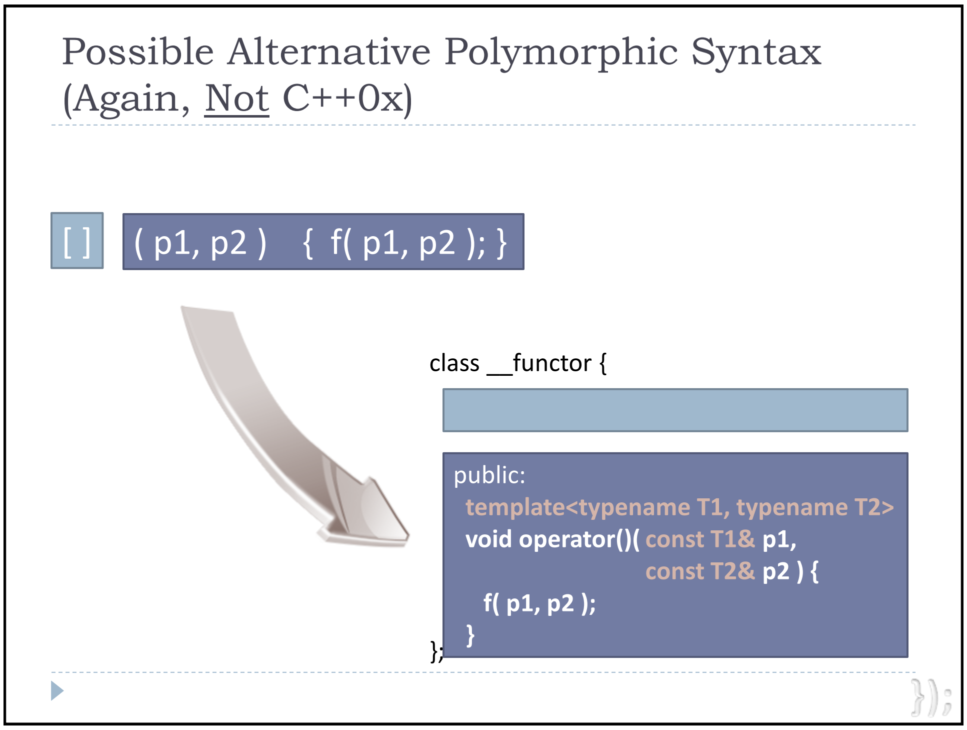
```
class __functor {

public:
  void operator()( P1 p1, const P2& p2 ) {
    f( p1, p2 );
  }

};
```



NB: "Polymorphic" Not Supported in C++0x

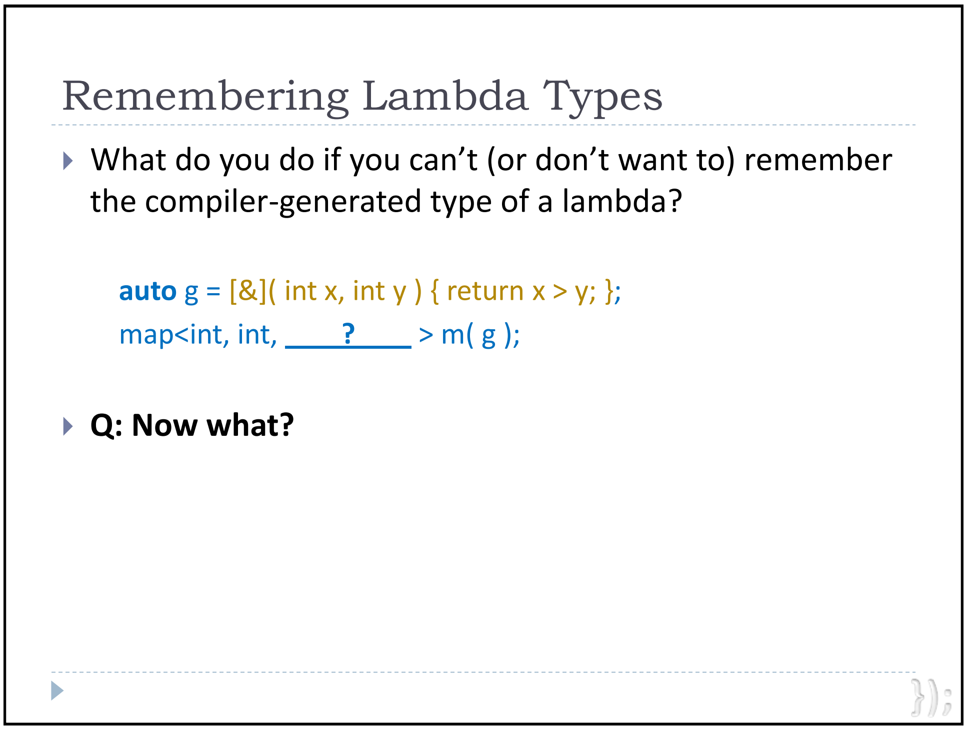
```
[ ]  ( auto p1, const auto& p2 )   { f( p1, p2 ); }
```

```
class __functor {

public:
  template<typename T1, typename T2>
  void operator()( T1 p1, const T2& p2 ) {
    f( p1, p2 );
  }
};
```



Possible Alternative Polymorphic Syntax (Again, Not C++0x)

```
[ ]  ( p1, p2 )   { f( p1, p2 ); }

class __functor {

public:
  template<typename T1, typename T2>
  void operator()( const T1& p1,
                   const T2& p2 ) {
    f( p1, p2 );
  }
};
```

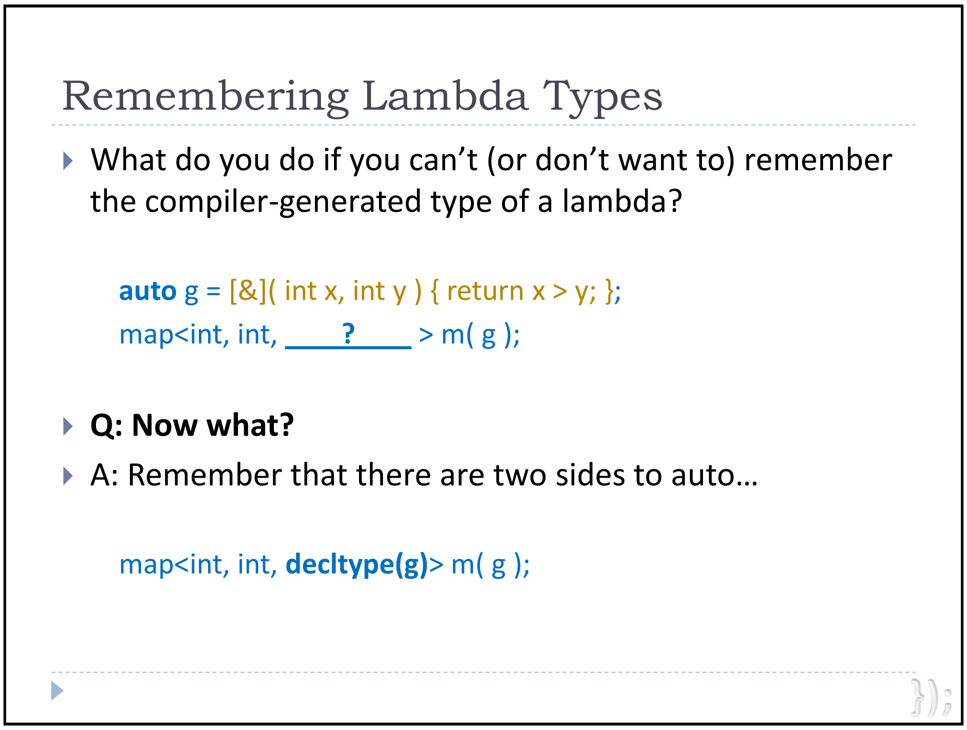


Remembering Lambda Types

- What do you do if you can’t (or don’t want to) remember the compiler-generated type of a lambda?

```
auto g = [&]( int x, int y ) { return x > y; };
map<int, int, ___?___ > m( g );
```

- **Q: Now what?**



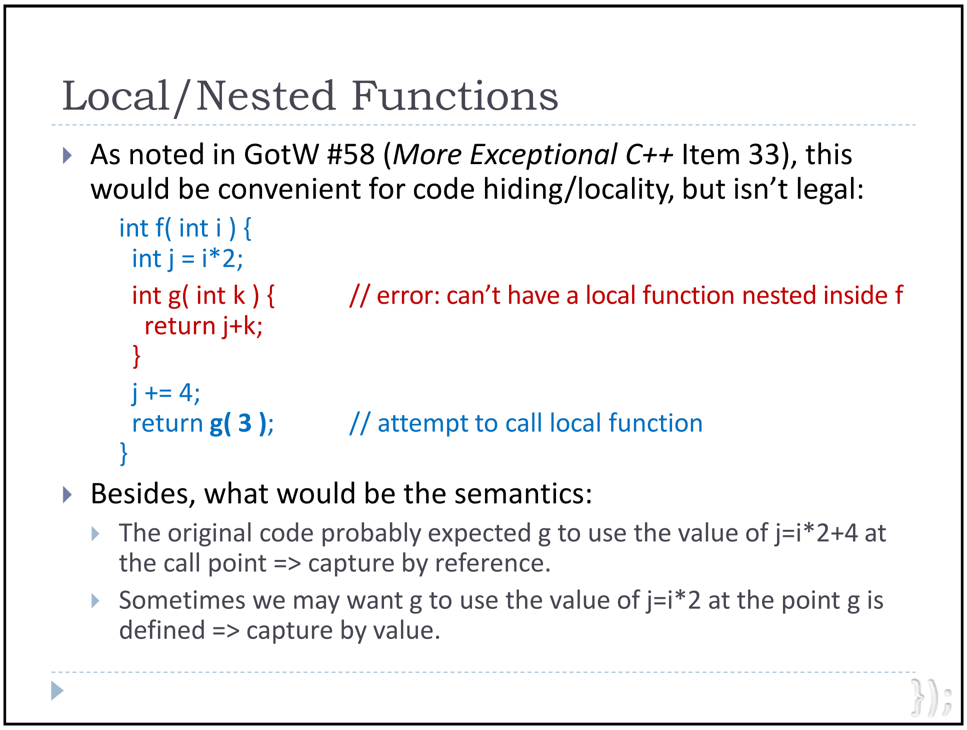
Remembering Lambda Types

- What do you do if you can't (or don't want to) remember the compiler-generated type of a lambda?

```
auto g = [&]( int x, int y ) { return x > y; };
map<int, int, ____?____ > m( g );
```

- **Q: Now what?**
- A: Remember that there are two sides to auto…

```
map<int, int, decltype(g)> m( g );
```



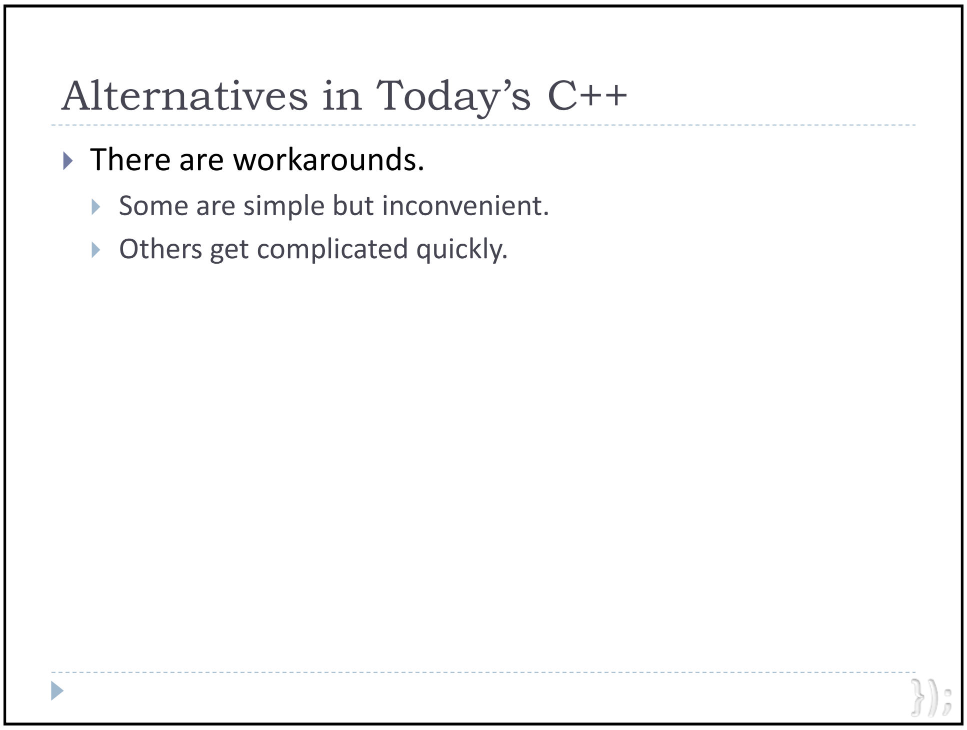
Local/Nested Functions

- As noted in GotW #58 (*More Exceptional C++* Item 33), this would be convenient for code hiding/locality, but isn't legal:

```
int f( int i ) {
  int j = i*2;
  int g( int k ) {          // error: can't have a local function nested inside f
    return j+k;
  }
  j += 4;
  return g( 3 );            // attempt to call local function
}
```

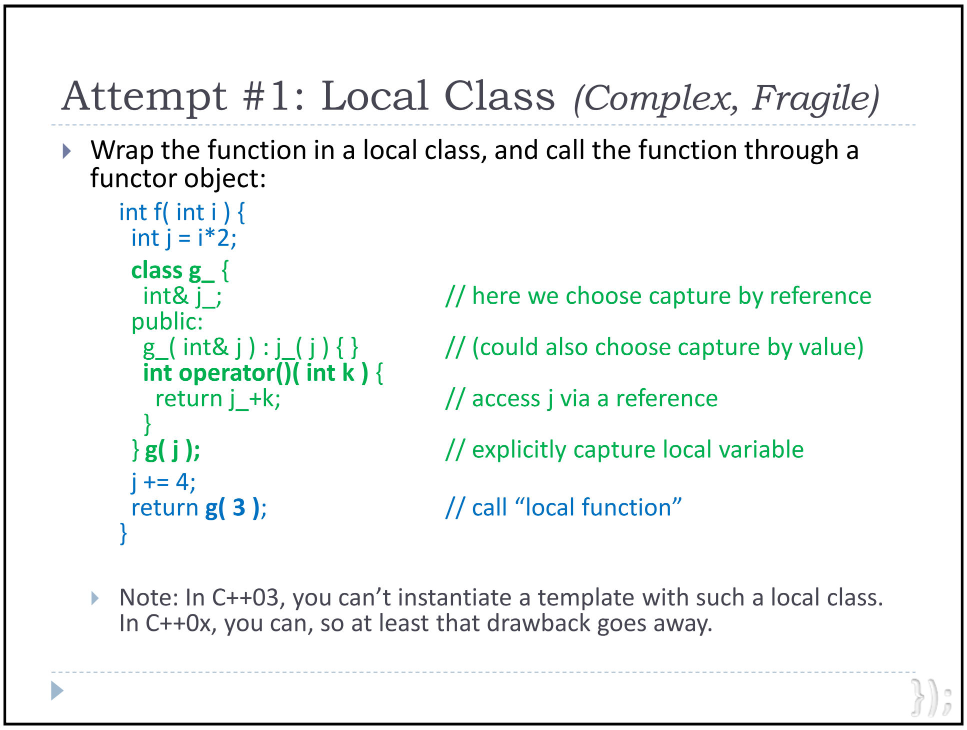
- Besides, what would be the semantics:
  - The original code probably expected g to use the value of j=i*2+4 at the call point => capture by reference.
  - Sometimes we may want g to use the value of j=i*2 at the point g is defined => capture by value.

Herb Sutter
Software Development Consultant
www.gotw.ca/training



Alternatives in Today's C++

- There are workarounds.
  - Some are simple but inconvenient.
  - Others get complicated quickly.

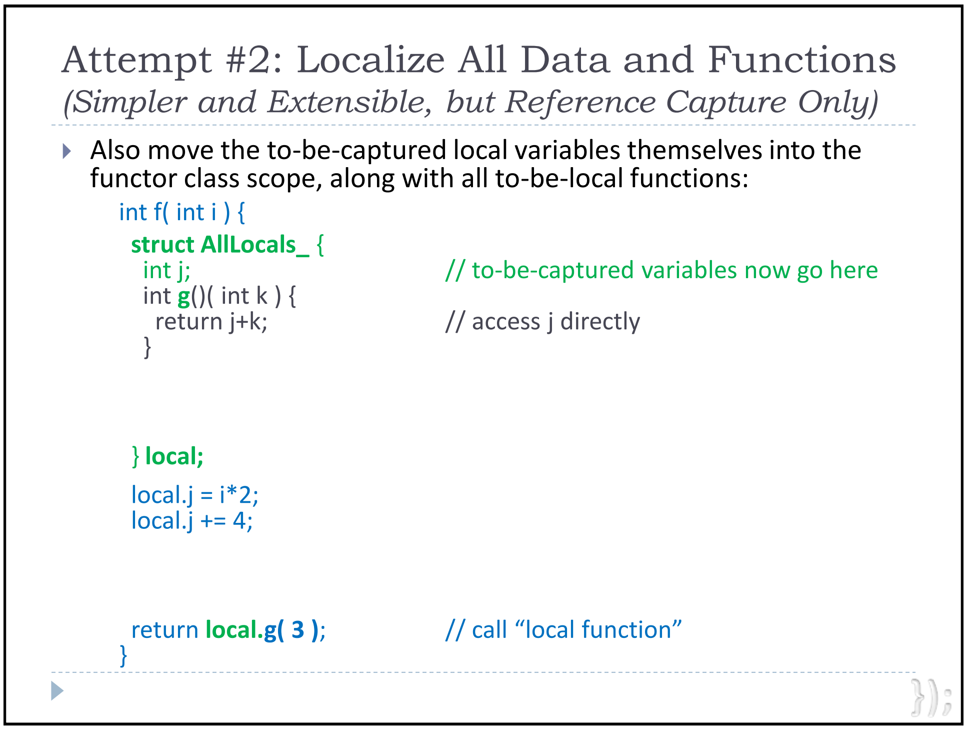


Attempt #1: Local Class *(Complex, Fragile)*

- Wrap the function in a local class, and call the function through a functor object:

```
int f( int i ) {
  int j = i*2;
  class g_ {
    int& j_;                     // here we choose capture by reference
  public:
    g_( int& j ) : j_( j ) { }   // (could also choose capture by value)
    int operator()( int k ) {
      return j_+k;               // access j via a reference
    }
  } g( j );                      // explicitly capture local variable
  j += 4;
  return g( 3 );                 // call "local function"
}
```

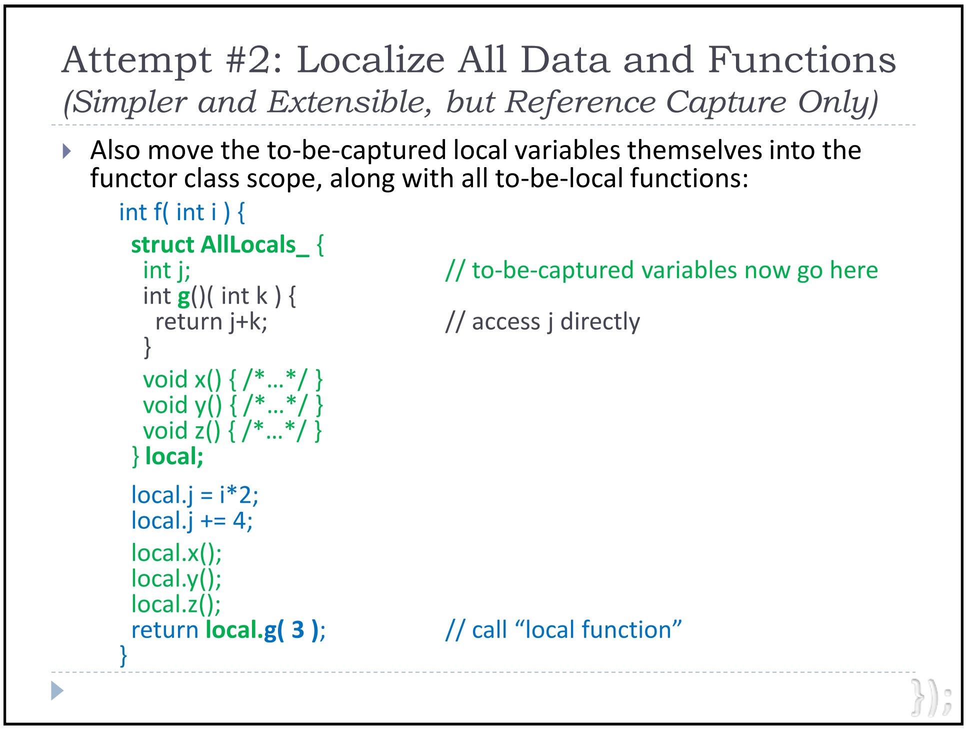
  - Note: In C++03, you can't instantiate a template with such a local class. In C++0x, you can, so at least that drawback goes away.



Attempt #2: Localize All Data and Functions
(Simpler and Extensible, but Reference Capture Only)

- Also move the to-be-captured local variables themselves into the functor class scope, along with all to-be-local functions:

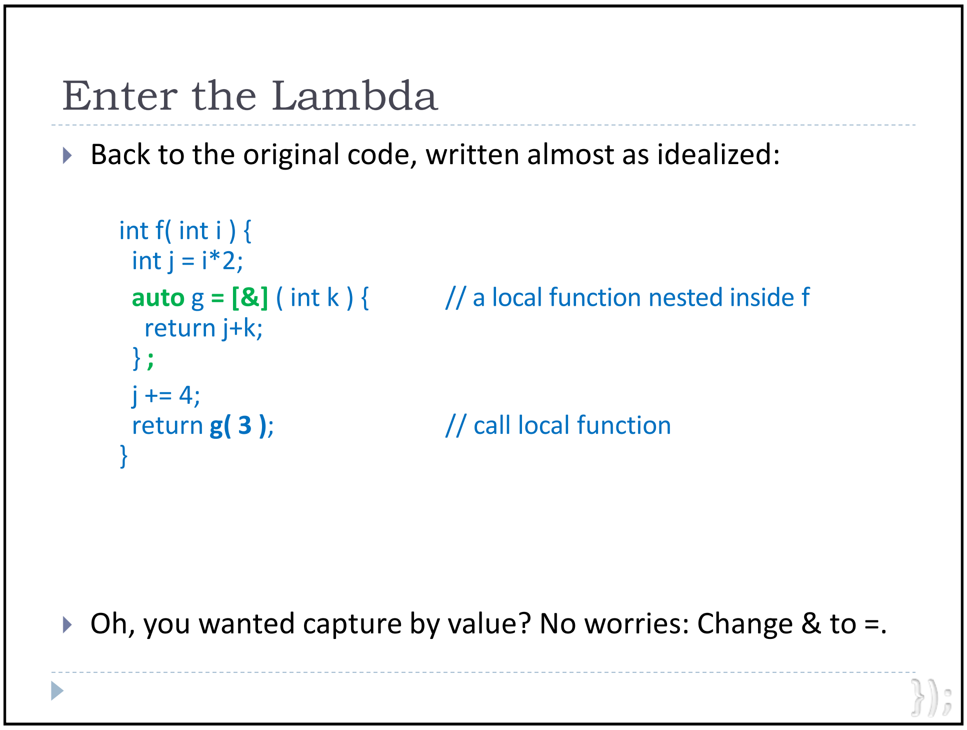
```
int f( int i ) {
  struct AllLocals_ {
    int j;                     // to-be-captured variables now go here
    int g()( int k ) {
      return j+k;              // access j directly
    }



  } local;

  local.j = i*2;
  local.j += 4;



  return local.g( 3 );         // call "local function"
}
```



Attempt #2: Localize All Data and Functions
(Simpler and Extensible, but Reference Capture Only)

- Also move the to-be-captured local variables themselves into the functor class scope, along with all to-be-local functions:

```
int f( int i ) {
  struct AllLocals_ {
    int j;                     // to-be-captured variables now go here
    int g()( int k ) {
      return j+k;              // access j directly
    }
    void x() { /*…*/ }
    void y() { /*…*/ }
    void z() { /*…*/ }
  } local;

  local.j = i*2;
  local.j += 4;

  local.x();
  local.y();
  local.z();
  return local.g( 3 );         // call "local function"
}
```



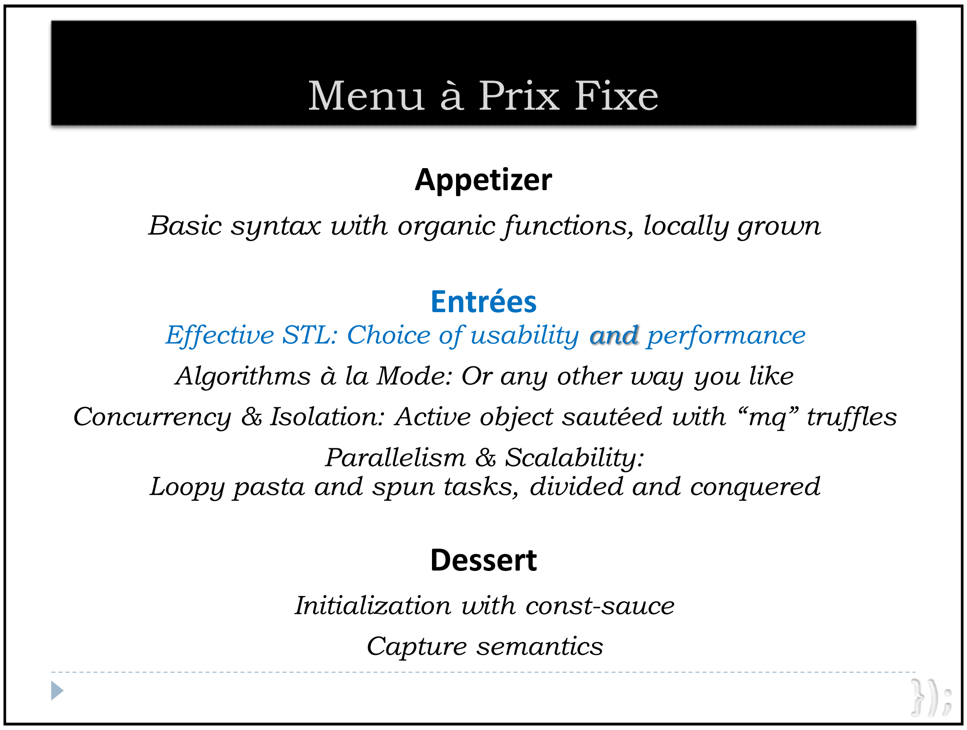
Enter the Lambda

- Back to the original code, written almost as idealized:

```
int f( int i ) {
  int j = i*2;
  auto g = [&] ( int k ) {          // a local function nested inside f
    return j+k;
  };
  j += 4;
  return g( 3 );                    // call local function
}
```

- Oh, you wanted capture by value? No worries: Change & to =.

});



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

Concurrency & Isolation: Active object sautéed with "mq" truffles

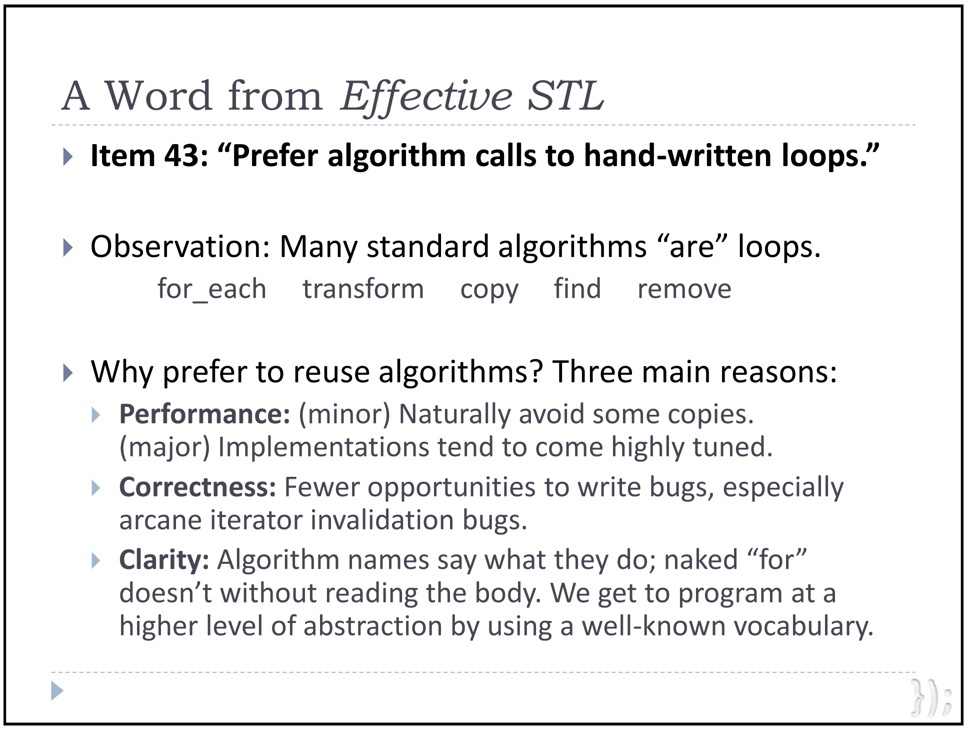
Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

});

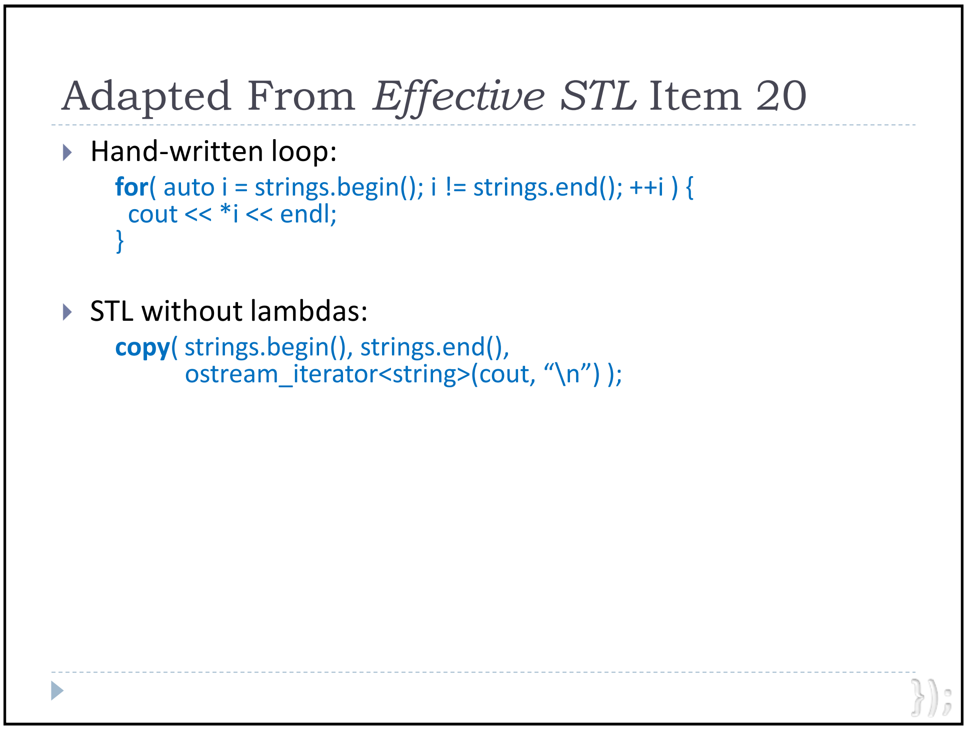


A Word from *Effective STL*

- **Item 43: "Prefer algorithm calls to hand-written loops."**
- Observation: Many standard algorithms "are" loops.

  for_each   transform   copy   find   remove

- Why prefer to reuse algorithms? Three main reasons:
  - **Performance:** (minor) Naturally avoid some copies.
    (major) Implementations tend to come highly tuned.
  - **Correctness:** Fewer opportunities to write bugs, especially arcane iterator invalidation bugs.
  - **Clarity:** Algorithm names say what they do; naked "for" doesn't without reading the body. We get to program at a higher level of abstraction by using a well-known vocabulary.



Adapted From *Effective STL* Item 20

- Hand-written loop:

```
for( auto i = strings.begin(); i != strings.end(); ++i ) {
  cout << *i << endl;
}
```

- STL without lambdas:

```
copy( strings.begin(), strings.end(),
      ostream_iterator<string>(cout, "\n") );
```

Adapted From *Effective STL* Item 20

- Hand-written loop:

```
for( auto i = strings.begin(); i != strings.end(); ++i ) {
  cout << *i << endl;
}
```

- STL without lambdas:

```
copy( strings.begin(), strings.end(),
          ostream_iterator<string>(cout, "\n") );
```

- STL with lambdas:

```
for_each( strings.begin(), strings.end(), []( string& s ) {
  cout << s << endl;
} );
```

- Q: Which is more self-documenting: **for**? **copy**? or **for_each**?

News Flash: *"Most Loop Bodies Aren't One-Liners!"* Film at 11…

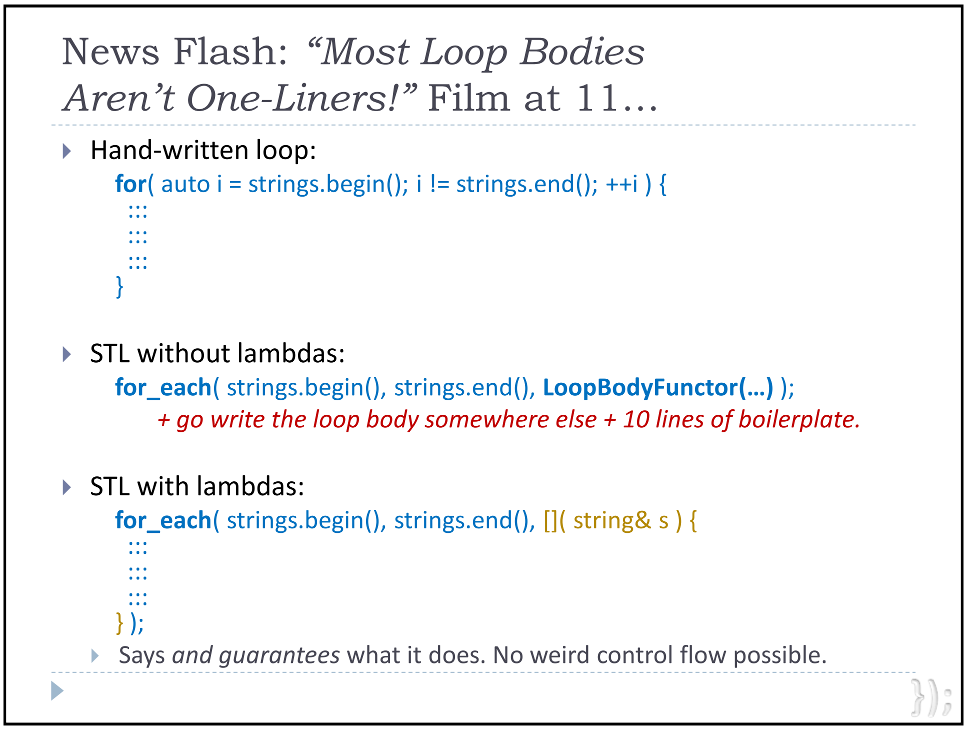
- Hand-written loop:

```
for( auto i = strings.begin(); i != strings.end(); ++i ) {
  :::
  :::
  :::
}
```

- STL without lambdas:

```
for_each( strings.begin(), strings.end(), LoopBodyFunctor(…) );
```

+ go write the loop body somewhere else + 10 lines of boilerplate.



News Flash: *“Most Loop Bodies Aren’t One-Liners!”* Film at 11…

- Hand-written loop:

```
for( auto i = strings.begin(); i != strings.end(); ++i ) {
  :::
  :::
  :::
}
```

- STL without lambdas:

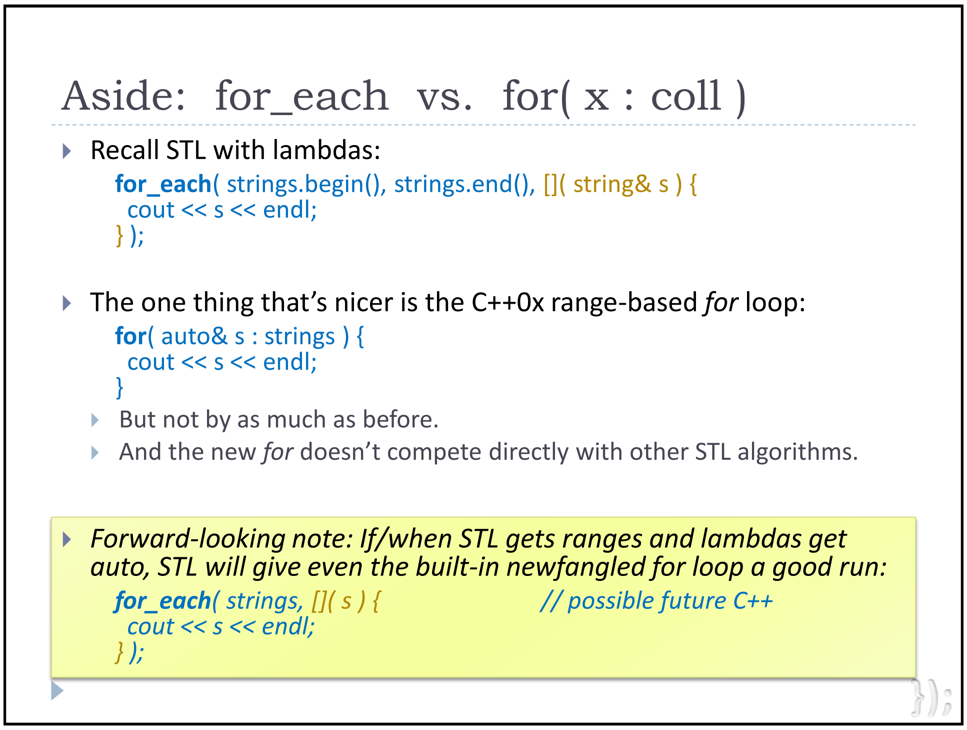
```
for_each( strings.begin(), strings.end(), LoopBodyFunctor(…) );
```

  + go write the loop body somewhere else + 10 lines of boilerplate.

- STL with lambdas:

```
for_each( strings.begin(), strings.end(), []( string& s ) {
  :::
  :::
  :::
} );
```

  - Says *and guarantees* what it does. No weird control flow possible.



Aside: for_each vs. for(x : coll)

- Recall STL with lambdas:

```
for_each( strings.begin(), strings.end(), []( string& s ) {
  cout << s << endl;
} );
```

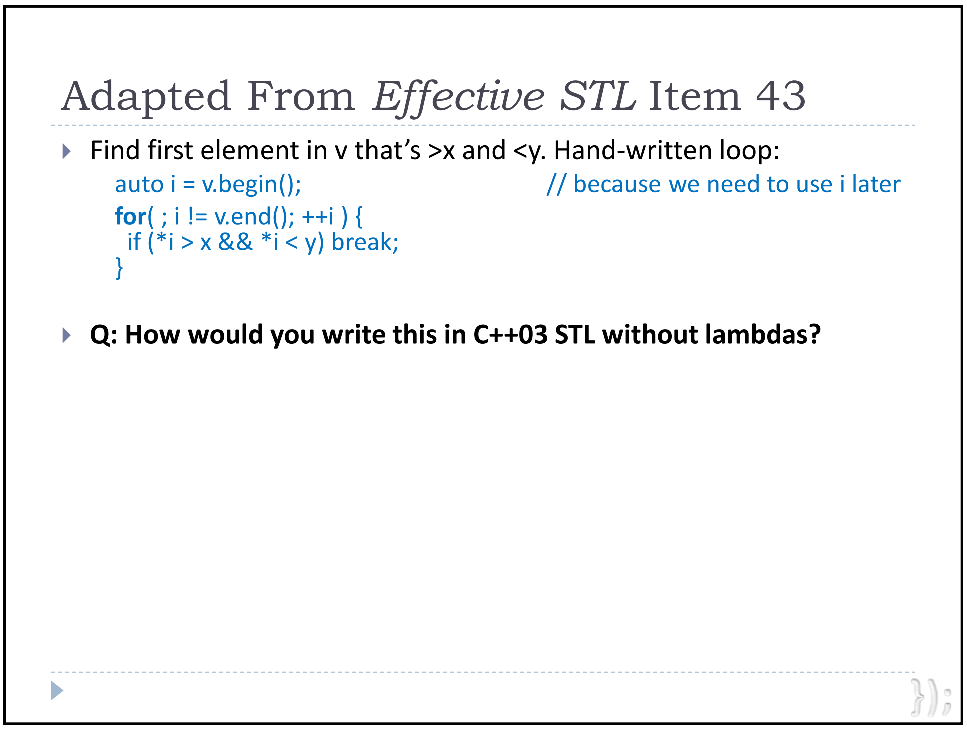
- The one thing that’s nicer is the C++0x range-based *for* loop:

```
for( auto& s : strings ) {
  cout << s << endl;
}
```

  - But not by as much as before.
  - And the new *for* doesn’t compete directly with other STL algorithms.

- *Forward-looking note: If/when STL gets ranges and lambdas get auto, STL will give even the built-in newfangled for loop a good run:*

```
for_each( strings, []( s ) {                // possible future C++
  cout << s << endl;
} );
```

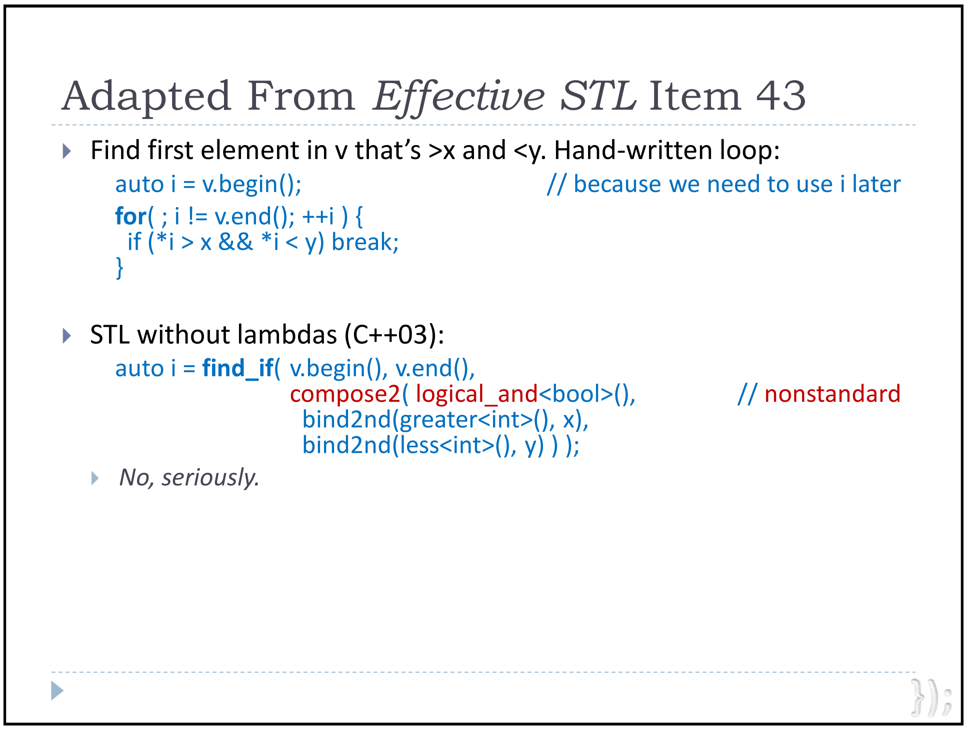


Adapted From *Effective STL* Item 43

- Find first element in v that's >x and <y. Hand-written loop:

```
auto i = v.begin();                    // because we need to use i later
for( ; i != v.end(); ++i ) {
  if (*i > x && *i < y) break;
}
```

- **Q: How would you write this in C++03 STL without lambdas?**



Adapted From *Effective STL* Item 43

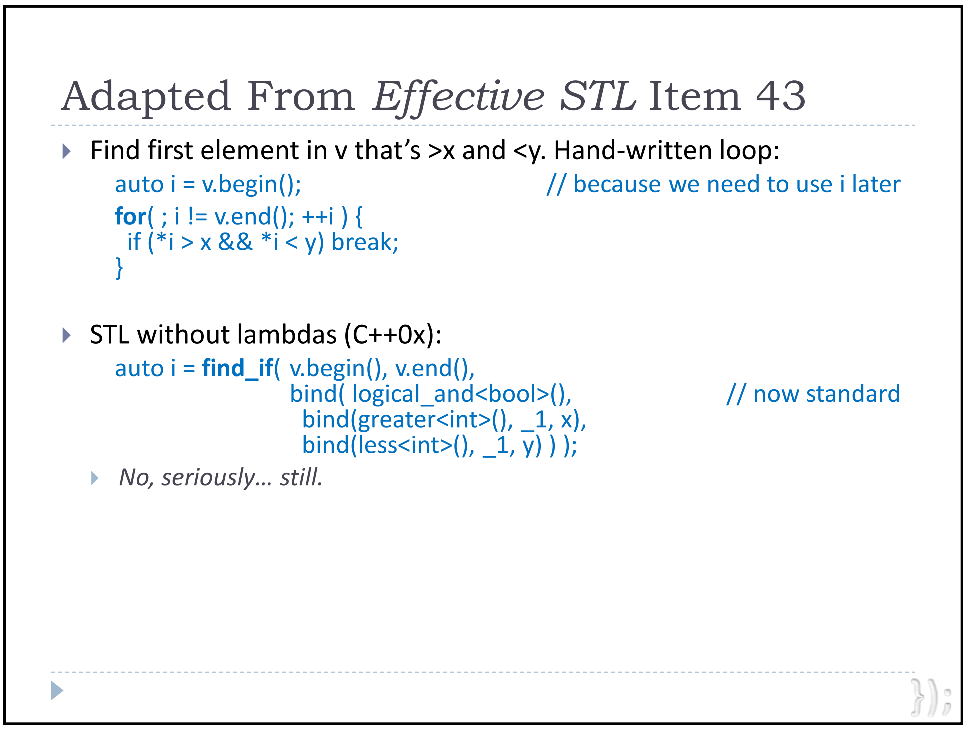
- Find first element in v that's >x and <y. Hand-written loop:

```
auto i = v.begin();                    // because we need to use i later
for( ; i != v.end(); ++i ) {
  if (*i > x && *i < y) break;
}
```

- STL without lambdas (C++03):

```
auto i = find_if( v.begin(), v.end(),
                 compose2( logical_and<bool>(),        // nonstandard
                   bind2nd(greater<int>(), x),
                   bind2nd(less<int>(), y) ) );
```

  - *No, seriously.*



Adapted From *Effective STL* Item 43

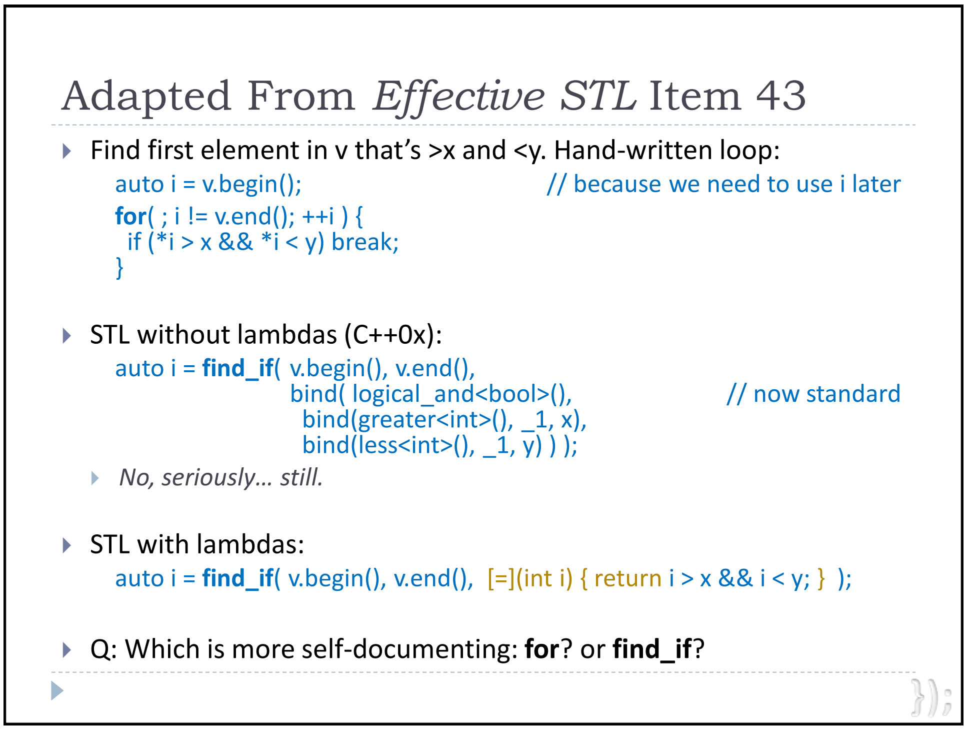
- Find first element in v that's >x and <y. Hand-written loop:

```
auto i = v.begin();                      // because we need to use i later
for( ; i != v.end(); ++i ) {
  if (*i > x && *i < y) break;
}
```

- STL without lambdas (C++0x):

```
auto i = find_if( v.begin(), v.end(),
                  bind( logical_and<bool>(),             // now standard
                   bind(greater<int>(), _1, x),
                   bind(less<int>(), _1, y) ) );
```

  - *No, seriously… still.*



Adapted From *Effective STL* Item 43

- Find first element in v that's >x and <y. Hand-written loop:

```
auto i = v.begin();                      // because we need to use i later
for( ; i != v.end(); ++i ) {
  if (*i > x && *i < y) break;
}
```

- STL without lambdas (C++0x):

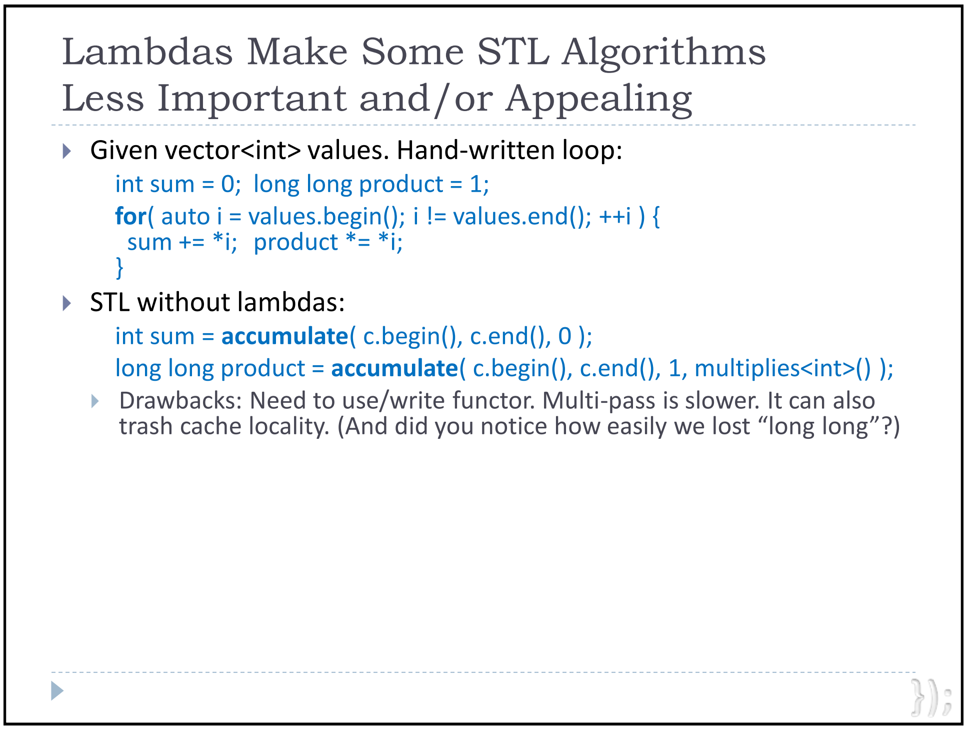
```
auto i = find_if( v.begin(), v.end(),
                  bind( logical_and<bool>(),             // now standard
                   bind(greater<int>(), _1, x),
                   bind(less<int>(), _1, y) ) );
```

  - *No, seriously… still.*

- STL with lambdas:

```
auto i = find_if( v.begin(), v.end(),  [=](int i) { return i > x && i < y; }  );
```

- Q: Which is more self-documenting: **for**? or **find_if**?



Lambdas Make Some STL Algorithms Less Important and/or Appealing

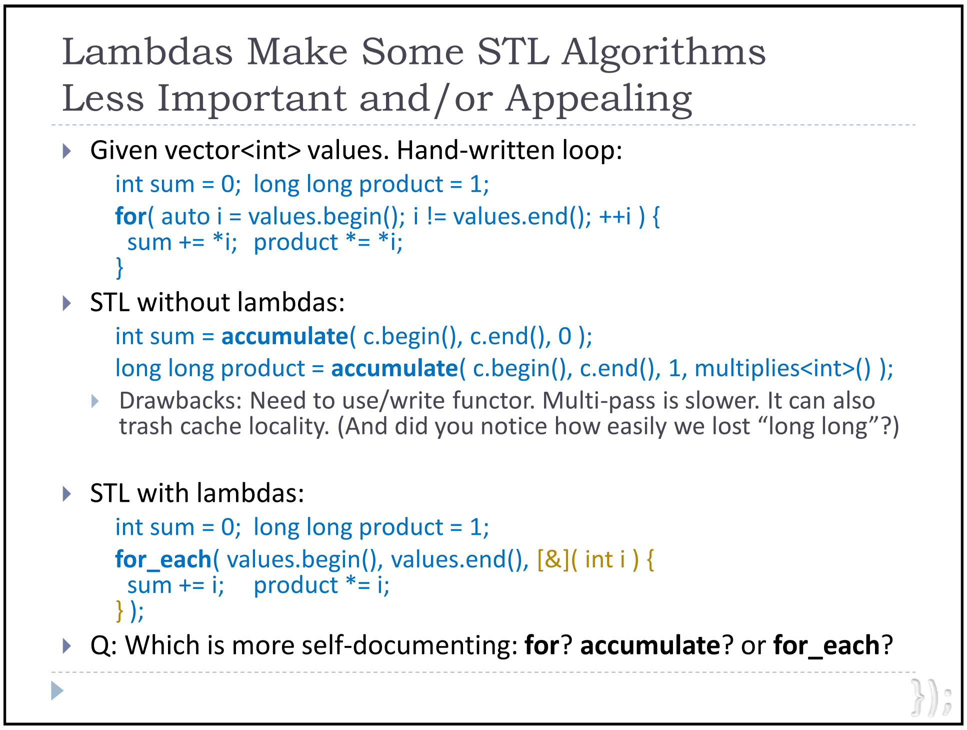
- Given vector<int> values. Hand-written loop:

```
int sum = 0;  long long product = 1;
for( auto i = values.begin(); i != values.end(); ++i ) {
  sum += *i;  product *= *i;
}
```

- STL without lambdas:

```
int sum = accumulate( c.begin(), c.end(), 0 );
long long product = accumulate( c.begin(), c.end(), 1, multiplies<int>() );
```

  - Drawbacks: Need to use/write functor. Multi-pass is slower. It can also trash cache locality. (And did you notice how easily we lost "long long"?)



Lambdas Make Some STL Algorithms Less Important and/or Appealing

- Given vector<int> values. Hand-written loop:

```
int sum = 0;  long long product = 1;
for( auto i = values.begin(); i != values.end(); ++i ) {
  sum += *i;  product *= *i;
}
```

- STL without lambdas:

```
int sum = accumulate( c.begin(), c.end(), 0 );
long long product = accumulate( c.begin(), c.end(), 1, multiplies<int>() );
```

  - Drawbacks: Need to use/write functor. Multi-pass is slower. It can also trash cache locality. (And did you notice how easily we lost "long long"?)

- STL with lambdas:

```
int sum = 0;  long long product = 1;
for_each( values.begin(), values.end(), [&]( int i ) {
  sum += i;     product *= i;
} );
```

- Q: Which is more self-documenting: **for**? **accumulate**? or **for_each**?

Performance Footnote

- Given:

```
char str[] =
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

- Baseline: Raw pointer loop + near-empty payload (built-in int ++).

```
char* end = &str[52];
for( char* p = str; p != end; ++p )
  ++*p;
```

}};

Performance Footnote

- Given:

```
char str[] =
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

- Baseline: Raw pointer loop + near-empty payload (built-in int ++).

```
char* end = &str[52];
for( char* p = str; p != end; ++p )
  ++*p;
```

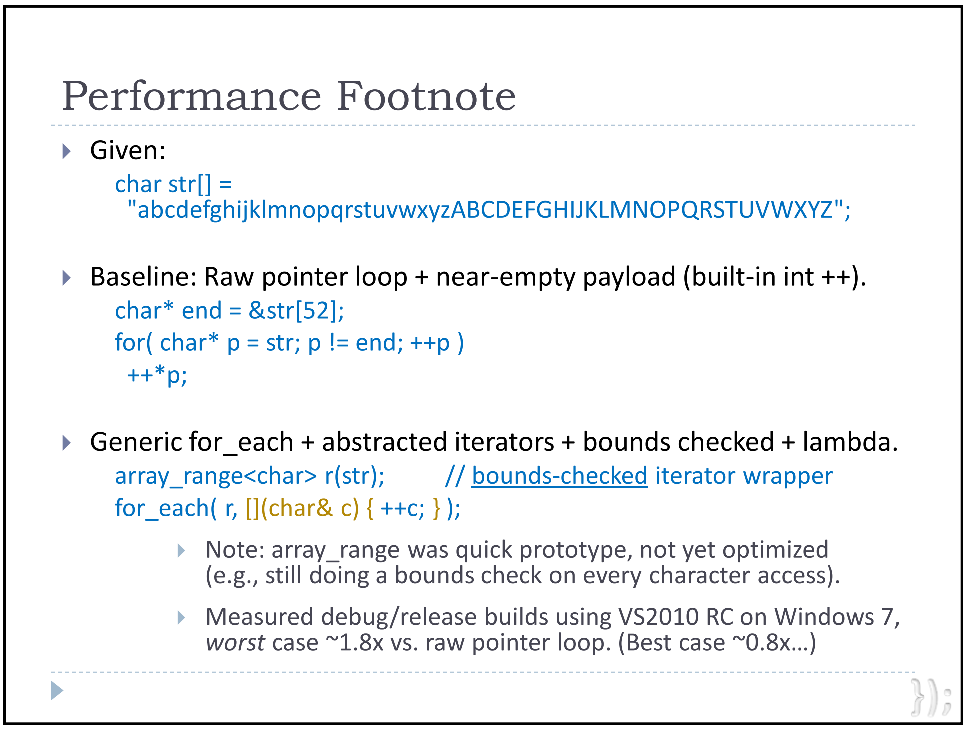
- Generic for_each + abstracted iterators + bounds checked + lambda.

```
array_range<char> r(str);        // bounds-checked iterator wrapper
for_each( r, [](char& c) { ++c; } );
```

  - Note: array_range was quick prototype, not yet optimized (e.g., still doing a bounds check on every character access).

    Measured debug/release builds using VS2010 RC on Windows 7, **best/worst case…?)**

}};



Performance Footnote

- Given:

```
char str[] =
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

- Baseline: Raw pointer loop + near-empty payload (built-in int ++).

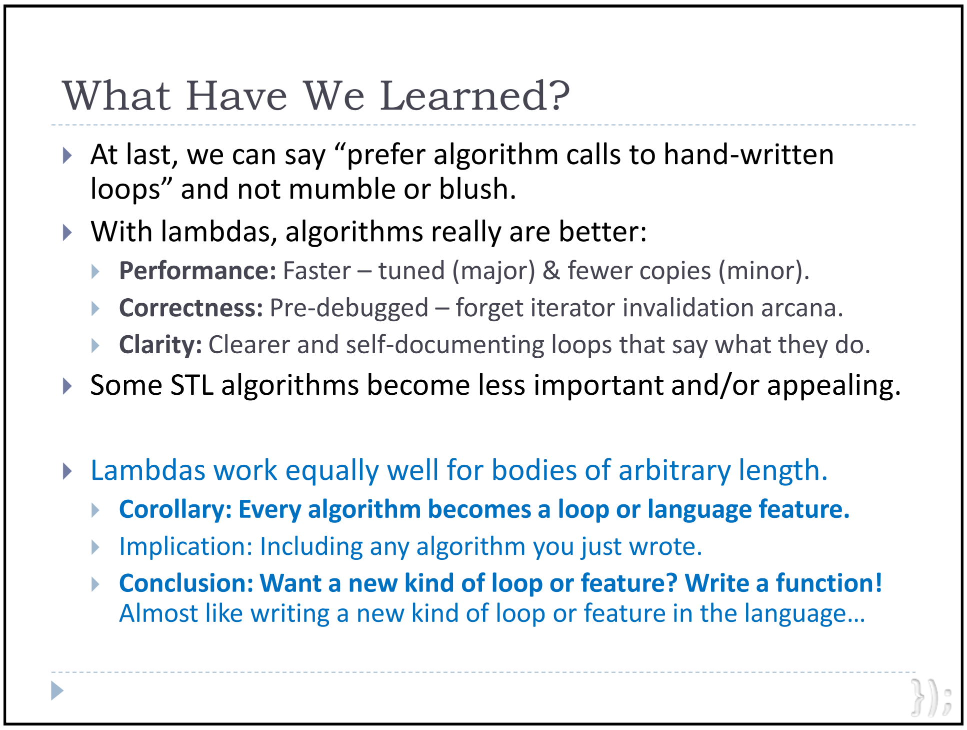
```
char* end = &str[52];
for( char* p = str; p != end; ++p )
  ++*p;
```

- Generic for_each + abstracted iterators + bounds checked + lambda.

```
array_range<char> r(str);        // bounds-checked iterator wrapper
for_each( r, [](char& c) { ++c; } );
```

  - Note: array_range was quick prototype, not yet optimized (e.g., still doing a bounds check on every character access).
  - Measured debug/release builds using VS2010 RC on Windows 7, *worst* case ~1.8x vs. raw pointer loop. (Best case ~0.8x…)

});

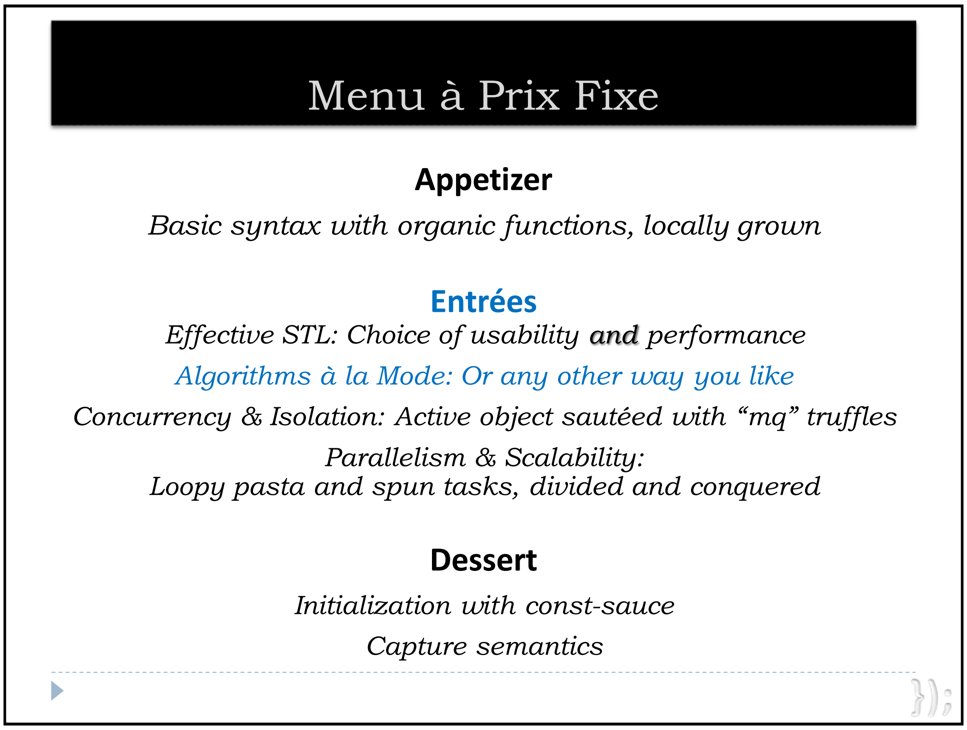


What Have We Learned?

- At last, we can say "prefer algorithm calls to hand-written loops" and not mumble or blush.
- With lambdas, algorithms really are better:
  - **Performance:** Faster – tuned (major) & fewer copies (minor).
  - **Correctness:** Pre-debugged – forget iterator invalidation arcana.
  - **Clarity:** Clearer and self-documenting loops that say what they do.
- Some STL algorithms become less important and/or appealing.

- Lambdas work equally well for bodies of arbitrary length.
  - **Corollary: Every algorithm becomes a loop or language feature.**
  - Implication: Including any algorithm you just wrote.
  - **Conclusion: Want a new kind of loop or feature? Write a function!** Almost like writing a new kind of loop or feature in the language…

});



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

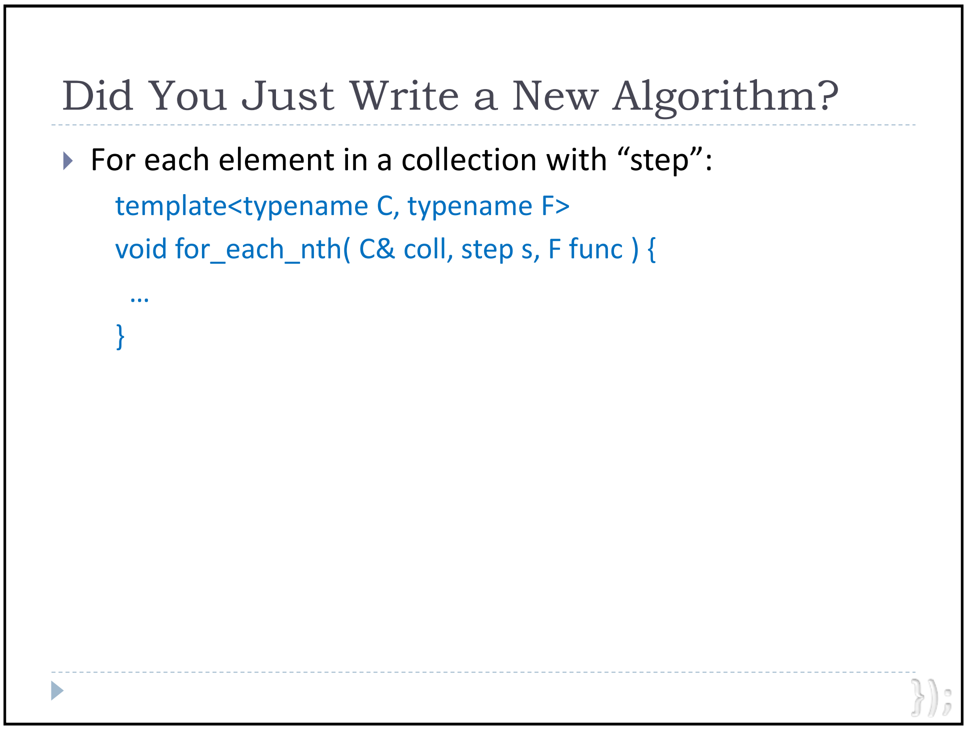
Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

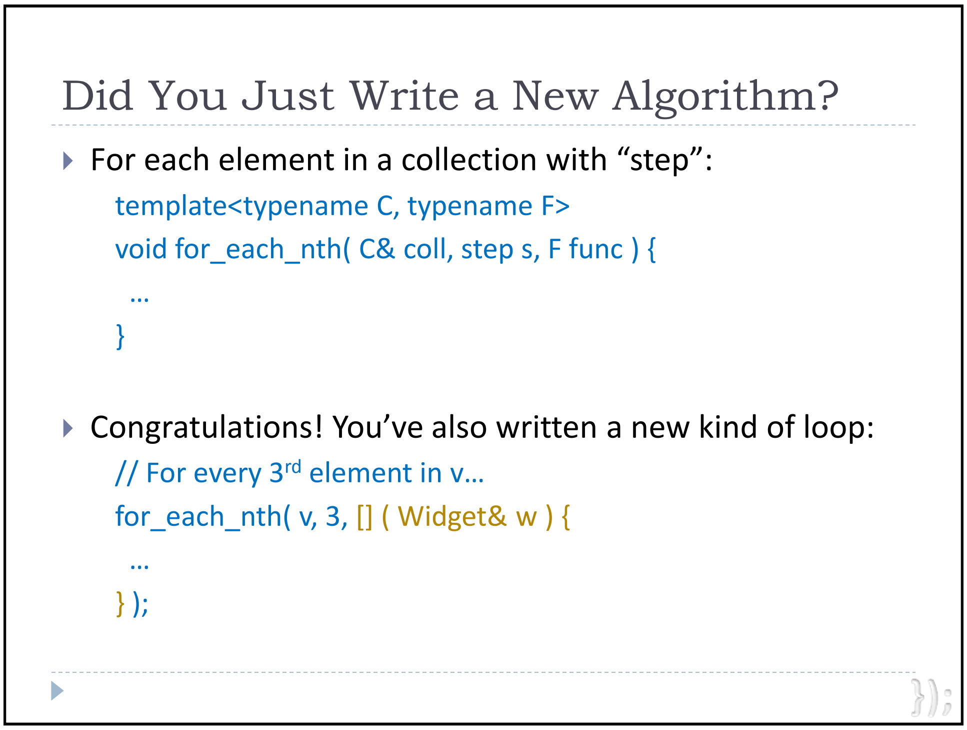


Did You Just Write a New Algorithm?

- For each element in a collection with "step":

```
template<typename C, typename F>
void for_each_nth( C& coll, step s, F func ) {
  …
}
```

Herb Sutter
Software Development Consultant
www.gotw.ca/training



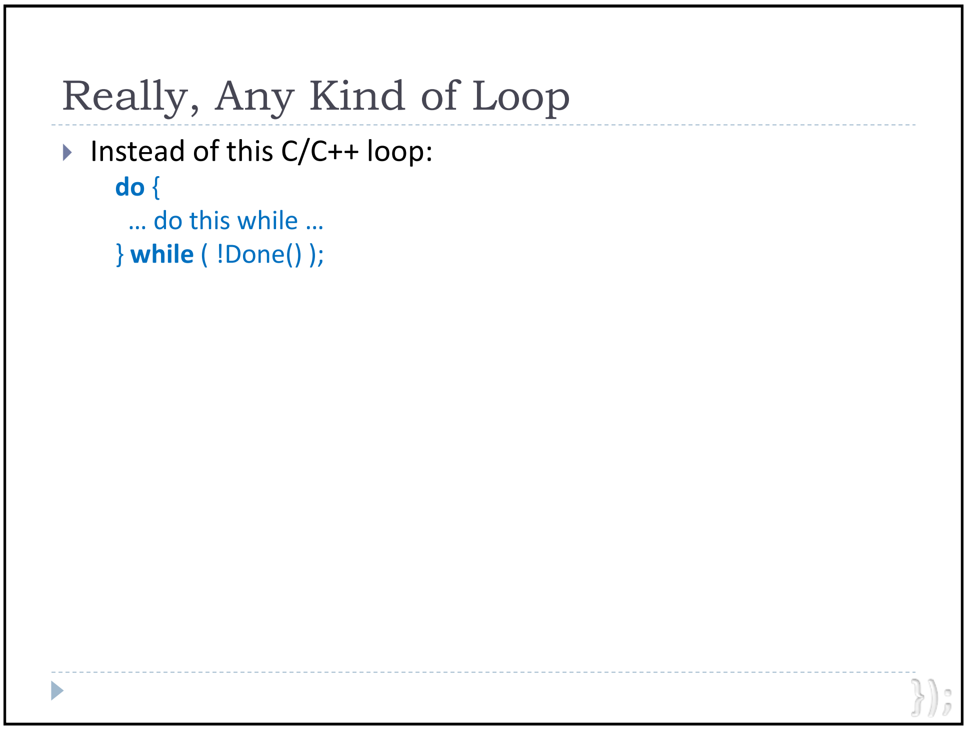
Did You Just Write a New Algorithm?

- For each element in a collection with “step”:

```
template<typename C, typename F>
void for_each_nth( C& coll, step s, F func ) {
  …
}
```

- Congratulations! You’ve also written a new kind of loop:

```
// For every 3rd element in v…
for_each_nth( v, 3, [] ( Widget& w ) {
  …
} );
```



Really, Any Kind of Loop

- Instead of this C/C++ loop:

```
do {
  … do this while …
} while ( !Done() );
```

Herb Sutter
Software Development Consultant
www.gotw.ca/training

Really, Any Kind of Loop

- Instead of this C/C++ loop:

```
do {
  … do this while …
} while ( !Done() );
```

- It's just about as easy to write this:

```
do_while( [&]{
  … do this while …
} , []{ return !Done(); } );
```

- Or if you like Pascal syntax, call it repeat_until and drop "!":

```
repeat_until( [&]{
  … do this until …
} , []{ return Done(); } );
```

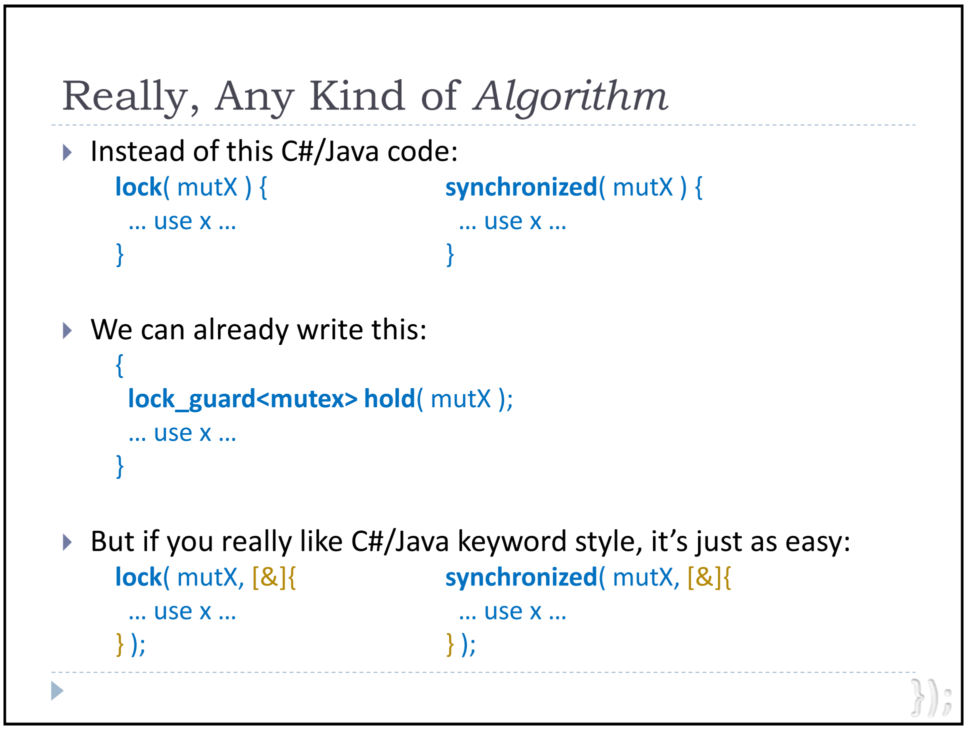
Really, Any Kind of *Algorithm*

- Instead of this C#/Java code:

```
lock( mutX ) {                  synchronized( mutX ) {
  … use x …                       … use x …
}                               }
```

- We can already write this:

```
{
  lock_guard<mutex> hold( mutX );
  … use x …
}
```



Really, Any Kind of *Algorithm*

- Instead of this C#/Java code:

```
lock( mutX ) {                    synchronized( mutX ) {
  … use x …                         … use x …
}                                 }
```

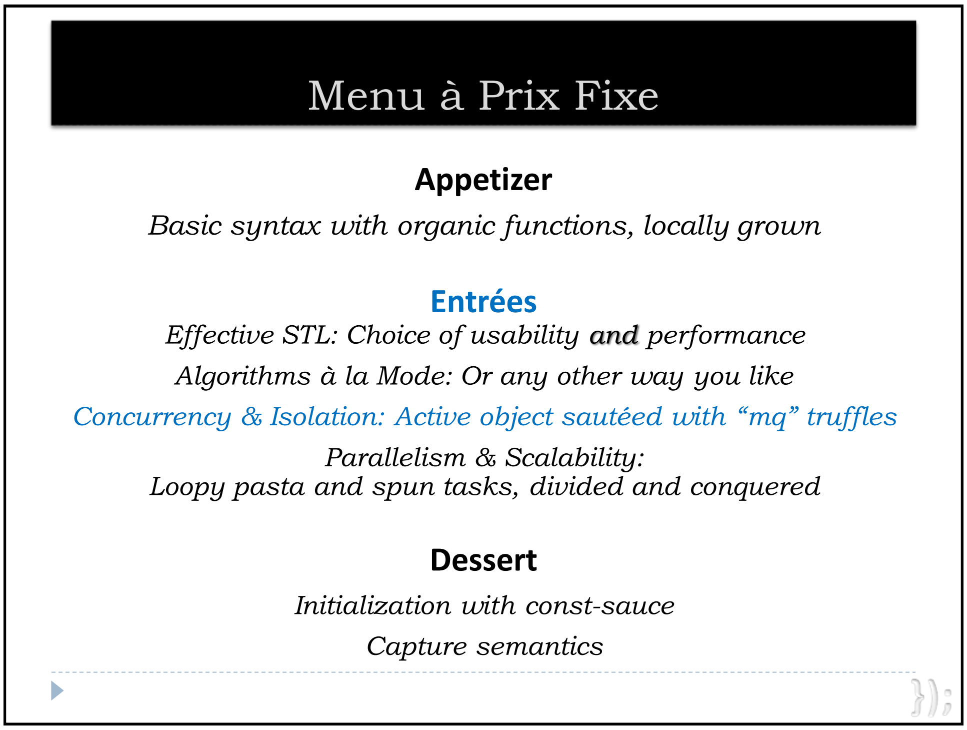
- We can already write this:

```
{
  lock_guard<mutex> hold( mutX );
  … use x …
}
```

- But if you really like C#/Java keyword style, it's just as easy:

```
lock( mutX, [&]{                  synchronized( mutX, [&]{
  … use x …                         … use x …
} );                              } );
```

});



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

});

Active Objects: What We're Aiming For

- Summary: An active object runs on its own thread.
  - Therefore each active object is an asynchronous agent.
- Method calls become async messages.
  - Method syntax is familiar and understandable, and lets generic code treat active and ordinary objects uniformly.
  - The active object's thread mainline is a message pump. Processing messages sequentially makes them atomic with respect to each other, avoiding the need to do internal or external locking/synchronization.
- Example:

```
class Active {          // (has to be coded up specially)
public:
  void Func() { … }
};

// in calling code, using an Active object
Active a;
a.Func();               // call is nonblocking
… more work …           // this code can execute in parallel with a.Func()
```

Example: Background Worker Agent

- Say we want to have an agent that does background work:

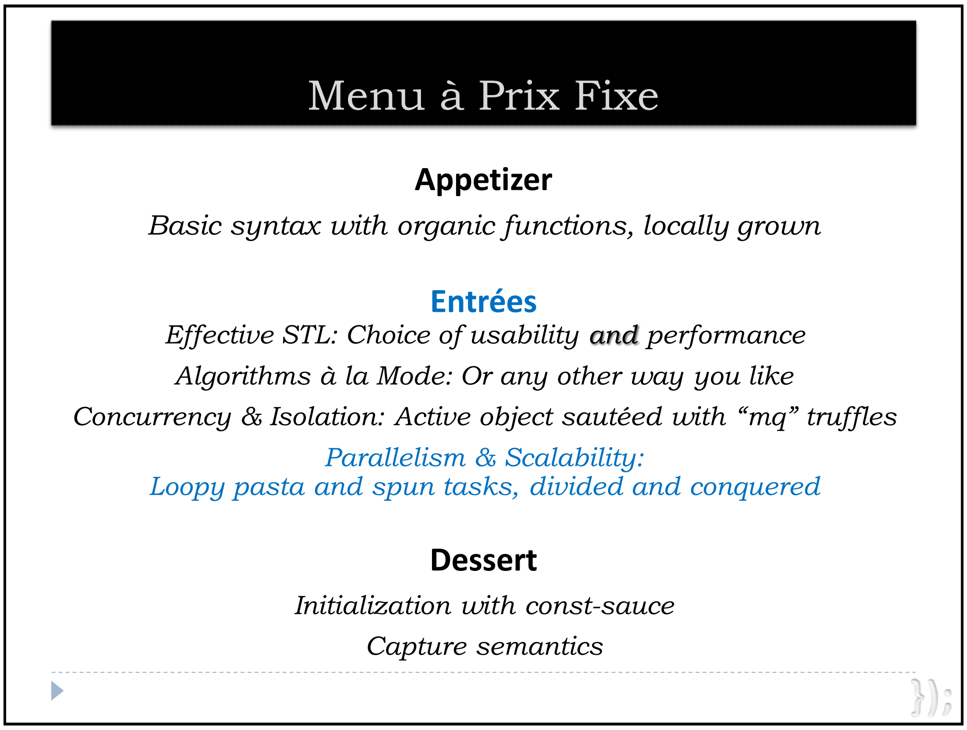
```
class Backgrounder {
public:
  void Save( string filename ) {
    a.Send( bind( &Backgrounder::DoSave, this, filename ) );
  }

  void Print( Data& data ) {
    a.Send( bind( &Backgrounder::DoPrint, this, ref(data) ) );
  }                                          // note: important ref()!

private:
  …          // thread-private data
  Active a;  // helper that encapsulates thread + msg queue/loop

  void DoSave( string filename ) { … }

  void DoPrint( Data& data ) { … }
};
```

Same Example, With Lambdas

```
class Backgrounder {
public:
  void Save( string filename ) { a.Send( [=] {
    …
  } ); }
  void Print( Data& data ) { a.Send( [=, &data] {
    …
  } ); }
private:
  …          // thread-private data
  Active a;  // helper that encapsulates thread + msg queue/loop
};
```

What Have We Learned?

- Lambdas make the active object pattern simple:
  - Writing a robust active class is almost as simple and natural as writing an ordinary class.
  - Just add a helper member, and write "a.Send([=] {" (or similar) at the beginning of each member function.
  - Not bad for not having active objects built into the language.



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

Concurrency & Isolation: Active object sautéed with "mq" truffles

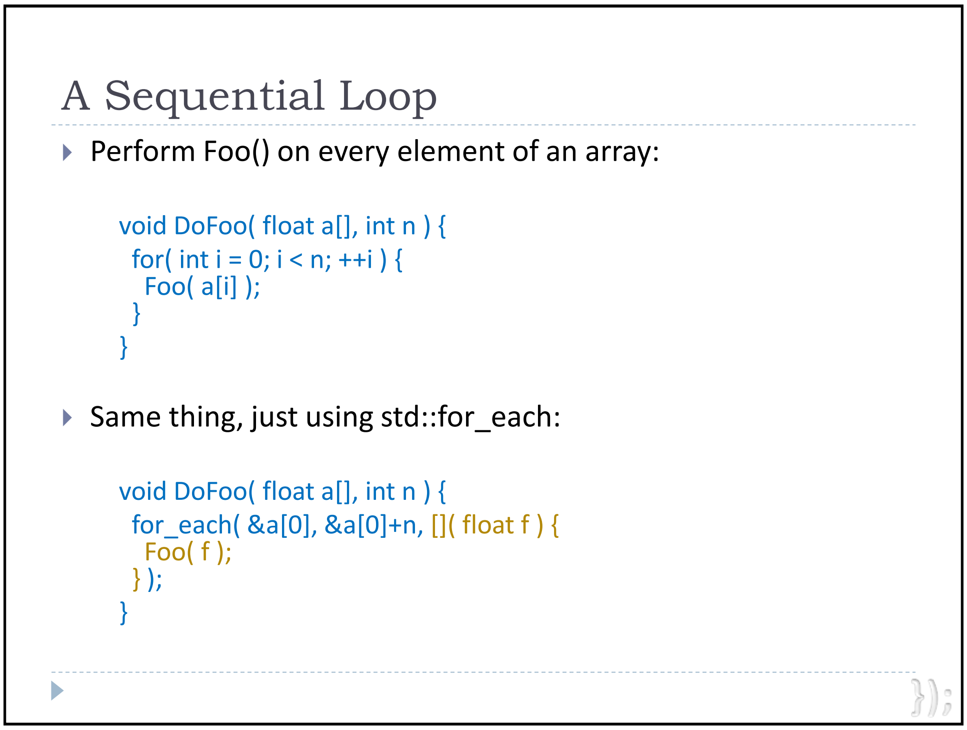
Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

});



A Sequential Loop

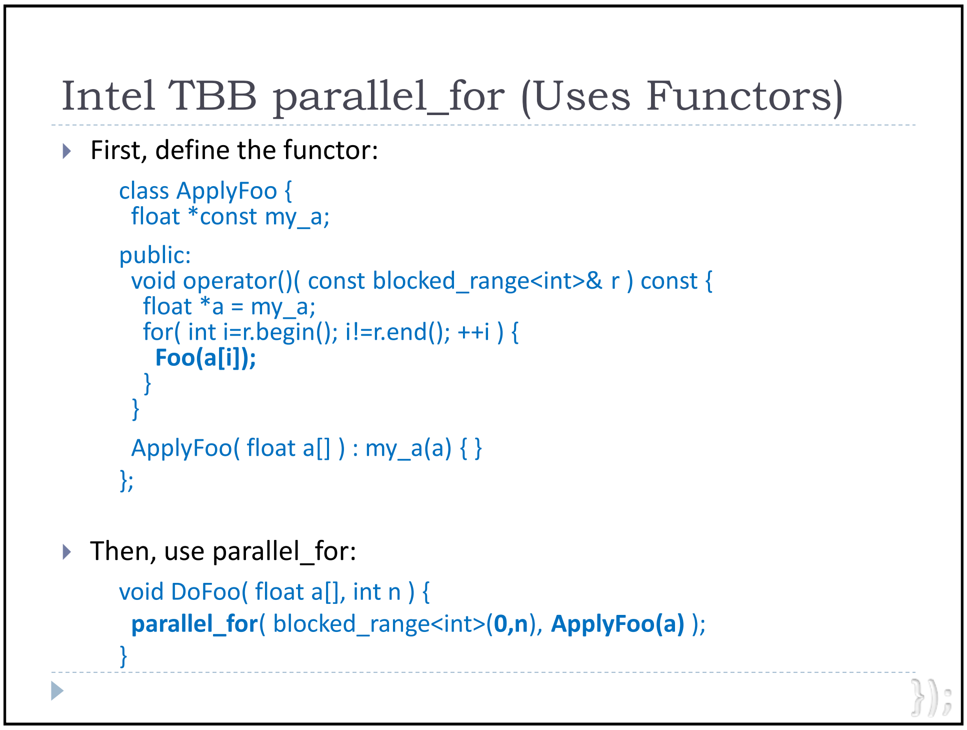
- Perform Foo() on every element of an array:

```
void DoFoo( float a[], int n ) {
  for( int i = 0; i < n; ++i ) {
    Foo( a[i] );
  }
}
```

- Same thing, just using std::for_each:

```
void DoFoo( float a[], int n ) {
  for_each( &a[0], &a[0]+n, []( float f ) {
    Foo( f );
  } );
}
```

});



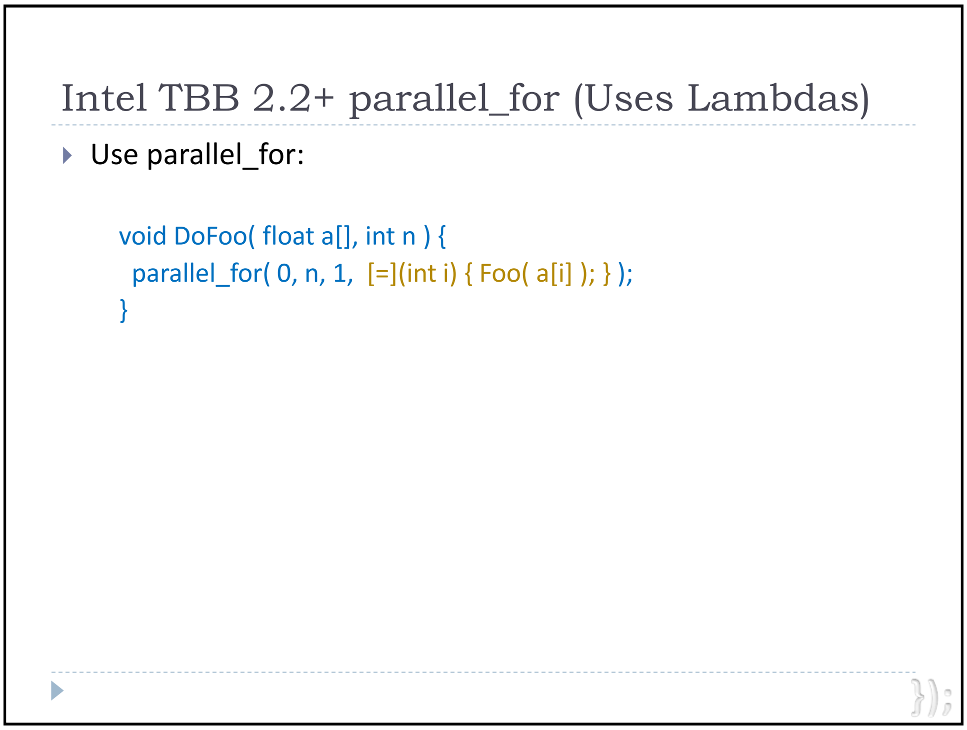
Intel TBB parallel_for (Uses Functors)

- First, define the functor:

```
class ApplyFoo {
  float *const my_a;

public:
  void operator()( const blocked_range<int>& r ) const {
    float *a = my_a;
    for( int i=r.begin(); i!=r.end(); ++i ) {
      Foo(a[i]);
    }
  }

  ApplyFoo( float a[] ) : my_a(a) { }
};
```

- Then, use parallel_for:

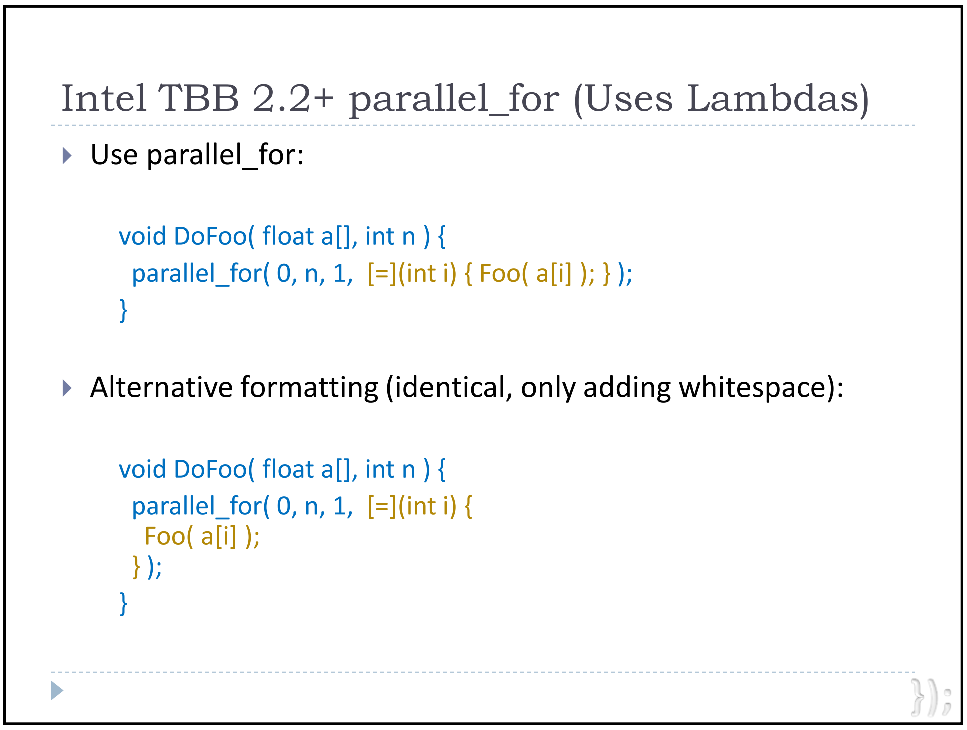
```
void DoFoo( float a[], int n ) {
  parallel_for( blocked_range<int>(0,n), ApplyFoo(a) );
}
```



Intel TBB 2.2+ parallel_for (Uses Lambdas)

- Use parallel_for:

```
void DoFoo( float a[], int n ) {
  parallel_for( 0, n, 1,  [=](int i) { Foo( a[i] ); } );
}
```



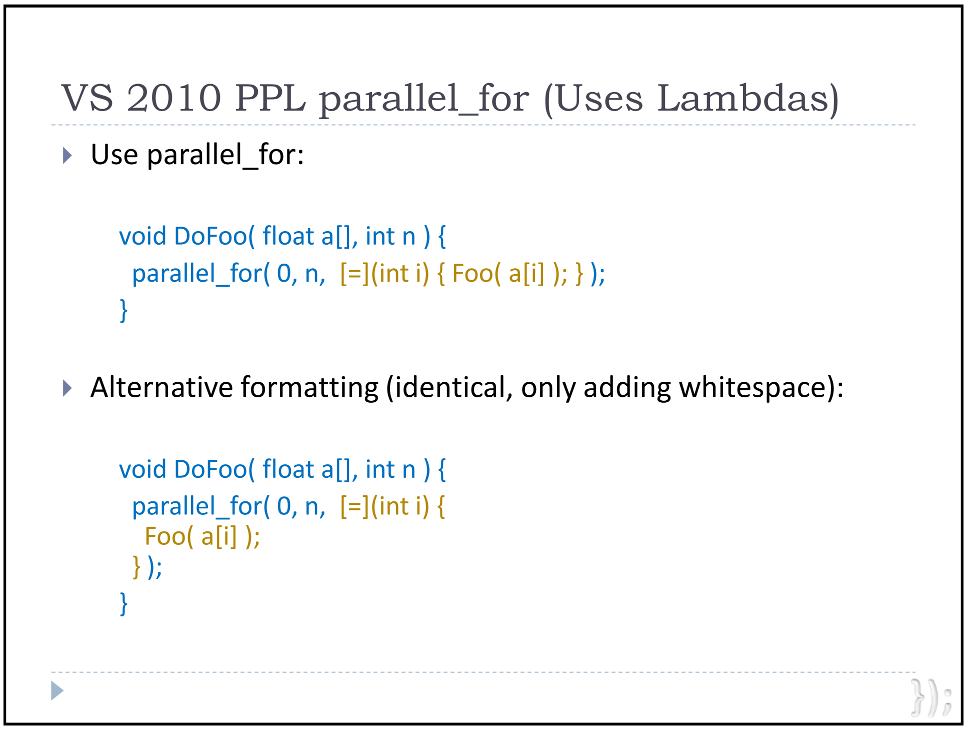
Intel TBB 2.2+ parallel_for (Uses Lambdas)

- Use parallel_for:

```
void DoFoo( float a[], int n ) {
  parallel_for( 0, n, 1,  [=](int i) { Foo( a[i] ); } );
}
```

- Alternative formatting (identical, only adding whitespace):

```
void DoFoo( float a[], int n ) {
  parallel_for( 0, n, 1,  [=](int i) {
    Foo( a[i] );
  } );
}
```



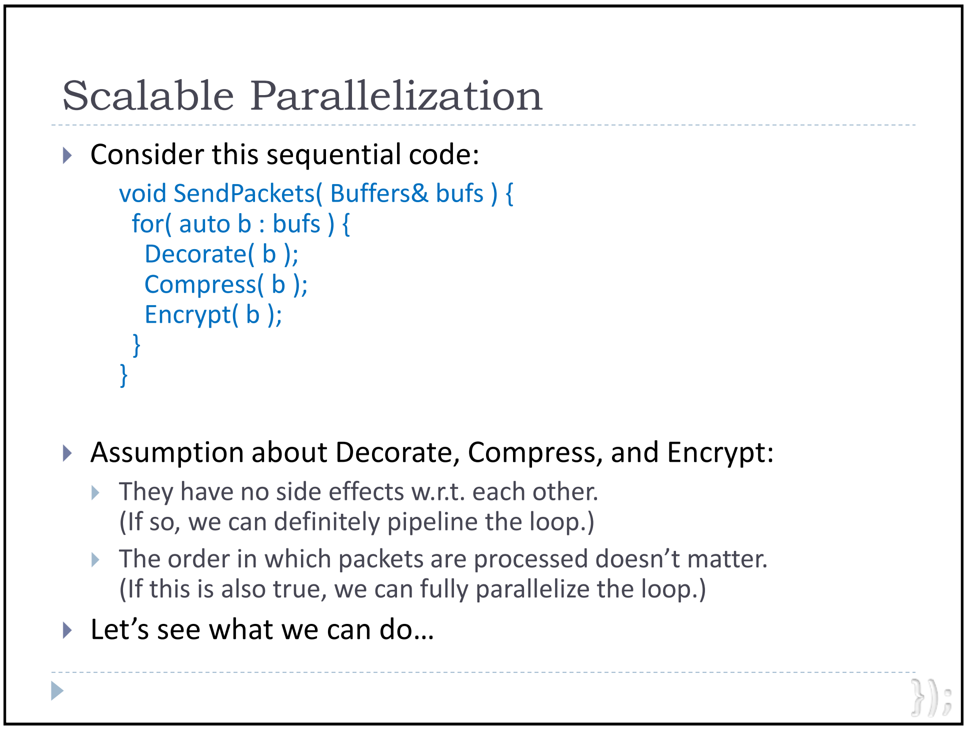
VS 2010 PPL parallel_for (Uses Lambdas)

- Use parallel_for:

```
void DoFoo( float a[], int n ) {
  parallel_for( 0, n,  [=](int i) { Foo( a[i] ); } );
}
```

- Alternative formatting (identical, only adding whitespace):

```
void DoFoo( float a[], int n ) {
  parallel_for( 0, n,  [=](int i) {
    Foo( a[i] );
  } );
}
```

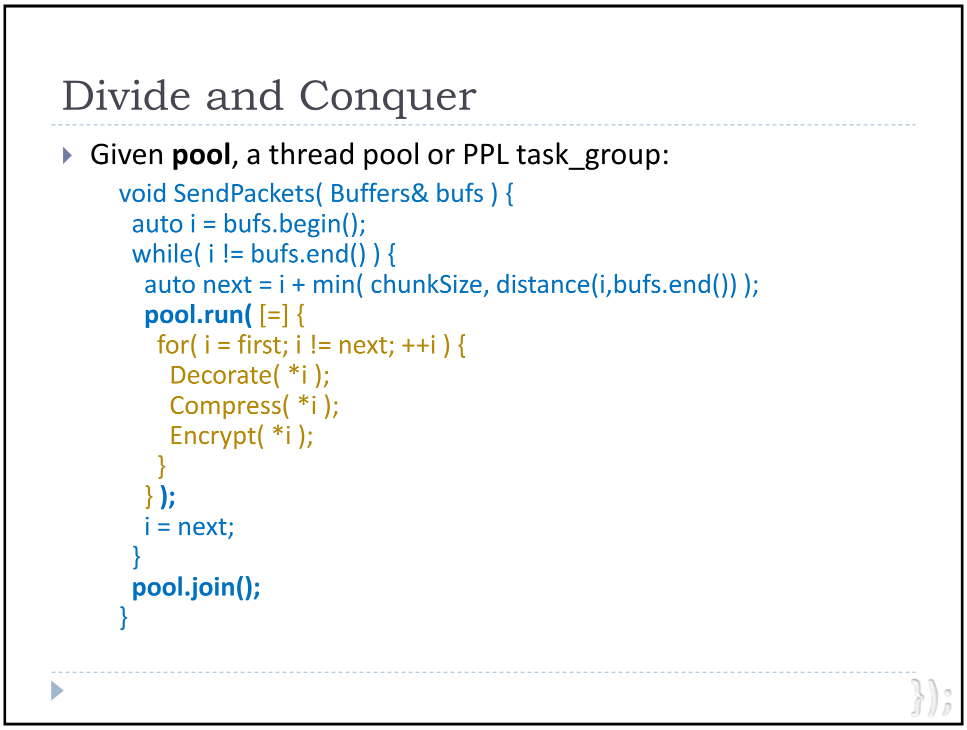


Scalable Parallelization

- Consider this sequential code:

```
void SendPackets( Buffers& bufs ) {
  for( auto b : bufs ) {
    Decorate( b );
    Compress( b );
    Encrypt( b );
  }
}
```

- Assumption about Decorate, Compress, and Encrypt:
  - They have no side effects w.r.t. each other.
    (If so, we can definitely pipeline the loop.)
  - The order in which packets are processed doesn't matter.
    (If this is also true, we can fully parallelize the loop.)
- Let's see what we can do…



Divide and Conquer

- Given **pool**, a thread pool or PPL task_group:

```
void SendPackets( Buffers& bufs ) {
  auto i = bufs.begin();
  while( i != bufs.end() ) {
    auto next = i + min( chunkSize, distance(i,bufs.end()) );
    pool.run( [=] {
      for( i = first; i != next; ++i ) {
        Decorate( *i );
        Compress( *i );
        Encrypt( *i );
      }
    } );
    i = next;
  }
  pool.join();
}
```

What Have We Learned?

- Lambdas make the parallel loops, parallel decomposition, fork/join, etc. all simple:
  - Just because the body of our algorithm will run in parallel, we don't need to lose the locality of seeing our algorithm all in one place.
  - Plus the code remains structured, which is important.

});

Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics

});

Initialization

- Example question from "tohava" (*comp.lang.c++.moderated*, March 2010):

```
int x = 1;                        // should be const, but:
for( int i = 2; i <= N; ++i ) {   // this could be some
  x += i;                         // arbitrarily long code
}                                 // needed to initialize x
// from here, x should be const, but we can't say so in code
```

- Q: How can we make x be const?

Attempt #1: OK

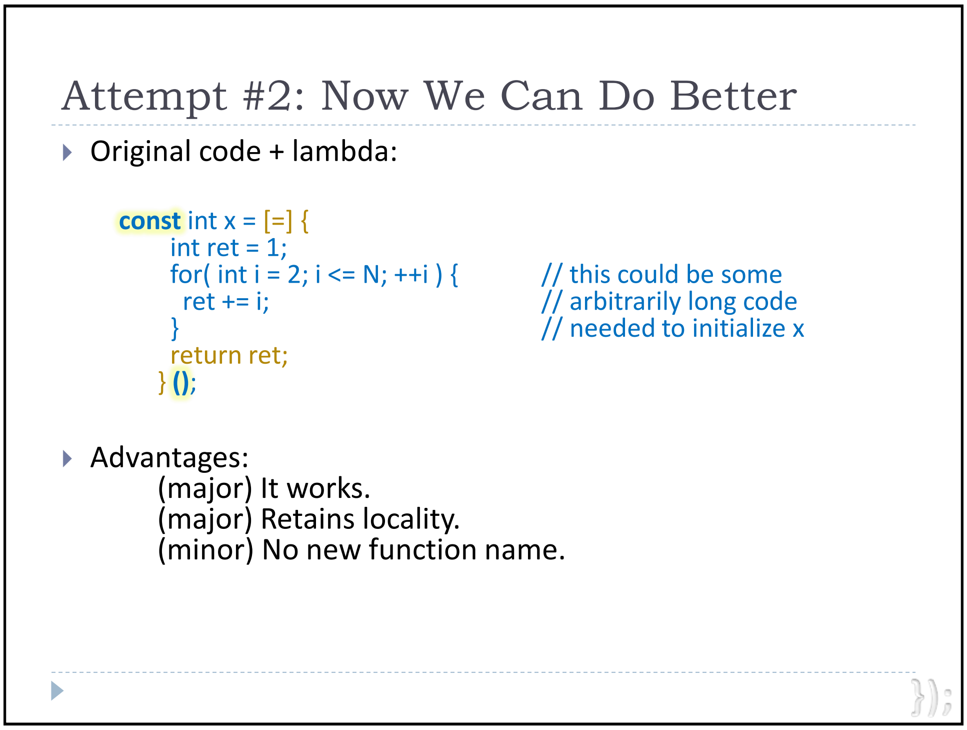
- Make the initialization a separate function:

```
int XInit( int N ) {
  int ret = 1;
  for( int i = 2; i <= N; ++i ) {   // this could be some
    ret += i;                       // arbitrarily long code
  }                                 // needed to initialize x
  return ret;
}
```

- Later on:

```
… elsewhere, elsewhen …
const int x = XInit( N );           // success
```

- Advantage: It works.
- Drawbacks:
  - (major) Loses locality.
  - (minor) Burns a function name.

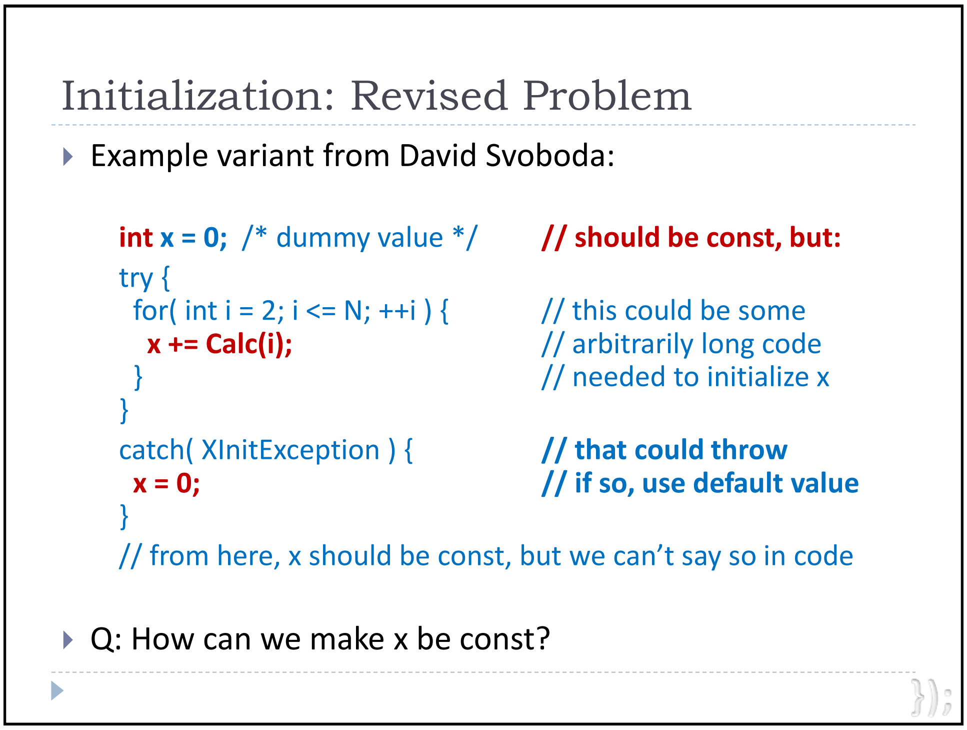


Attempt #2: Now We Can Do Better

- Original code + lambda:

```
const int x = [=] {
        int ret = 1;
        for( int i = 2; i <= N; ++i ) {          // this could be some
          ret += i;                              // arbitrarily long code
        }                                        // needed to initialize x
        return ret;
      } ();
```

- Advantages:
  (major) It works.
  (major) Retains locality.
  (minor) No new function name.

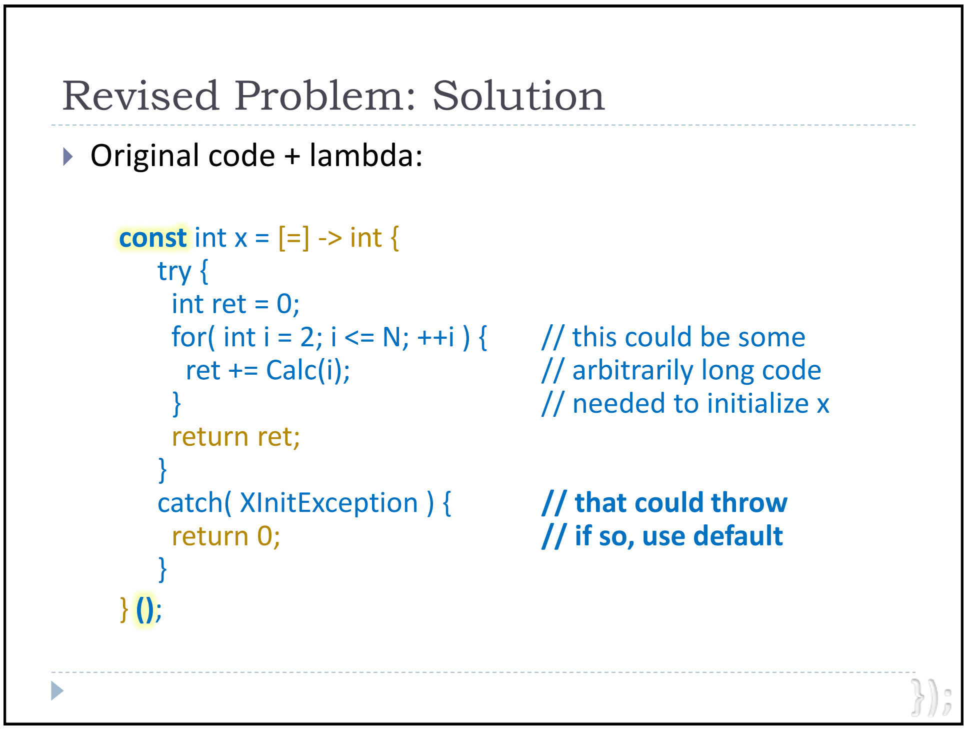


Initialization: Revised Problem

- Example variant from David Svoboda:

```
int x = 0;  /* dummy value */          // should be const, but:
try {
  for( int i = 2; i <= N; ++i ) {      // this could be some
    x += Calc(i);                      // arbitrarily long code
  }                                    // needed to initialize x
}
catch( XInitException ) {             // that could throw
  x = 0;                               // if so, use default value
}
// from here, x should be const, but we can't say so in code
```

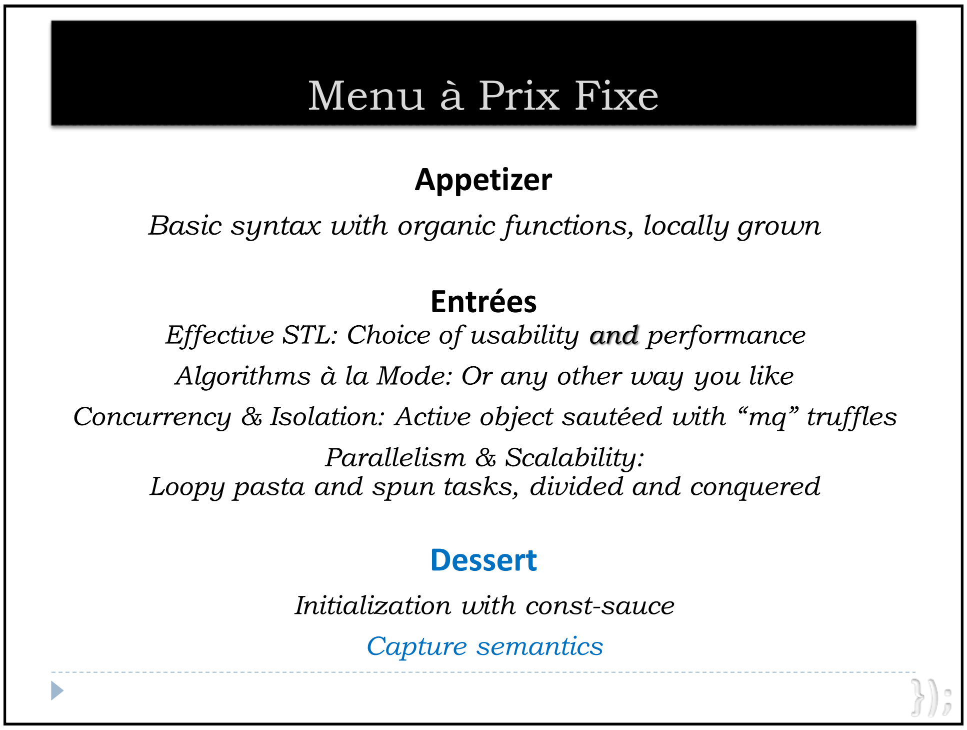
- Q: How can we make x be const?



Revised Problem: Solution

- Original code + lambda:

```
const int x = [=] -> int {
    try {
      int ret = 0;
      for( int i = 2; i <= N; ++i ) {        // this could be some
        ret += Calc(i);                      // arbitrarily long code
      }                                      // needed to initialize x
      return ret;
    }
    catch( XInitException ) {                // that could throw
      return 0;                              // if so, use default
    }
} ();
```



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

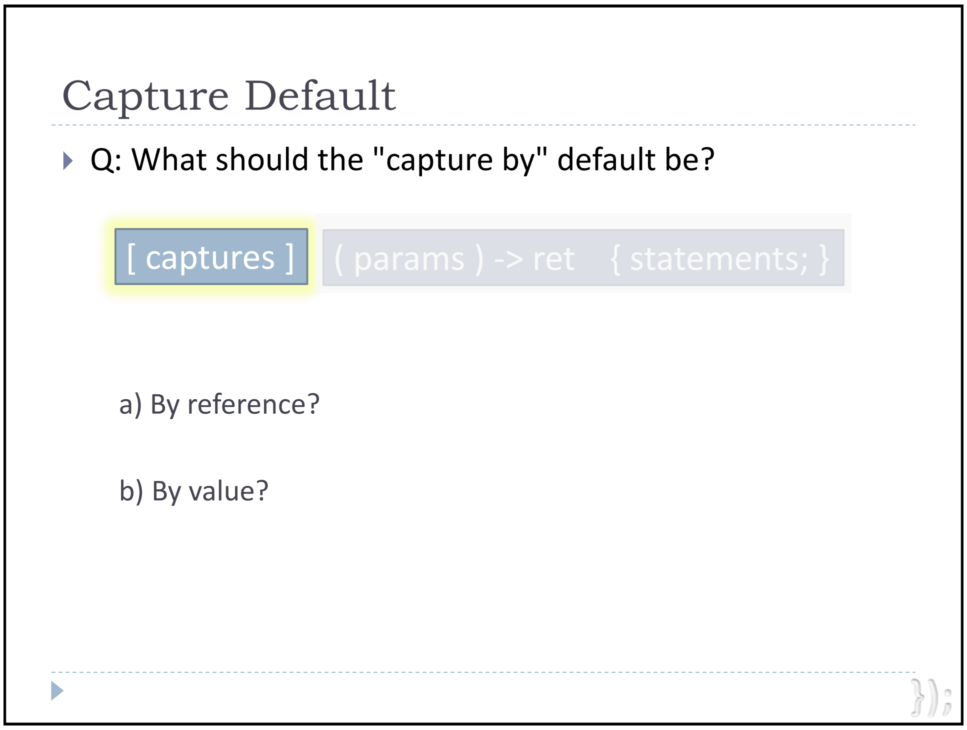
Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

Initialization with const-sauce

Capture semantics



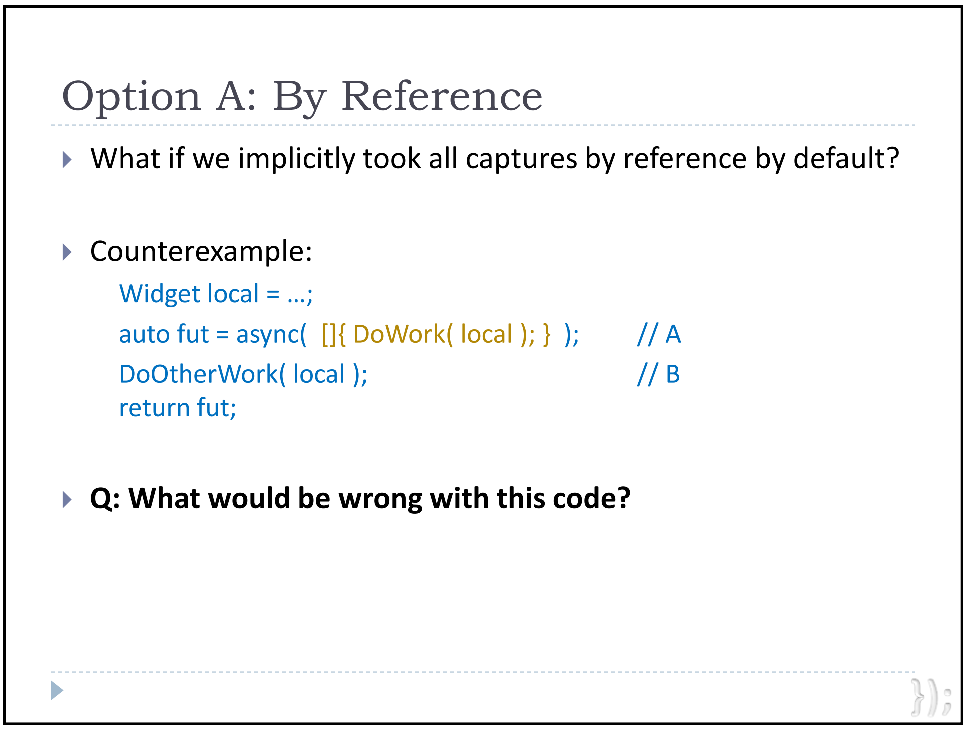
Capture Default

- Q: What should the "capture by" default be?

[captures] (params) -> ret { statements; }

a) By reference?

b) By value?



Option A: By Reference

- What if we implicitly took all captures by reference by default?
- Counterexample:

```
Widget local = …;
auto fut = async( []{ DoWork( local ); } );        // A
DoOtherWork( local );                              // B
return fut;
```

- **Q: What would be wrong with this code?**

Option A: By Reference

- What if we implicitly took all captures by reference by default?
- Counterexample:

```
Widget local = …;
auto fut = async(  []{ DoWork( local ); }  );        // A
DoOtherWork( local );                                // B
return fut;
```

- **Q: What would be wrong with this code?**
  - **A1: Aliasing.**
    Unintended aliasing causes race on *local* on lines A and B.
  - **A2: Lifetime.**
    The lambda might execute after *local* is destroyed. (Oops.)

});

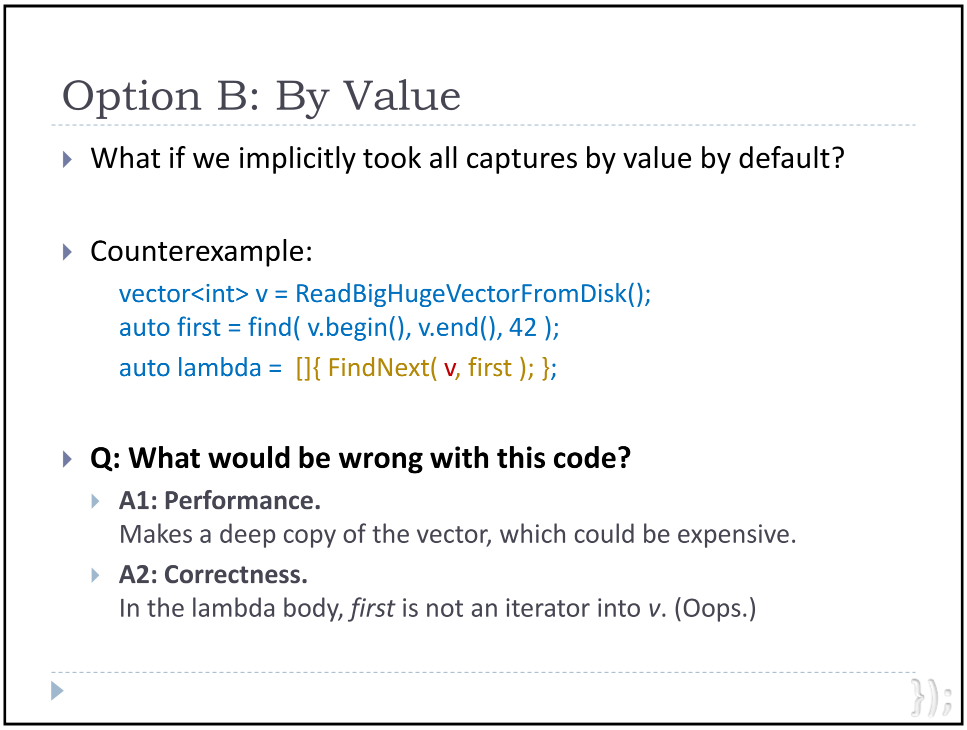
Option B: By Value

- What if we implicitly took all captures by value by default?
- Counterexample:

```
vector<int> v = ReadBigHugeVectorFromDisk();
auto first = find( v.begin(), v.end(), 42 );
auto lambda =  []{ FindNext( v, first ); };
```

- **Q: What would be wrong with this code?**

});

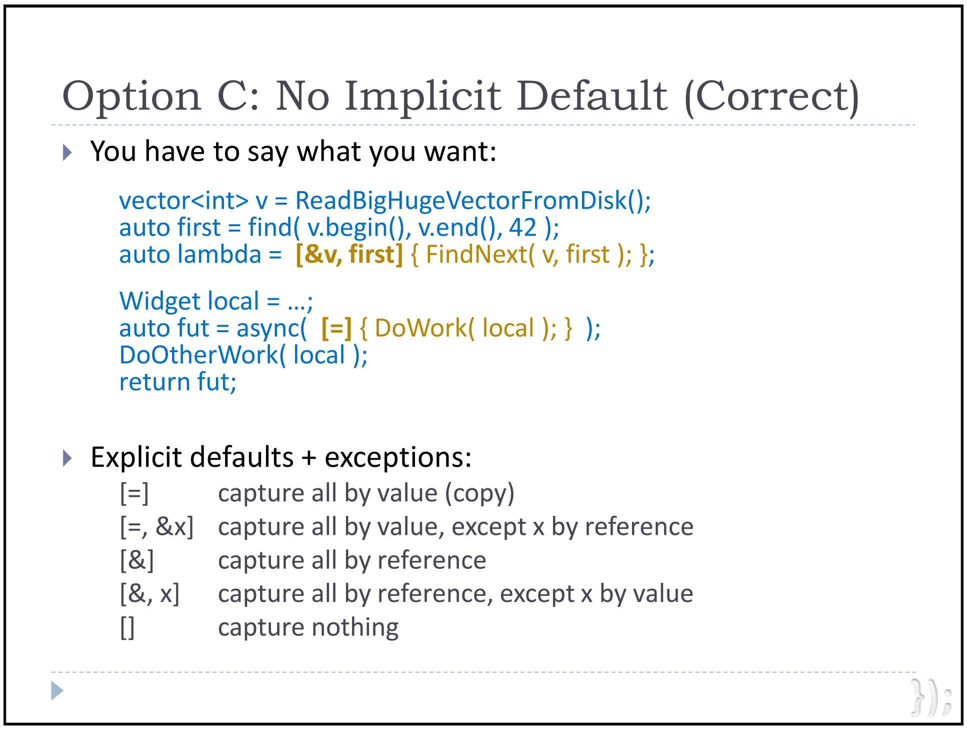


Option B: By Value

- What if we implicitly took all captures by value by default?
- Counterexample:

```
vector<int> v = ReadBigHugeVectorFromDisk();
auto first = find( v.begin(), v.end(), 42 );
auto lambda =  []{ FindNext( v, first ); };
```

- **Q: What would be wrong with this code?**
  - **A1: Performance.**
    Makes a deep copy of the vector, which could be expensive.
  - **A2: Correctness.**
    In the lambda body, *first* is not an iterator into *v*. (Oops.)



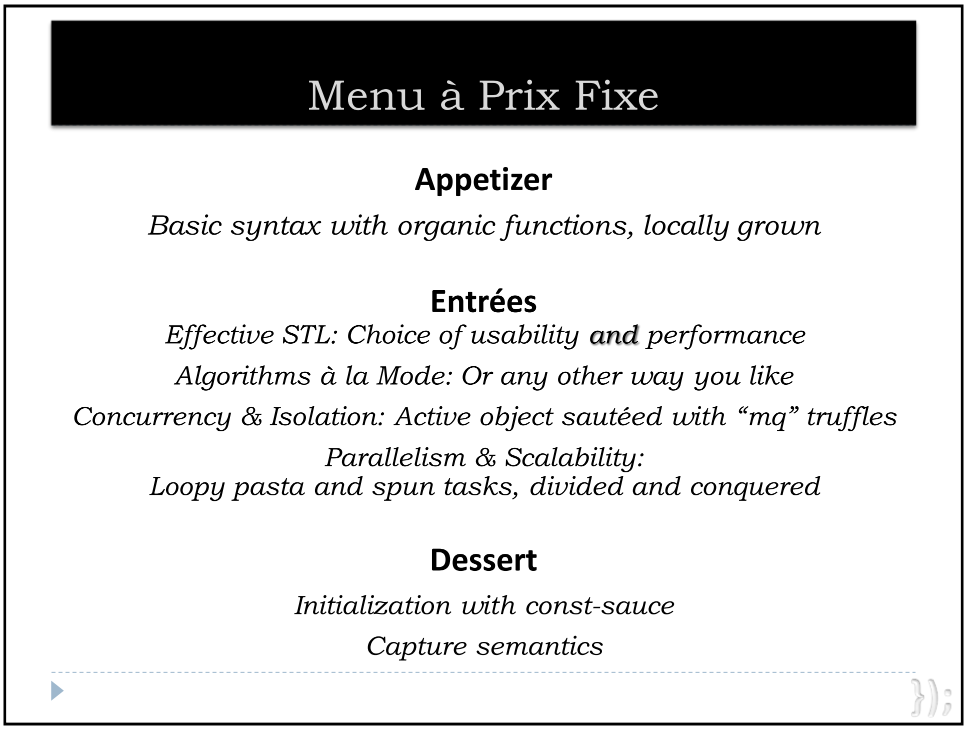
Option C: No Implicit Default (Correct)

- You have to say what you want:

```
vector<int> v = ReadBigHugeVectorFromDisk();
auto first = find( v.begin(), v.end(), 42 );
auto lambda =  [&v, first] { FindNext( v, first ); };

Widget local = …;
auto fut = async(  [=] { DoWork( local ); }  );
DoOtherWork( local );
return fut;
```

- Explicit defaults + exceptions:

[=]	capture all by value (copy)
[=, &x]	capture all by value, except x by reference
[&]	capture all by reference
[&, x]	capture all by reference, except x by value
[]	capture nothing



Menu à Prix Fixe

Appetizer

Basic syntax with organic functions, locally grown

Entrées

*Effective STL: Choice of usability **and** performance*

Algorithms à la Mode: Or any other way you like

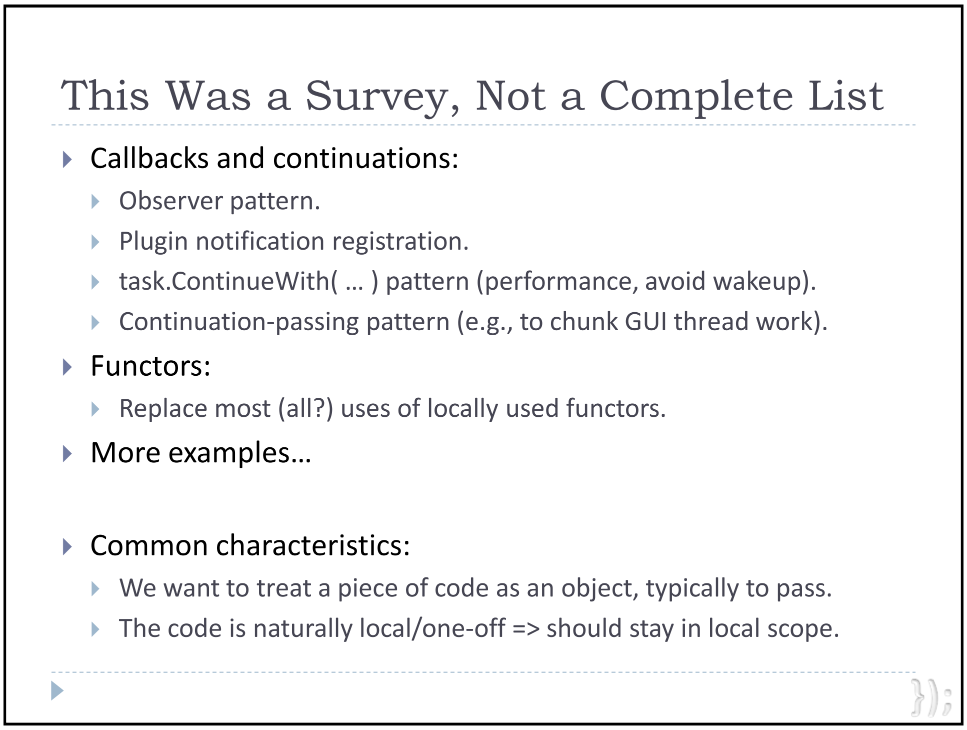
Concurrency & Isolation: Active object sautéed with "mq" truffles

Parallelism & Scalability:
Loopy pasta and spun tasks, divided and conquered

Dessert

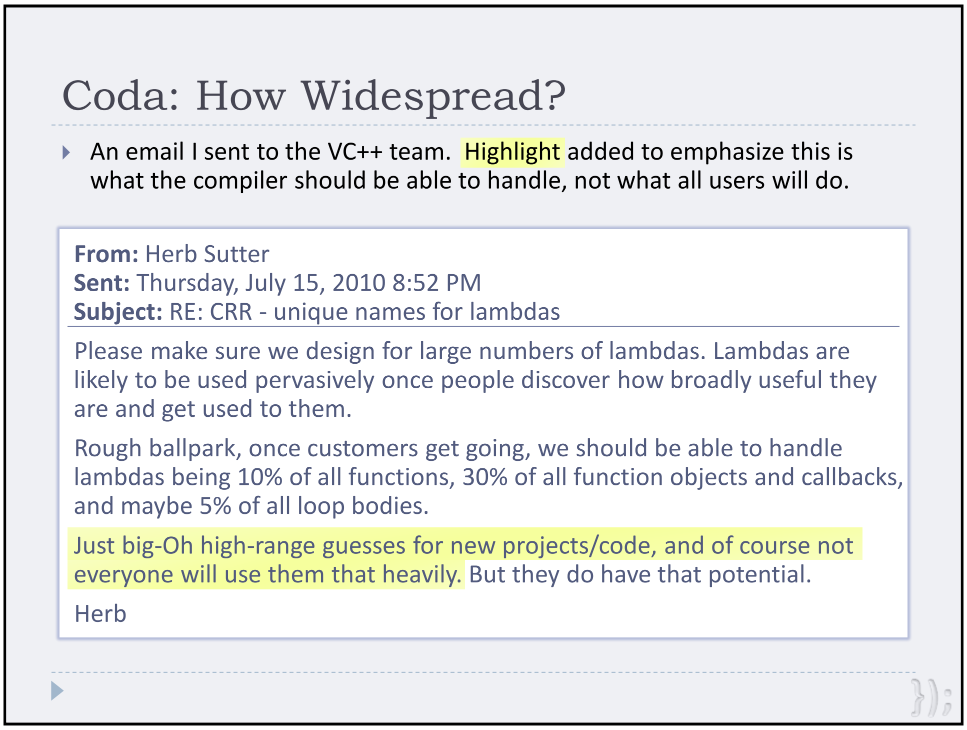
Initialization with const-sauce

Capture semantics



This Was a Survey, Not a Complete List

- Callbacks and continuations:
  - Observer pattern.
  - Plugin notification registration.
  - task.ContinueWith(…) pattern (performance, avoid wakeup).
  - Continuation-passing pattern (e.g., to chunk GUI thread work).
- Functors:
  - Replace most (all?) uses of locally used functors.
- More examples…
- Common characteristics:
  - We want to treat a piece of code as an object, typically to pass.
  - The code is naturally local/one-off => should stay in local scope.



Coda: How Widespread?

- An email I sent to the VC++ team. Highlight added to emphasize this is what the compiler should be able to handle, not what all users will do.

From: Herb Sutter
Sent: Thursday, July 15, 2010 8:52 PM
Subject: RE: CRR - unique names for lambdas

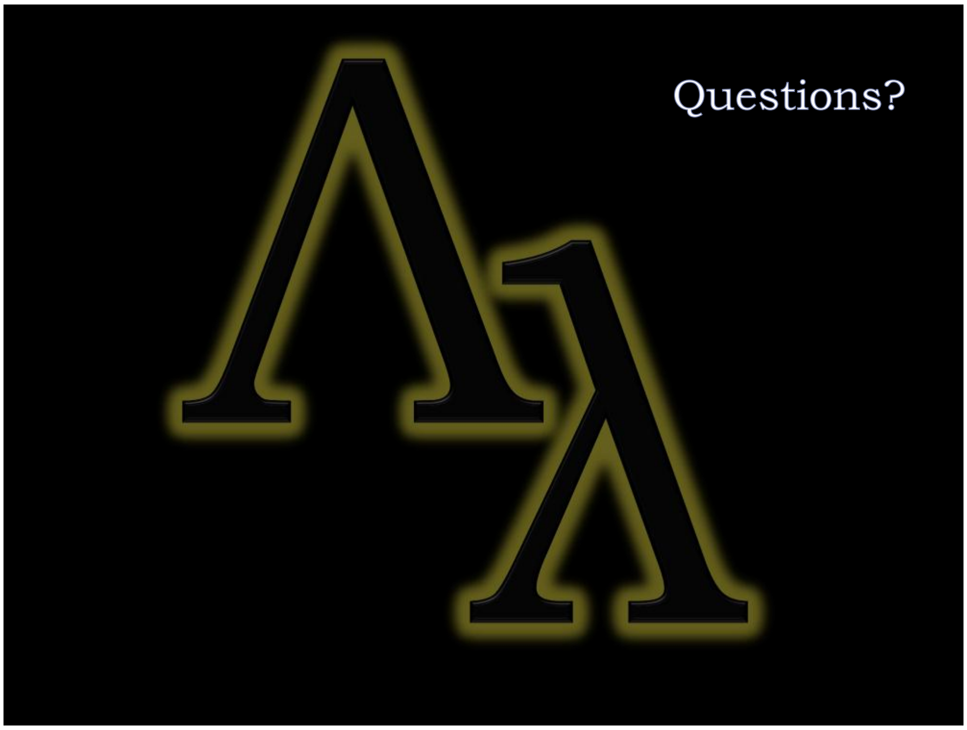
Please make sure we design for large numbers of lambdas. Lambdas are likely to be used pervasively once people discover how broadly useful they are and get used to them.

Rough ballpark, once customers get going, we should be able to handle lambdas being 10% of all functions, 30% of all function objects and callbacks, and maybe 5% of all loop bodies.

Just big-Oh high-range guesses for new projects/code, and of course not everyone will use them that heavily. But they do have that potential.

Herb

});



Questions?