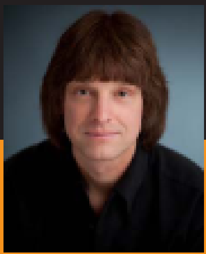


Scott Meyers

Presentation Materials

Overview of The New C++ (C++0x)

artima

Overview of the New C++ (C++0x)

Artima Press is an imprint of Artima, Inc.
P.O. Box 305,Walnut Creek, California 94597

Copyright © 2010 Scott Meyers. All rights reserved.

First version published April 5, 2010
This version published April 30, 2011
Produced in the United States of America

Cover photo by Stephan Jockel. Used with permission.

No part of this publication may be reproduced, modified, distributed, stored in a retrieval system, republished, displayed, or performed, for commercial or noncommercial purposes or for compensation of any kind without prior written permission from Artima, Inc.

This PDF eBook is prepared exclusively for its purchaser, who may use it for personal purposes only, as described by the Artima eBook license (http://www.artima.com/ebook_license.html). In addition, the purchaser may modify this PDF eBook to highlight sections, add comments and annotations, etc., except that the "For the exclusive use of " text that identifies the purchaser may not be modified in any way.

All information and materials in this eBook are provided "as is" and without warranty of any kind.

The term "Artima" and the Artima logo are trademarks or registered trademarks of Artima, Inc. All other company and/or product names may be trademarks or registered trademarks of their owners.

Overview of the New C++ (C++0x)

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/974-1887

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Last Revised: 4/24/11

These are the official notes for Scott Meyers' training course, "Overview of the New C++ (C++0x)". The course description is at http://www.aristeia.com/C++0x.html . Licensing information is at http://aristeia.com/Licensing/licensing.html.

Please send bug reports and improvement suggestions to smeyers@aristeia.com.

In these notes, references to numbered documents preceded by N (e.g., N3290) are references to C++ standardization documents. All such documents are available via http://www.open-std.org/jtc1/sc22/wg21/docs/papers/.

References to sections of draft C++0x are of the form *[chapter.section.subsection]*. Such symbolic names don't change from draft to draft. References also give section numbers and (following a slash) paragraph numbers of specific drafts; those numbers may vary across drafts. Hence [basic.fundamental] (3.9.1/5 in N3290) refers to the section with (permanent) symbolic name [basic.fundamental]—in particular to section 3.9.1 paragraph 5 in N3290.

[Comments in braces, such as this, are aimed at instructors presenting the course. All other comments should be helpful for both instructors and people reading the notes on their own.]

[Day 1 usually ends somewhere in the discussion of the C++0x concurrency API. Day 2 usually goes to the end of the library material.]

© 2011 Scott Meyers, all rights reserved.

Overview

- **Introduction**
  - History, vocabulary, quick C++98/C++0x comparison
- **Features for Everybody**
  - auto, range-based for, lambdas, threads, etc.
- **Library Enhancements**
  - Really more features for everybody
  - TR1-based functionality, forward_list, unique_ptr, etc.
- **Features for Class Authors**
  - Move semantics, perfect forwarding, delegating/inheriting ctors, etc.
- **Features for Library Authors**
  - Variadic templates, decltype, etc.
- **Yet More Features**
- **Removed and Deprecated Features**
- **Further Information**

© 2011 Scott Meyers, all rights reserved.

This course is an *overview*, so there isn't time to cover the details on most features. In general, the features earlier in the course (the ones applicable to more programmers) get more thorough treatments than the features later in the course.

Rvalue references aren't listed on this page, because it's part of move semantics.

© 2011 Scott Meyers, all rights reserved.

History and Vocabulary

1998: ISO C++ Standard officially adopted ("C++98").

- 776 pages.

2003: TC1 ("Technical Corrigendum 1") published ("C++03").

- Bug fixes for C++98.
- 786 pages.

2005: TR1 (Library "Technical Report 1") published.

- 14 likely new components for the standard library.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

History and Vocabulary

2008: Draft for new C++ standard ("C++0x") achieves CD status.

- 13 TR1-derived components plus much more.
- 1265 pages.

2009: (Limited) C++0x feature availability becomes common.

2011: Ratification of new standard expected.

- Final Draft International Standard ("FDIS") approved in March.
  - 1353 pages.
- "C++0x" now effectively a code name.

2012?: TR2

- Additional likely future standard library components.

© 2011 Scott Meyers, all rights reserved.

An overview of support for C++0x features in various compilers is available at http://www.aristeia.com/C++0x/C++0xFeatureAvailability.htm.

Stephan T. Lavavej notes (9/15/09) that "The Boost::FileSystem library was the only thing incorporated into TR2 before work on it was paused."

© 2011 Scott Meyers, all rights reserved.

Sample Code Caveat

Some of the code in this course is untested :-(

- Compilers don't support all features or combinations of features.

I *believe* the code is correct, but I offer no guarantee.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

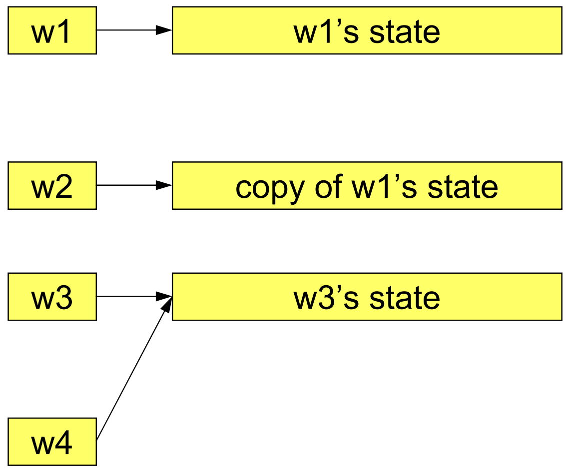
Copying vs. Moving

C++ has always supported copying object state:

- *Copy* constructors, *copy* assignment operators

C++0x adds support for requests to *move* object state:

```
Widget w1;
...
// copy w1's state to w2
Widget w2(w1);

Widget w3;
...
// move w3's state to w4
Widget w4(std::move(w3));
```



Note: w3 continues to exist in a valid state after creation of w4.

© 2011 Scott Meyers, all rights reserved.

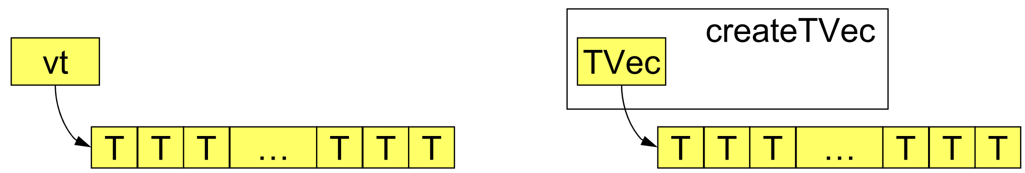
The diagrams on this slide make up a PowerPoint animation.

© 2011 Scott Meyers, all rights reserved.

Copying vs. Moving

Temporary objects are prime candidates for moving:

```
typedef std::vector<T> TVec;
TVec createTVec();                 // factory function
TVec vt;
…
vt = createTVec();                 // in C++98, copy return value to
                                   // vt, then destroy return value
```



© 2011 Scott Meyers, all rights reserved.

The diagrams on this slide make up a PowerPoint animation.

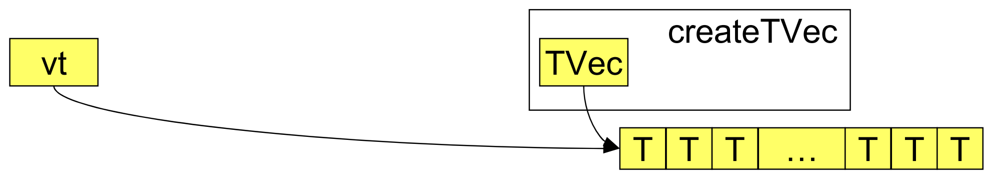
In this discussion, I use a container of T, rather than specifying a particular type, e.g., container of string or container of int. The motivation for move semantics is largely independent of the types involved, although the larger and more expensive the types are to copy, the stronger the case for moving over copying.

© 2011 Scott Meyers, all rights reserved.

Copying vs. Moving

C++0x turns such copy operations into move requests:

```
TVec vt;
…
vt = createTVec();                  // implicit move request in C++0x
```



© 2011 Scott Meyers, all rights reserved.

The diagrams on this slide make up a PowerPoint animation.

© 2011 Scott Meyers, all rights reserved.

Copying vs. Moving

Move semantics examined in detail later, but:

- **Moving a key new C++0x idea.**
  - Usually an optimization of copying.
- Most standard types in C++0x are *move-enabled*.
  - They support move requests.
  - E.g., STL containers.
- Some types are *move-only*:
  - Copying prohibited, but moving is allowed.
  - E.g., stream objects, std::thread objects, std::unique_ptr, etc.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Sample C++98 vs. C++0x Program

List the 20 most common words in a set of text files.

```
countWords Alice_in_Wonderland.txt War_and_Peace.txt
           Dracula.txt The_Kama_Sutra.txt The_Iliad.txt
70544 words found.  Most common:
  the             58272
  and             34111
  of              27066
  to              26992
  a               16937
  in              14711
  his             12615
  he              11261
  that            11059
  was              9861
  with             9780
  I                8663
  had              6737
  as               6714
  not              6608
  her              6446
  is               6277
  at               6202
  on               5981
  for              5801
```


© 2011 Scott Meyers, all rights reserved.

The data shown is from the plain text versions of the listed books as downloaded from Project Gutenberg (http://www.gutenberg.org/).

© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++98

```
#include <cstdio>                    // easier than iostream for formatted output
#include <iostream>
#include <iterator>
#include <string>
#include <fstream>
#include <algorithm>
#include <vector>
#include <map>

typedef std::map<std::string, std::size_t> WordCountMapType;

WordCountMapType wordsInFile(const char * const fileName)    // for each word
{                                                            // in file, return
  std::ifstream file(fileName);                              // # of
  WordCountMapType wordCounts;                               // occurrences

  for (std::string word; file >> word; ) {
    ++wordCounts[word];
  }

  return wordCounts;
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 11

It would be better software engineering to have wordsInFile check the file name for validity and then call another function (e.g., "wordsInStream") to do the actual counting, but the resulting code gets a bit more complicated in the serial case (C++98) and yet more complicated in the concurrent case (C++0x), so to keep this example program simple and focused on C++0x features, we assume that every passed file name is legitimate, i.e., we embrace the "nothing could possibly go wrong" assumption.

© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++98

```
struct Ptr2Pair2ndGT {                                                      // compare 2nd
  template<typename It>                                                     // components of
  bool operator()(It it1, It it2) const { return it1->second > it2->second; }  // pointed-to pairs
};

template<typename MapIt>                                                    // print n most
void showCommonWords(MapIt begin, MapIt end, const std::size_t n)           // common words
{                                                                           // in [begin, end)
  typedef std::vector<MapIt> TempContainerType;
  typedef typename TempContainerType::iterator IterType;

  TempContainerType wordIters;
  wordIters.reserve(std::distance(begin, end));
  for (MapIt i = begin; i != end; ++i) wordIters.push_back(i);

  IterType sortedRangeEnd = wordIters.begin() + n;

  std::partial_sort(wordIters.begin(), sortedRangeEnd, wordIters.end(), Ptr2Pair2ndGT());

  for (IterType it = wordIters.begin();
       it != sortedRangeEnd;
       ++it) {
    std::printf("  %-10s%10u\n", (*it)->first.c_str(), (*it)->second);
  }
}
```


© 2011 Scott Meyers, all rights reserved.

Using range initialization for wordIters (i.e., "TempContainerType wordIters(begin, end);") would be incorrect, because we want wordIters to hold the iterators themselves, not what they point to.

The use of "%u" to print an object of type std::size_t is technically incorrect, because there is no guarantee that std::size_t is of type unsigned. (It could be e.g., unsigned long.) The technically portable solution is probably to use the "%lu" format specifier and to cast (it*)->second to unsigned long (or to replace use of printf with iostreams), but I'm taking the lazy way out and ignoring the issue. Except in this note :-)

© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++98

```
int main(int argc, const char** argv)       // take list of file names on command line,
{                                           // print 20 most common words within
  WordCountMapType wordCounts;

  for (int argNum = 1; argNum < argc; ++argNum) {
    const WordCountMapType results =                  // copy map returned by
      wordsInFile(argv[argNum]);                      // wordsInFile (modulo
                                                      // compiler optimization)

    for ( WordCountMapType::const_iterator i = results.begin();
          i != results.end();
          ++i) {
      wordCounts[i->first] += i->second;
    }
  }

  std::cout << wordCounts.size() << " words found.  Most common:\n" ;

  const std::size_t maxWordsToShow = 20;
  showCommonWords( wordCounts.begin(), wordCounts.end(),
                   std::min(wordCounts.size(), maxWordsToShow));
}
```


© 2011 Scott Meyers, all rights reserved.

results is initialized by copy constructor, which, because WordCountMapType is a map holding strings, could be quite expensive. Because this is an initialization (rather than an assignment), compilers may optimize the copy operation away.

Technically, maxWordsToShow should be of type WordCountMapType::size_type instead of std::size_t, because there is no guarantee that these are the same type (and if they are not, the call to std::min likely won't compile), but I am unaware of any implementations where they are different types, and using the officially correct form causes formatting problems in the side-by-side program comparison coming up in a few slides, so I'm cutting a corner here.

© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++0x

```
#include <cstdio>
#include <iostream>
#include <iterator>
#include <string>
#include <fstream>
#include <algorithm>
#include <vector>
#include <unordered_map>
#include <future>

using WordCountMapType = std::unordered_map<std::string, std::size_t>;

WordCountMapType wordsInFile(const char * const fileName)    // for each word
{                                                            // in file, return
   std::ifstream file(fileName);                             // # of
   WordCountMapType wordCounts;                              // occurrences

   for (std::string word; file >> word; ) {
      ++wordCounts[word];
   }

   return wordCounts;
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++0x

```
template<typename MapIt>                                            // print n most
void showCommonWords(MapIt begin, MapIt end, const std::size_t n)   // common words
{                                                                   // in [begin, end)
  // typedef std::vector<MapIt> TempContainerType;
  //  typedef typename TempContainerType::iterator IterType;

  std::vector<MapIt> wordIters;
  wordIters.reserve(std::distance(begin, end));
  for (auto i = begin; i != end; ++i) wordIters.push_back(i);

  auto sortedRangeEnd = wordIters.begin() + n;

  std::partial_sort(wordIters.begin(), sortedRangeEnd, wordIters.end(),
                    [](MapIt it1, MapIt it2){ return it1->second > it2->second; });

  for (auto it = wordIters.cbegin();
       it != sortedRangeEnd;
       ++it) {
    std::printf("  %-10s%10zu\n", (*it)->first.c_str(), (*it)->second);
  }
}
```


© 2011 Scott Meyers, all rights reserved.

sortedRangeEnd is initialized with the result of an expression using begin, not cbegin, because sortedRangeEnd will later be passed to partial_sort, and partial_sort instantiation will fail with a mixture of iterators and const_iterators. The begin and end iterators in that call must be iterators (not const_iterators), because partial_sort will be moving things around.

%z is a format specifier (added in C99). Followed by u, it correctly prints variables of type size_t.

© 2011 Scott Meyers, all rights reserved.

Counting Words Across Files: C++0x

```
int main(int argc, const char** argv)      // take list of file names on command line,
{                                           // print 20 most common words within;
                                            // process files concurrently
   std::vector<std::future<WordCountMapType>> futures;

   for (int argNum = 1; argNum < argc; ++argNum) {
      futures.push_back(std::async([=]{ return wordsInFile(argv[argNum]); }));
   }

   WordCountMapType wordCounts;
   for (auto& f : futures) {
      const auto results = f.get();         // move map returned by wordsInFile

      for (const auto& wordCount : results) {
         wordCounts[wordCount.first] += wordCount.second;
      }
   }

   std::cout << wordCounts.size() << " words found.  Most common:\n" ;

   const std::size_t maxWordsToShow = 20;
   showCommonWords( wordCounts.begin(), wordCounts.end(),
                    std::min(wordCounts.size(), maxWordsToShow));
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 16

This code has the main thread wait for each file to be processed on a separate thread rather than processing one of the files itself. That's just to keep the example simple.

© 2011 Scott Meyers, all rights reserved.

Comparison

```
#include <cstdio>
#include <iostream>
#include <iterator>
#include <string>
#include <fstream>
#include <algorithm>
#include <vector>
#include <map>


typedef std::map<std::string, std::size_t>
    WordCountMapType;

WordCountMapType
wordsInFile(const char * const fileName)
{
    std::ifstream file(fileName);
    WordCountMapType wordCounts;

    for (std::string word; file >> word; ) {
        ++wordCounts[word];
    }

    return wordCounts;
}
```

```
#include <cstdio>
#include <iostream>
#include <iterator>
#include <string>
#include <fstream>
#include <algorithm>
#include <vector>
#include <unordered_map>
#include <future>

using WordCountMapType =
    std::unordered_map<std::string, std::size_t>;

WordCountMapType
wordsInFile(const char * const fileName)
{
    std::ifstream file(fileName);
    WordCountMapType wordCounts;

    for (std::string word; file >> word; ) {
        ++wordCounts[word];
    }

    return wordCounts;
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Comparison

```
struct Ptr2Pair2ndGT {
    template<typename It>
    bool operator()(It it1, It it2) const
    { return it1->second > it2->second; }
};

template<typename MapIt>
void showCommonWords(MapIt begin, MapIt end,
                                    const std::size_t n)
{
    typedef std::vector<MapIt> TempContainerType;
    typedef typename TempContainerType::iterator IterType;

    TempContainerType wordIters;
    wordIters.reserve(std::distance(begin, end));
    for (MapIt i = begin; i != end; ++i) wordIters.push_back(i);

    IterType sortedRangeEnd = wordIters.begin() + n;

    std::partial_sort( wordIters.begin(), sortedRangeEnd,
                       wordIters.end(), Ptr2Pair2ndGT());

    for ( IterType it = wordIters.begin();
          it != sortedRangeEnd;
          ++it) {
        std::printf("  %-10s%10u\n", (*it)->first.c_str(),
                    (*it)->second);
    }
}
```

```
template<typename MapIt>
void showCommonWords(MapIt begin, MapIt end,
                                    const std::size_t n)
{
    std::vector<MapIt> wordIters;
    wordIters.reserve(std::distance(begin, end));
    for (auto i = begin; i != end; ++i) wordIters.push_back(i);

    auto sortedRangeEnd = wordIters.begin() + n;

    std::partial_sort( wordIters.begin(), sortedRangeEnd,
                       wordIters.end(),
                       [](MapIt it1, MapIt it2)
                       { return it1->second > it2->second; });

    for (auto it = wordIters.cbegin();
          it != sortedRangeEnd;
          ++it) {
        std::printf("  %-10s%10zu\n", (*it)->first.c_str(),
                    (*it)->second);
    }
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Comparison

```
int main(int argc, const char** argv)
{
  WordCountMapType wordCounts;

  for (int argNum = 1; argNum < argc; ++argNum) {





    const WordCountMapType results =
      wordsInFile(argv[argNum]);
    for ( WordCountMapType::const_iterator i = results.begin();
          i != results.end();
          ++i) {
      wordCounts[i->first] += i->second;
    }
  }

  std::cout  << wordCounts.size()
             << " words found.  Most common:\n" ;

  const std::size_t maxWordsToShow = 20;
  showCommonWords( wordCounts.begin(), wordCounts.end(),
                   std::min(wordCounts.size(),
                            maxWordsToShow));
}
```

```
int main(int argc, const char** argv)
{
  std::vector<std::future<WordCountMapType>> futures;

  for (int argNum = 1; argNum < argc; ++argNum) {
    futures.push_back(
      std::async([=]{ return wordsInFile(argv[argNum]); })
    );
  }

  WordCountMapType wordCounts;
  for (auto& f : futures) {
    const auto results =
      f.get();
    for (const auto& wordCount : results) {


      wordCounts[wordCount.first] += wordCount.second;
    }
  }

  std::cout  << wordCounts.size()
             << " words found.  Most common:\n" ;

  const std::size_t maxWordsToShow = 20;
  showCommonWords(wordCounts.begin(), wordCounts.end(),
                  std::min(wordCounts.size(),
                           maxWordsToShow));
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Overview

- Introduction
- **Features for Everybody**
- Library Enhancements
- Features for Class Authors
- Features for Library Authors
- Yet More Features
- Further Information

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

“>>”as Nested Template Closer

“>>” now closes a nested template when possible:

```
std::vector<std::list<int>> vi1;      // fine in C++0x, error in C++98
```

The C++98 “extra space” approach remains valid:

```
std::vector<std::list<int> > vi2;     // fine in C++0x and C++98
```

For a shift operation, use parentheses:

- I.e., “>>” now treated like “>” during template parsing.

```
const int n = … ;                         // n, m are compile-
const int m = … ;                         // time constants

std::array<int, n>m?n:m > a1;             // error (as in C++98)

std::array<int, (n>m?n:m) > a2;           // fine (as in C++98)

std::list<std::array<int, n>>2 >> L1;     // error in '98: 2 shifts;
                                          // error in '0x: 1st “>>”
                                          // closes both templates

std::list<std::array<int, (n>>2) >> L2;   // fine in C++0x,
                                          // error in '98 (2 shifts)
```


© 2011 Scott Meyers, all rights reserved.

[std::array has not yet been introduced.]

© 2011 Scott Meyers, all rights reserved.

auto for Type Declarations

auto variables have the type of their initializing expression:

```
auto x1 = 10;                     // x1: int

std::map<int, std::string> m;
auto i1 = m.begin();              // i1: std::map<int, std::string>::iterator
```

const/volatile and reference/pointer adornments may be added:

```
const auto *x2 = &x1;             // x2: const int*

const auto& i2 = m;               // i2: const std::map<int, std::string>&
```

To get a const_iterator, use the new cbegin container function:

```
auto ci = m.cbegin();             // ci: std::map<int, std::string>::const_iterator
```

- cend, crbegin, and crend exist, too.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

auto for Type Declarations

Type deduction for auto is akin to that for template parameters:

```
template<typename T> void f(T t);
…
f(expr);                  // deduce t's type from expr
auto v = expr;            // do essentially the same thing for v's type
```


© 2011 Scott Meyers, all rights reserved.

Rules governing auto are in [dcl.spec.auto](7.1.6.4 in N3290).

As far as I know, the only way that auto type deduction is not the same as template parameter type deduction is when deducing a type from brace initialization lists. auto deduces "{ x, y, z }" to be of type std::initializer_list<T> (where T is the type of x, y, and z), but template parameter deduction does not apply to brace initialization lists. (It's a "non-deduced context.")

As noted in the discussion on rvalue references, the fact that auto uses the type deduction rules as templates means that variables of type auto&& may, after reference collapsing, turn out to be lvalue references:

```
int x;
auto&& a1 = x;                // x is lvalue, so type of a1 is int&
auto&& a2 = std::move(x);     // std::move(x) is rvalue, so type of a2 is int&&
```


© 2011 Scott Meyers, all rights reserved.

auto for Type Declarations

For variables *not* explicitly declared to be a reference:

- Top-level consts/volatiles in the initializing type are ignored.
- Array and function names in initializing types decay to pointers.

```
const std::list<int> li;
auto v1 = li;                // v1: std::list<int>
auto& v2 = li;               // v2: const std::list<int>&
float data[BufSize];
auto v3 = data;              // v3: float*
auto& v4 = data;             // v4: float (&)[BufSize]
```

Examples from earlier:

```
auto x1 = 10;                // x1: int
std::map<int, std::string> m;
auto i1 = m.begin();         // i1: std::map<int, std::string>::iterator
const auto *x2 = &x1;        // x2: const int* (const isn't top-level)
const auto& i2 = m;          // i2: const std::map<int, std::string>&
auto ci = m.cbegin();        // ci: std::map<int, std::string>::const_iterator
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

auto for Type Declarations

auto can be used to declare multiple variables:

```
void f(std::string& s)
{
  auto temp = s, *pOrig = &s;        // temp: std::string,
  …                                  // pOrig: std::string*
}
```

Each initialization must yield the same deduced type.

```
auto i = 10, d = 5.0;                // error!
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

auto for Type Declarations

Both direct and copy initialization syntaxes are permitted.

```
auto v1(expr);          // direct initialization syntax
auto v2 = expr;         // copy initialization syntax
```

For auto, both syntaxes have the same meaning.

© 2011 Scott Meyers, all rights reserved.

The fact that in ordinary initializations, direct initialization syntax can call explicit constructors and copy initialization syntax cannot is irrelevant, because no conversion is at issue here: the type of the initializing expression will determine what type auto deduces.

Technically, if the type of the initializing expression has an explicit copy constructor, only direct initialization is permitted. From Daniel Krügler:

```
struct Explicit {
  Explicit(){}
  explicit Explicit(const Explicit&){}
} ex;

auto ex2 = ex;          // Error

auto ex3(ex);           // OK
```


© 2011 Scott Meyers, all rights reserved.

Range-Based for Loops

Looping over a container can take this streamlined form:

```
std::vector<int> v;
…
for (int i : v) std::cout << i;          // iteratively set i to every
                                         // element in v
```

The iterating variable may also be a reference:

```
for (int& i : v) std::cout << ++i;       // increment and print
                                         // everything in v
```

auto, const, and volatile are allowed:

```
for (auto i : v) std::cout << i;                // same as above
for (auto& i : v) std::cout << ++i;             // ditto
for (volatile int i : v) someOtherFunc(i);      // or "volatile auto i"
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Range-Based for Loops

Valid for any type supporting the notion of a *range*.

- Given object obj of type T, begin(obj) and end(obj) are valid.

Includes:

- All C++0x library containers.
- Arrays and valarrays.
- Initializer lists.
- Regular expression matches.
- Any UDT T where begin(T) and end(T) yield suitable iterators.

© 2011 Scott Meyers, all rights reserved.

[Initializer lists, regular expressions, and tuples have not yet been introduced.]

Iteration over regular expression matches is supported, because std::match_results offers begin and end member functions for iterating over submatches within the match.

"UDT" = "User Defined Type".

© 2011 Scott Meyers, all rights reserved.

Range-Based for Loops

Examples:

```
std::unordered_multiset<std::shared_ptr<Widget>> msspw;
…
for (const auto& p : msspw) {
  std::cout << p << '\n';                    // print pointer value
}

short vals[ArraySize];
…
for (auto& v : vals) { v = -v; }
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 29

[unordered_multiset and shared_ptr have not yet been introduced.]

The loop variable p is declared a reference, because copying the shared_ptrs in msspw would cause otherwise unnecessary reference count manipulations, which could have a performance impact in multi-threaded code (or even in single-threaded code where shared_ptr uses thread-safe reference count increments/decrements).

© 2011 Scott Meyers, all rights reserved.

Range-Based for Loop Specification

```
for ( iterVarDeclaration : expression ) statementToExecute
```

is essentially equivalent to

```
{
  auto&& range = expression;
  for (auto  b = begin(range), e = end(range);
       b != e;
       ++b ) {
    iterVarDeclaration = *b;
    statementToExecute
  }
}
```

Standardese somewhat more complex.

© 2011 Scott Meyers, all rights reserved.

This slide is for reference only and is not expected to be self-explanatory. Among the details not mentioned are that (1) arrays get special handling rather than calling begin/end, (2) when using ADL to find begin/end, the versions in the standard namespace are always available, and (3) *expression* may be a braced initializer list.

© 2011 Scott Meyers, all rights reserved.

Range-Based for Loops

Range form valid only for for-loops.

- Not do-loops, not while-loops.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

nullptr

A new keyword. Indicates a null pointer.

- Convertible to any pointer type and to bool, but nothing else.
  - ➡ Can't be used as an integral value.

```
const char *p = nullptr;          // p is null
if (p) …                          // code compiles, test fails
int i = nullptr;                  // error!
```

Traditional uses of 0 and NULL remain valid:

```
int *p1 = nullptr;                // p1 is null
int *p2 = 0;                      // p2 is null
int *p3 = NULL;                   // p3 is null
if (p1 == p2 && p1 == p3) …       // code compiles, test succeeds
```


© 2011 Scott Meyers, all rights reserved.

The term "keyword" is stronger than "reserved word." Keywords are unconditionally reserved (except as attribute names, sigh), while, e.g., "main" is reserved only when used as the name of a function at global scope.

The type of nullptr is std::nullptr_t. Other pointer types may be cast to this type via static_cast (or C-style cast). The result is always a null pointer.

© 2011 Scott Meyers, all rights reserved.

nullptr

Only nullptr is unambiguously a pointer:

```
void f(int *ptr);            // overloading on ptr and int
void f(int val);

f(nullptr);                  // calls f(int*)

f(0);                        // calls f(int)

f(NULL);                     // probably calls f(int)
```

- The last call compiles unless NULL isn't defined to be 0
  - E.g., it could be defined to be 0L.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

nullptr

Unlike 0 and NULL, nullptr works well with forwarding templates:

```
template<typename F, typename P>       // make log entry, then
void logAndCall(F func, P param)       // invoke func on param
{
  …                                    // write log entry
  func(param);
}

void f(int* p);                        // some function to call

f(0);                                  // fine
f(nullptr);                            // also fine

logAndCall(f, 0);                      // error! P deduced as
                                       // int, and f(int) invalid

logAndCall(f, NULL);                   // error!

logAndCall(f, nullptr);                // fine, P deduced as
                                       // std::nullptr_t, and
                                       // f(std::nullptr_t) is okay
```


© 2011 Scott Meyers, all rights reserved.

nullptr thus meshes with C++0s's support for perfect forwarding, which is mentioned later in the course.

© 2011 Scott Meyers, all rights reserved.

Unicode Support

Two new character types:

```
char16_t        // 16-bit character (if available);
                // akin to uint_least16_t
char32_t        // 32-bit character (if available);
                // akin to uint_least32_t
```

Literals of these types prefixed with u/U, are UCS-encoded:

```
u'x'            // 'x' as a char16_t using UCS-2
U'x'            // 'x' as a char32_t using UCS-4/UTF-32
```

C++98 character types still exist, of course:

```
'x'             // 'x' as a char
L'x'            // 'x' as a wchar_t
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 35

From draft C++0 ([basic.fundamental], 3.9.1/5 in N3290): "Types char16_t and char32_t denote distinct types with the same size, signedness, and alignment as uint_least16_t and uint_least32_t, respectively, in <stdint.h>, called the underlying types."

UCS-2 is a 16-bit/character encoding that matches the entries in the Basic Multilingual Plane (BMP) of UTF-16. UTF-16 can use surrogate pairs to represent code points outside the BMP. UCS-2 cannot. UCS-4 and UTF-32 are essentially identical.

char16_t character literals can represent only UCS-2, because it's not possible to fit a UTF-16 surrogate pair (i.e., two 16-bit values) in a single char16_t object. Notes [lex.ccon] (2.14.3/2 in N3290), "A character literal that begins with the letter u, such as u'y', is a character literal of type char16_t. ... If the value is not representable within 16 bits, the program is ill-formed."

© 2011 Scott Meyers, all rights reserved.

Unicode Support

There are corresponding string literals:

```
u"UCS-2 string literal"              // ⇒ char16_ts in UTF-16
U"UCS-4 string literal"              // ⇒ char32_ts in UCS-4/UTF-32
"Ordinary/narrow string literal"     // "ordinary/narrow" ⇒ chars
L"Wide string literal"               // "wide" ⇒ wchar_ts
```

UTF-8 string literals are also supported:

```
u8"UTF-8 string literal"             // ⇒ chars in UTF-8
```

Code points can be specified via \u*nnnn* and \U*nnnnnnnn*:

```
u8"G clef: \U0001D11E"                     // )
u"Thai character Khomut: \u0E5B"           // ๛
U"Skull and crossbones: \u2620"            // ☠
```


© 2011 Scott Meyers, all rights reserved.

A code point is a specific character/glyph, i.e., a specific member of the Unicode character set. UTF-8 and UTF-16 are multibyte encodings, UCS-n and UTF-32 are fixed-size encodings. All except UCS-2 can represent every code point of the full Unicode character set. UTF-8, UTF-16, and UCS-4/UTF-32 are all defined by ISO 10646 as well as by the Unicode standard. Per the Unicode FAQ (http://unicode.org/faq/unicode_iso.html), "Although the character codes and encoding forms are synchronized between Unicode and ISO/IEC 10646, the Unicode Standard imposes additional constraints on implementations to ensure that they treat characters uniformly across platforms and applications. To this end, it supplies an extensive set of functional character specifications, character data, algorithms and substantial background material that is *not* in ISO/IEC 10646."

Although u-qualified character literals are not permitted to yield UTF-16 surrogate pairs, characters in u-qualified string literals appear to be. Per [lex.string] (2.14.5/9 in N3290), "A char16_t string literal ... is initialized with the given characters. A single *c-char* may produce more than one char16_t character in the form of surrogate pairs.."

The results of appending string literals of different types (if supported) are implementation-defined:

```
u8"abc" "def" u"ghi"      // implementation-defined results
```


© 2011 Scott Meyers, all rights reserved.

Unicode Support

There are std::basic_string typedefs for all character types:

```
std::string s1;        // std::basic_string<char>
std::wstring s2;       // std::basic_string<wchar_t>
std::u16string s3;     // std::basic_string<char16_t>
std::u32string s4;     // std::basic_string<char32_t>
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Conversions Among Encodings

C++98 guarantees only two codecvt facets:

- char ⇄ char (std::codecvt<char, char, std::mbstate_t>)
  - "Degenerate" – no conversion performed.
- wchar_t ⇄ char (std::codecvt<wchar_t, char, std::mbstate_t>)

C++0x adds:

- UTF-16 ⇄ UTF-8 (std::codecvt<char16_t, char, std::mbstate_t>)
- UTF-32 ⇄ UTF-8 (std::codecvt<char32_t, char, std::mbstate_t>)
- UTF-8 ⇄ UCS-2, UTF-8 ⇄ UCS-4 (std::codecvt_utf8)
- UTF-16 ⇄ UCS-2, UTF-16 ⇄ UCS-4 (std::codecvt_utf16)
- UTF-8 ⇄ UTF-16 (std::codecvt_utf8_utf16)
  - Behaves like std::codecvt<char16_t, char, std::mbstate_t>.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 38

The "degenerate" char ⇄ char conversion allows for code to be written that always pipes things through a codecvt facet, even in the (common) case where no conversion is needed. Such behavior is essentially mandated for std::basic_filebuf in both C++98 and C++0x.

P.J. Plauger, who proposed codecvt_utf8_utf16 for C++0x, explains the two seemingly redundant UTF-16 ⇄ UTF-8 conversion instantiations: "The etymologies of the two are different. There should be no behavioral difference."

© 2011 Scott Meyers, all rights reserved.

Conversions Among Encodings

C++98 supports only IO-based conversions.

- Designed for multibyte external strings ⇄ wide internal strings.
- Requires changing locale associated with stream.

New in C++0x:

- std::wbuffer_convert does IO-based encoding conversions w/o changing stream locale.
- std::wstring_convert does in-memory encoding conversions.
  - E.g., std::u16string/std::u32string ⇒ std::string.

Usage details esoteric, hence omitted in this overview.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 39

Changing the locale associated with a stream is accomplished via the imbue member function, which is a part of several standard iostream classes, e.g., ios_base.

Among the esoteric details are that the existence of protected destructors mean that none of the the standard code_cvt facets work with std::wbuffer_convert and std::wstring_convert. Instead, users must derive classes from the standard facets and add public destructors. More information on this issue (and others) is in the comp.std.c++ thread at http://tinyurl.com/ykup5qe.

© 2011 Scott Meyers, all rights reserved.

Raw String Literals

String literals where "special" characters aren't special:

- E.g., escaped characters and double quotes:

```
std::string noNewlines(R"(\n\n)");
std::string cmd(R"(ls /home/docs | grep ".pdf")");
```

- E.g., newlines:

```
std::string withNewlines(R"(Line 1 of the string...
                            Line 2...
                            Line 3)");
```

"Rawness" may be added to any string encoding:

```
LR"(Raw Wide string literal \t (without a tab))"
u8R"(Raw UTF-8 string literal \n (without a newline))"
uR"(Raw UTF-16 string literal \\ (with two backslashes))"
UR"(Raw UTF-32 string literal \u2620 (without a code point))"
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 40

"R" must come after "u8", "u", "U", etc. – it can't come in front of those specifiers.

© 2011 Scott Meyers, all rights reserved.

Raw String Literals

Raw text delimiters may be customized:

- Useful when)" is in raw text, e.g., in regular expressions:

```
std::regex re1(R"!("operator\(\)"|"operator->")!");       // "operator()"|
                                                           // "operator->"
std::regex re2(R"xyzzy("\([A-Za-z_]\w*\)")xyzzy");        // "(identifier)"
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 41

Green text shows what would be interpreted as closing the raw string if the default raw text delimiters were being used.

Custom delimiter text (e.g., xyzzy in re2's initializer) must be no more than 16 characters in length and may not contain whitespace.

The backslashes in front of the parentheses inside the regular expressions are to prevent them from being interpreted as demarcating capture groups.

\w means a word character (i.e., letter, digit, or underscore).

© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

C++98 offers multiple initialization forms.

- Initialization ≠ assignment.
  - E.g., const objects can be initialized, not assigned.

Examples:

```
const int y(5);                    // "direct initialization" syntax
const int x = 5;                   // "copy initialization" syntax

int arr[] = { 5, 10, 15 };         // brace initialization

struct Point1 { int x, y; };
const Point1 p1 = { 10, 20 };      // brace initializtion

class Point2 {
public:
  Point2(int x, int y);
};

const Point2 p2(10, 20);           // function call syntax
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

None of the consts on this page are important to the examples. They're present only to emphasize that we are talking about *initialization*.

© 2011 Scott Meyers, all rights reserved.

Initialization in C++98

Containers require another container:

```
int vals[] = { 10, 20, 30 };
const std::vector<int> cv(vals, vals+3);       // init from another
                                               // container
```

Member and heap arrays are impossible:

```
class Widget {
public:
  Widget(): data(???) {}
private:
  const int data[5];                           // not initializable
};

const float * pData = new const float[4];      // not initializable
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

Brace initialization syntax now allowed everywhere:

```
const int val1 {5};
const int val2 {5};

int a[] { 1, 2, val1, val1+val2 };

struct Point1 { … };                    // as before
const Point1 p1 {10, 20};

class Point2 { … };                     // as before
const Point2 p2 {10, 20};               // calls Point2 ctor

const std::vector<int> cv { a[0], 20, val2 };

class Widget {
public:
  Widget(): data {1, 2, a[3], 4, 5} {}
private:
  const int data[5];
};

const float * pData = new const float[4] { 1.5, val1-val2, 3.5, 4.5 };
```


© 2011 Scott Meyers, all rights reserved.

When initializing a data member via brace initializer, the brace initializer may be enclosed in parentheses, e.g., the Widget constructor above could be written like this:

```
Widget(): data({1, 2, a[3], 4, 5}) {}
```


© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

Really, everywhere:

```
Point2 makePoint() { return { 0, 0 }; }         // return expression;
                                                // calls Point2 ctor

void f(const std::vector<int>& v);              // func. declaration

f({ val1, val2, 10, 20, 30 });                  // function argument
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

Semantics differ for aggregates and non-aggregates:

- Aggregates (e.g., arrays and structs):
  - Initialize members/elements beginning-to-end.
- Non-aggregates:
  - Invoke a constructor.

© 2011 Scott Meyers, all rights reserved.

The technical definition of an aggregate is slightly more flexible than what's above. From [dcl.init.aggr] (8.5.1/1 in N3290): "An aggregate is an array or a class with no user-provided constructors, no [default] initializers for non-static data members, no private or protected non-static data members, no base classes, and no virtual functions."

Uniform initialization syntax can be used with unions, but only the first member of the union may be so initialized:

```
union u { int a; char* b; };
u a = { 1 };                // okay
u d = { 0, "asdf" };        // error
u e = { "asdf" };           // error (can't initialize an int with a char array)
```

Per [dcl.init.list] (8.5.4/4 in N3290), elements in an initialization list are evaluated left to right.

© 2011 Scott Meyers, all rights reserved.

Brace-Initializing Aggregates

Initialize members/elements beginning-to-end.

- Too many initializers ⇒ error.
- Too few initializers ⇒ remaining objects are *value-initialized*:
  - Built-in types initialized to 0.
  - UDTs with constructors are default-constructed.
  - UDTs without constructors: members are value-initialized.

```
struct Point1 { int x, y; };                 // as before
const Point1 p1 = { 10 };                    // same as { 10, 0 }
const Point1 p2 = { 1, 2, 3 };               // error!  too many
                                             // initializers
```

  - std::array is also an aggregate:

```
long f();
std::array<long, 3> arr = { 1, 2, f(), 4, 5 };   // error!  too many
                                                 // initializers
```


© 2011 Scott Meyers, all rights reserved.

"UDT" = "User Defined Type".

C++0x does not support C99's designated initializers:

```
struct Point {
  int x, y, z;
};
Point p { .x = 5, .z = 8 };                  // error!
```


© 2011 Scott Meyers, all rights reserved.

Brace-Initializing Non-Aggregates

Invoke a constructor.

```
class Point2 {                          // as before
public:
    Point2(int x, int y);
};

short a, b;
…
const Point2 p1 {a, b};                 // same as p1(a, b)
const Point2 p2 {10};                   // error!  too few ctor args
const Point2 p3 {5, 10, 20};            // error!  too many ctor args
```

- True even for containers (details shortly):

```
std::vector<int> v { 1, a, 2, b, 3 };              // calls a vector ctor
std::unordered_set<float> s { 0, 1.5, 3 };         // calls an
                                                   // unordered_set ctor
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

Use of "=" with brace initialization typically allowed:

```
const int val1 = {5};
const int val2 = {5};

int a[] = { 1, 2, val1, val1+val2 };

struct Point1 { … };
const Point1 p1 = {10, 20};

class Point2 { … };
const Point2 p2 = {10, 20};

const std::vector<int> cv = { a[0], 20, val2 };
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

But not always:

```
class Widget {
public:
  Widget(): data = {1, 2, a[3], 4, 5} {}               // error!
private:
  const int data[5];
};

const float * pData =
  new const float[4] = { 1.5, val1-val2, 3.5, 4.5 };   // error!

Point2 makePoint() { return = { 0, 0 }; }              // error!

void f(const std::vector<int>& v);                     // as before

f( = { val1, val2, 10, 20, 30 });                      // error!
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

And "*T var = expr*" syntax can't call explicit constructors:

```
class Widget {
public:
  explicit Widget(int);
  …
};

Widget w1(10);      // okay, direct init: explicit ctor callable
Widget w2{10};      // ditto

Widget w3 = 10;     // error! copy init: explicit ctor not callable
Widget w4 = {10};   // ditto
```

Develop the habit of using brace initialization without "=".

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Syntax

Uniform initialization syntax a feature *addition*, not a replacement.

- Almost all initialization code valid in C++98 remains valid.
  - ➡ Rarely a need to modify existing code.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Brace Initialization and Implicit Narrowing

Sole exception: implicit narrowing.

- C++98 allows it via brace initialization, C++0x doesn't:

```
struct Point { int x, y; };
Point p1 { 1, 2.5 };                          // fine in C++98:
                                              // implicit double ⇒ int
                                              // conversion;
                                              // error in C++0x
Point p2 { 1, static_cast<int>(2.5) };        // fine in both C++98
                                              // and C++0x
```


© 2011 Scott Meyers, all rights reserved.

Narrowing conversions are defined in [decl.init.list] (8.5.4/7 in N3290). Basically, a conversion is narrowing if (1) the target type can't represent all the values of the source type and (2) the compiler can't guarantee that the source value will be within the range of the target type, e.g.,

```
int x { 2.5 };          // error: all conversions from floating point
                        // to integer type are narrowing
double d { x };         // error: double can't exactly represent all ints
unsigned u { x };       // error: unsigned can't represent all ints
unsigned u { 25 };      // okay: compiler knows that unsigned can represent 25
```


© 2011 Scott Meyers, all rights reserved.

Brace Initialization and Implicit Narrowing

Direct constructor calls and brace initialization thus differ subtly:

```
class Widget {
public:
  Widget(unsigned u);
  …
};

int i;
…
Widget w1(i);                 // okay, implicit int ⇒ unsigned
Widget w2 {i};                // error! int ⇒ unsigned narrows

unsigned u;

Widget w3(u);                 // fine
Widget w4 {u};                // also fine, same as w3's init.
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Initializer Lists

A mechanism to generalize array aggregate initialization:

- Available to all UDTs.

```
int x, y;
…
int a[] { x, y, 7, 22, -13, 44 };            // array initialization
std::vector<int> v { 99, -8, x-y, x*x };     // std. library type
Widget w { a[0]+a[1], x, 25, 16 };           // arbitrary UDT
```

- Available for more than just initialization, e.g.

```
std::vector<int> v {};                       // initialization
v.insert(v.end(), { 99, 88, -1, 15 });       // multi-element insertion
v = { 0, 1, x, y };                          // replace v's value
```

➡ *Any* function can use an “initializer” list.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 55

“UDT” = “User Defined Type”.

The statement

```
v = { 0, 1, x, y };
```

creates no temporary vector for the assignment, because there’s a vector::operator= taking a parameter of type std::initializer_list.

© 2011 Scott Meyers, all rights reserved.

Initializer Lists

Approach startlingly simple:

- Brace initializer lists convertible to std::initializer_list objects.
- Functions can declare parameters of this type.
- std::initializer_list stores initializer values in an array and offers these member functions:
  - size // # of elements in the array
  - begin // ptr to first array element
  - end // ptr to one-beyond-last array element

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.

There are no cbegin/cend member functions for initializer_list, presumably because initializer_list<T>::begin and initializer_list<T>::end both return const T*. There are no rbegin/rend member functions, either, presumably because initialization lists are supposed to be processed front-to-back.

In the standard library, std::initializer_list objects are always passed by value. On gcc 4.5 and MSVC 10, sizeof(std::initializer_list<T>) is 8.

Scott Meyers, Software Development Consultant © 2011 Scott Meyers, all rights reserved.

Initializer Lists

```
#include <initializer_list>          // necessary header
std::u16string getName(int ID);      // lookup name with given ID

class Widget {
public:
  Widget(std::initializer_list<int> nameIDs)
  {
    names.reserve(nameIDs.size());
    for (auto id: nameIDs) names.push_back(getName(id));
  }
private:
  std::vector<std::u16string> names;
};
...
Widget w { a[0]+a[1], x, 25, 16 };   // copies values into an array
                                     // wrapped by an initializer_list
                                     // passed to the Widget ctor.
```


© 2011 Scott Meyers, all rights reserved.

The idea behind this example is that the Widget is initialized with a list of IDs, which are then converted into UTF-16-formatted names during construction. The names are stored in the Widget.

Move semantics would be used when passing the result of getName to push_back.

© 2011 Scott Meyers, all rights reserved.

Initializer Lists

std::initializer_list parameters may be used with other parameters:

```
class Widget {
public:
  Widget(const std::string& name, double epsilon,
         std::initializer_list<int> il);
  …
};

std::string name("Buffy");

Widget w { name, 0.5,        // same as
           {5, 10, 15} };    // Widget w(name, 0.5,
                             //          std::initializer_list({5,10,15}));
```

- Note the nested brace sets.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Initializer Lists

They may be templatized:

```
class Widget {
public:
  template<typename T> Widget(std::initializer_list<T> il);
  ...
};

...

Widget w1 { -55, 25, 16 };          // fine, T = int
```

Only homogeneous initializer lists allow type deduction to succeed:

```
Widget w2 { -55, 2.5, 16 };         // error, T can't be deduced
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Initializer Lists and Overload Resolution

When resolving constructor calls, std::initializer_list parameters are preferred for brace-delimited arguments:

```
class Widget {
public:
  Widget(double value, double uncertainty);        // #1
  Widget(std::initializer_list<double> values);    // #2
  …
};

double d1, d2;
…
Widget w1 { d1, d2 };                              // calls #2
```

For **brace-delimited arguments**:

```
Widget w2(d1, d2);                                 // calls #1
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Initializer Lists and Overload Resolution

std::initializer_list parameters are *always* preferred over other types:

```
class Widget {
public:
  Widget(double value, double uncertainty);          // #1
  Widget(std::initializer_list<std::string> values); // #2
  …
};

double d1, d2;
…
Widget w1 { d1, d2 };          // tries to call #2; fails because
                               // no double ⇒ string conversion
```

Braced initializers are viewed as std::initializer_lists if at all possible.

True only for **braced initializers**:

```
Widget w2(d1, d2);             // still calls #1
```


© 2011 Scott Meyers, all rights reserved.

The relevant parts of draft C++0x wrt this topic are [over.match.list] (13.3.1.7 in N3290), [dcl.init.list] (8.5.4/2-3 in N3290), and [temp.deduct.call] (14.8.2.1/1 in N3290).

© 2011 Scott Meyers, all rights reserved.

Initializer Lists and Overload Resolution

Given multiple std::initialization_list candidates, best match is determined by worst element conversion:

```
class Widget {
public:
  Widget(std::initializer_list<int>);           // #1
  Widget(std::initializer_list<double>);        // #2
  Widget(std::initializer_list<std::string>);   // #3

  Widget(int, int, int);              // due to above ctors, this ctor not
};                                    // considered for "{... }" args

Widget w1 { 1, 2.0, 3 };              // int ⇒ double same rank as
                                      // double ⇒ int, so ambiguous

Widget w2 { 1.0f, 2.0, 3.0 };         // float ⇒ double better than
                                      // float ⇒ int, so calls #2

std::string s;
Widget w3 { s, "Init", "Lists" };     // calls #3
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Initializer Lists and Overload Resolution

If best match involves a narrowing conversion, call is invalid:

```
class Widget {
public:
  Widget(std::initializer_list<int>);

  Widget(int, int, int);          // due to above ctor, not
};                                // considered for "{... }" args

Widget w { 1, 2.0, 3 };          // error!  double ⇒ int narrows
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Uniform Initialization Summary

- Brace initialization syntax now available everywhere.
  - ➡ Aggregates initialized top-to-bottom/front-to-back.
  - ➡ Non-aggregates initialized via constructor.
- Implicit narrowing not allowed.
- std::initializer_list parameters allow "initialization" lists to be passed to functions.
  - ➡ Not actually limited to initialization (e.g., std::vector::insert).

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Lambda Expressions

A quick way to create function objects at their point of use.

```
std::vector<int> v;
…
auto it = std::find_if(v.cbegin(), v.cend(),
                       [](int i) { return i > 0 && i < 10; });
```

Essentially generates:

```
class MagicType1 {
public:
  bool operator()(int i) const { return i > 0  && i < 10; }
};
auto it = std::find_if(v.cbegin(), v.cend(), MagicType1());
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 65

The generated *MagicType* above is not technically accurate, because closure types aren't default-constructible, but that detail isn't important for understanding the essence of what lambdas do.

I ignore mutable lambdas in this course, because use cases for them are uncommon, and this course is an overview, not an exhaustive treatment. I also ignore how capture-by-value retains the cv qualifiers of the captured variable, because, again, situations in which this is relevant are uncommon.

© 2011 Scott Meyers, all rights reserved.

Lambda Expressions

Another example:

```
typedef std::shared_ptr<Widget> SPWidget;
std::deque<SPWidget> d;
…
std::sort(d.begin(), d.end(),
          [](const SPWidget& sp1, const SPWidget& sp2)
          { return *sp1 < *sp2; });
```

Essentially generates:

```
class MagicType2 {
public:
  bool operator()(const SPWidget& p1, const SPWidget& p2) const
  { return *p1 < *p2; }
};
std::sort(d.begin(), d.end(), MagicType2());
```

Function objects created through lambda expressions are closures.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 66

Again, the generated *MagicType* above is not technically accurate, because closure types aren't default-constructible.

In this example, it would be possible to pass the parameters by value without changing the correctness of the code, but that would cause the shared_ptr reference counts to be modified, which could have a performance impact in multi-threaded code (or even in single-threaded code where shared_ptr uses thread-safe reference count increments/decrements).

Per [expr.prim.lambda] (5.1.2/2 in N3290), closures are rvalues (prvalues, to be precise).

© 2011 Scott Meyers, all rights reserved.

Variable References in Lambdas

Closures may outlive their creating function:

```
std::function<bool(int)> returnLambda(int a)  // return type to be
{                                             // discussed soon
  int b, c;
  ...
  return [](int x)                            // won't compile, but
         { return a*x*x + b*x + c == 0; };    // assume it would
}
auto f = returnLambda(10);                    // f is essentially a
                                              // copy of λ's closure
```

In this call,

```
if (f(22)) ...                                // "invoke the lambda"
```

what are the values of a, b, c?

- returnLambda no longer active!

© 2011 Scott Meyers, all rights reserved.

[std::function has not yet been introduced.]

"λ" is the (lowercase) Greek letter lambda.

"Invoke the lambda" is in quotes, because we're really invoking the copy of the lambda's closure that's stored inside the std::function object.

© 2011 Scott Meyers, all rights reserved.

Variable References in Lambdas

This version has no such problem:

```
int a;                                          // now at global or
                                                // namespace scope
std::function<bool(int)> returnLambda()
{
  static int b, c;                              // now static
  ...
  return  [](int x)                             // now compiles
          { return a*x*x + b*x + c == 0; };
}
auto f = returnLambda();                        // as before

...

if (f(22)) ...                                  // as before
```

a, b, c outlive returnLambda's invocation.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Variable References in Lambdas

Rules for variables lambdas may refer to:

- **Locals in the calling context referenceable only if "captured."**

```
std::function<bool(int)> returnLambda(int a)
{
  int b, c;
  ...
  return [](int x){ return a*x*x + b*x + c == 0; };   // to compile, must
}                                                      // capture a, b, c;
                                                       // this example
                                                       // won't compile
```

- **Non-locals always referenceable.**

```
int a;
std::function<bool(int)> returnLambda()
{
  static int b, c;
  ...
  return [](int x){ return a*x*x + b*x + c == 0; };   // no need to
}                                                      // capture a, b, c
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Capturing Local Variables

Capturing locals puts copies in closures:

```
{
  int minVal;
  double maxVal;
  …
  auto it = std::find_if(v.cbegin(), v.cend(),
                         [minVal, maxVal](int i)
                         { return i > minVal && i < maxVal; });
}
```

Corresponds to:

```
class MagicType {
public:
  MagicType(int v1, double v2): _minVal(v1), _maxVal(v2) {}
  bool operator()(int i) const { return i > _minVal && i < _maxVal; }
private:
  int _minVal;
  double _maxVal;
};
auto it = std::find_if(v.cbegin(), v.cend(), MagicType(minVal, maxVal));
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 70

There is no way to capture a move-only type. A workaround is to store the move-only type in a std::shared_ptr (e.g., std::shared_ptr(std::thread)), but that requires the creator of the lambda to create a std::shared_ptr that can then be copied into the closure. Another workaround is to eschew use of a lambda and manually create a custom functor class.

© 2011 Scott Meyers, all rights reserved.

Capturing Local Variables

Captures may also be by reference:

```
{
  int minVal;
  double maxVal;
  …
  auto it = std::find_if( v.cbegin(), v.cend(),
                          [&minVal, &maxVal](int i)
                          { return i > minVal && i < maxVal; });
}
```

Corresponds to:

```
class MagicType {
public:
  MagicType(int& v1, double& v2): _minVal(v1), _maxVal(v2) {}
  bool operator()(int i) const { return i > _minVal && i < _maxVal; }
private:
  int& _minVal;
  double& _maxVal;
};

auto it = std::find_if(v.cbegin(), v.cend(),                // same as
                       MagicType(minVal, maxVal));          // before
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 71

There is no "capture by const reference," although const locals captured by reference are essentially captured by const reference.

© 2011 Scott Meyers, all rights reserved.

Capturing Local Variables

Different locals may be captured differently:

```
{
  int minVal;
  double maxVal;
  …
  auto it = std::find_if(v.cbegin(), v.cend(),
                         [minVal, &maxVal](int i)
                         { return i > minVal && i < maxVal; });
}
```

Corresponds to:

```
class MagicType {
public:
  MagicType(int v1, double& v2): _minVal(v1), _maxVal(v2) {}
  bool operator()(int i) const { return i > _minVal && i < _maxVal; }
private:
  int _minVal;
  double& _maxVal;
};

auto it = std::find_if(v.cbegin(), v.cend(),                   // same as
                       MagicType(minVal, maxVal));              // before
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Capturing Local Variables

Capture mode defaults may be specified:

```
auto it = std::find_if( v.cbegin(), v.cend(),                 // default is
                        [=](int i)                            // by value
                        { return i > minVal && i < maxVal; });
auto it = std::find_if( v.cbegin(), v.cend(),                 // default is
                        [&](int i)                            // by ref
                        { return i > minVal && i < maxVal; });
```

With a default capture mode, captured variables need not be listed.

- As in examples above.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Capturing Local Variables

Default overridable on a per-variable basis:

```
auto it = std::find_if(v.cbegin(), v.cend(),          // default capture is
                       [=, &maxVal](int i)            // by value, but maxVal
                       { return i > minVal &&         // is by reference
                                i < maxVal; });
```

Corresponds to:

```
class MagicType {
public:
  MagicType(int v1, double& v2): _minVal(v1), _maxVal(v2) {}
  bool operator()(int i) const { return i > _minVal && i < _maxVal; }
private:
  int _minVal;
  double& _maxVal;
};

auto it = std::find_if(v.cbegin(), v.cend(), MagicType(minVal, maxVal));
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Capturing Class Members

To access class members within a member function, capture this:

```
class Widget {
public:
  void doSomething();
private:
  std::list<int> li;
  int minVal;
};

void Widget::doSomething() {
  auto it = std::find_if(li.cbegin(), li.cend(),
                         [minVal](int i) { return i > minVal; }      // error!
    );
  …
}

void Widget::doSomething() {
  auto it = std::find_if(li.cbegin(), li.cend(),
                         [this](int i) { return i > minVal; }        // fine
    );
  …
}
```


© 2011 Scott Meyers, all rights reserved.

Lambdas used in a member function yield closure types defined in that member function, hence within the class containing the member function. That's what makes it possible for the closure's operator() to refer to all members of the class, e.g., to minVal in the lambda on this page. There's no need for friendship, because the closure type is within (i.e., part of) the class.

© 2011 Scott Meyers, all rights reserved.

Capturing Class Members

A default capture mode also makes this available:

```
class Widget {
public:
  void doSomething();
private:
  std::list<int> li;
  int minVal;
};

void Widget::doSomething() {
  auto it = std::find_if(li.cbegin(), li.cend(),
                         [=](int i) { return i > minVal; }     // fine
    );
  …
}

void Widget::doSomething() {
  auto it = std::find_if(li.cbegin(), li.cend(),
                         [&](int i) { return i > minVal; }     // also fine, same
    );                                                         // effect (for “this”)
  …
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 76

Capturing this by reference may be less efficient than capturing it by value, because going through the reference requires double indirection (modulo compiler optimizations).

© 2011 Scott Meyers, all rights reserved.

Lambda Return Types

Optional when:

- Return type is void.
- Lambda body is "return *expr*;"
  - ➡ Return type is that of *expr*.

Otherwise must be specified via trailing return type syntax:

```
std::vector<double> v;
…
std::transform( v.begin(), v.end(), v.begin(),
                []( double d)->double
                {
                  makeLogEntry("std::transform", d);
                  return std::sqrt(std::abs(d));
                });
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Trailing Return Types

- Must be used with lambdas (when a return type is given).
- Often useful with decltype (described later).
- Permitted for any function (with a leading auto):

```
void f(int x);                  // traditional syntax

auto f(int x)->void;            // same declaration with trailing
                                // return type

class Widget {
public:
  void mf1(int x);              // traditional
  auto mf2() const -> bool;     // trailing return type
};
```

➡ Non-lambda non-decltype uses not expected to be common.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Lambdas without Parameter Lists

Lambdas without parameters may omit the parameter list.

- Such functions especially useful with threads:

```
void doWork(int x, int y);
void doMoreWork();

std::thread t1([]() { doWork(10, 20); doMoreWork(); });  // w/empty
                                                         // param list

std::thread t2([] { doWork(10, 20); doMoreWork(); });    // w/o empty
                                                         // param list
```

Omitting the optional parentheses seems to be common.

© 2011 Scott Meyers, all rights reserved.

[std::thread has not yet been introduced.]

mutable lambdas may not omit the parameter list, but this course does not discuss mutable lambdas.

© 2011 Scott Meyers, all rights reserved.

Lambda Expression Complexity

Lambdas may be arbitrarily complex:

- Multiple statements, multiple returns.
- Throw/catch exceptions.
- Essentially anything allowed in a "normal" function.

Maintainability considerations suggest:

- Short, clear, context-derived lambdas are best.

© 2011 Scott Meyers, all rights reserved.

Not absolutely everything allowed in a normal function is allowed in a lambda expression, e.g., there is no way to refer to the this pointer of the operator() function generated from the lambda.

© 2011 Scott Meyers, all rights reserved.

Storing Closures

Closure types not specified, but two easy ways to store closures:

- auto:

```
auto multipleOf5 = [](long x) { return x % 5 == 0; };
std::vector<long> vl;
…
vl.erase(std::remove_if(vl.begin(), vl.end(), multipleOf5), vl.end());
```

- std::function:

```
std::function<bool(long)> multipleOf5 =      // see next page for syntax
  [](long x) { return x % 5 == 0; };
…
vl.erase(std::remove_if(vl.begin(), vl.end(), multipleOf5), vl.end());
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.
Slide 81

Every lambda expression yields a unique closure type. VC10 names these types anonymous-namespace::<lambda0>, anonymous-namespace::<lambda1>, etc. gcc 4.5 naming is less obvious (e.g., UlvE_, UlvE0_, UlvE1_, etc.).

The closure types are created in the smallest block scope, class scope, or namespace scope that contains the lambda.

Lambdas can't be directly recursive, but the effect can be achieved by having a closure invoke a std::function object that has been initialized with the closure. For example:

```
std::function<int(int)> factorial = [&](int x) { return (x==1) ? 1 : (x * factorial(x-1)); };
```


© 2011 Scott Meyers, all rights reserved.

Specifying Function Types

A function's type is its declaration w/o any names:

```
void f1(int x);                    // type is void(int)
double f2(int x, std::string& s);  // type is double(int, std::string&)
```

C++ uses function types for e.g., std::function:

```
std::function<bool(long)> multipleOf5 =          // from prior slide
  [](long x) { return x % 5 == 0; };
```

Trailing return type syntax is equivalent:

```
std::function<auto (long)->bool> multipleOf5 =   // equivalent to
  [](long x) { return x % 5 == 0; };             // above
```


© 2011 Scott Meyers, all rights reserved.

VC10, despite support for trailing return type syntax in general, does not compile the second declaration of multipleOf5 on this page. gcc 4.5 accepts it.

© 2011 Scott Meyers, all rights reserved.

Storing Closures

auto more efficient than std::function, but not always applicable.

- Not allowed for function parameters or return types:

```
void useIt(auto func);                          // error!
void useIt(std::function<bool(long)> func);     // fine
template<typename Func>
void useIt(Func func);                          // fine, but generates
                                                // multiple functions

auto makeFunc();                                // error!
std::function<bool(long)> makeFunc();           // fine
template<typename Func>
Func makeFunc();                                // fine, but generates
                                                // multiple functions,
                                                // and callers must
                                                // specify Func
```


© 2011 Scott Meyers, all rights reserved.

Regarding efficiency of auto vs. std::function, Stephan T. Lavavej says, "A compiler would have to perform extreme heroics to get function to be as efficient as auto."

© 2011 Scott Meyers, all rights reserved.

Storing Closures

- Not allowed for class data members:

```
class Widget {
private:
  auto func;                       // error!
  ...
};

class Widget {
private:
  std::function<bool(long)> f;     // fine
  ...
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Stored Closures and Dangling References

Stored closures can hold dangling members.

- E.g., pointers to deleted heap objects.
- E.g., references to beyond-scope locals:

```
std::vector<long> vl;
std::function<bool(long)> f;
{                                                       // some block
  int divisor;
  …
  f = [&](long x) { return x % divisor == 0; };         // closure refers
  …                                                     // to local var
}                                                       // local var's
                                                        // scope ends
vl.erase(std::remove_if(vl.begin(), vl.end(), f),       // calls to f use
         vl.end());                                     // dangling ref!
```

It's your responsibility to avoid such problems.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Lambdas as Container Comparison Functions

Pass the closure to the container constructor:

```
auto cmpFnc = [](int *pa, int *pb)                // compare values,
              { return *pa < *pb; };              // not pointers

std::set<int*, decltype(cmpFnc)> s(cmpFnc);      // sort s that way
```


© 2011 Scott Meyers, all rights reserved.

[decltype has not been introduced yet.]

Closure types are not default-constructible, so this will fail:

```
std::set<int*, decltype(cmpFnc)> s;      // error! comparison object can't be
                                         // constructed
```


© 2011 Scott Meyers, all rights reserved.

Lambda/Closure Summary

- Lambda expressions generate closures.
- Calling state can be captured by value or by reference.
- Return types, when specified, use trailing return type syntax.
- Closures can be stored using auto or std::function.
  - ➡Be alert for dangling references/pointers in stored closures.
- Short, clear, context-derived lambdas are best.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Template Aliases

using declarations can now be used for "partially bound" templates:

```
template<typename T>
using MyAllocVec = std::vector<T, MyAllocator>;

MyAllocVec<int> v;                    // std::vector<int, MyAllocator>

template<std::size_t N>
using StringArray = std::array<std::string, N>;

StringArray<15> sa;                   // std::array<std::string, 15>

template<typename K, typename V>
using MapGT = std::map<K, V, std::greater<K>>;

MapGT<long long,                          // std::map<long long,
          std::shared_ptr<std::string>>   //   std::shared_ptr<std::string>,
  myMap;                                  //   std::greater<long long>>
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Template Aliases

Template aliases may not be specialized:

```
template<typename T>                                  // from prior
using MyAllocVec = std::vector<T, MyAllocator>;       // page

template<typename T>
using MyAllocVec = std::vector<T*, MyPtrAllocator>;   // error!
```

To achieve this effect, use a traits class:

```
template<typename T>                                  // primary
struct VecAllocator {                                 // template
  typedef MyAllocator type;
};

template<typename T>                                  // specialized
struct VecAllocator<T*> {                             // template
  typedef MyPtrAllocator type;
};

template<typename T>
using MyAllocVec = std::vector<T, typename VecAllocator<T>::type>;
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

using as typedef

Without templatization, usings can be equivalent to typedefs:

```
typedef std::unordered_set<int> IntHash;   // these 2 lines do
using IntHash = std::unordered_set<int>;   // the same thing
```

using declarations can be more comprehensible:

```
typedef void (*CallBackPtr)(int);          // func. ptr. typedef
using CallBackPtr = void (*)(int);         // equivalent using decl.
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Concurrency Support

Primary components:

- Threads for independent units of execution.
- std::async and Futures for asynchronous calls.
- Mutexes for controlled access to shared data.
- Condition Variables for block-until-true execution.
- Thread-Local Data for thread-specific data.

API relatively low level, but has some interesting generality.

Headers:

- <thread>
- <mutex>
- <condition_variable>
- <future>

© 2011 Scott Meyers, all rights reserved.

This course is about C++0x, not concurrency, so I assume that attendees are familiar with the basic issues in threading, including when it should and shouldn't be used, races, synchronization, deadlock, testing, etc. The feature list on this page is not exhaustive, and near the end of the concurrency discussion is a bullet list of "other features."

© 2011 Scott Meyers, all rights reserved.

Threads

std::thread takes any "callable object" and runs it asynchronously:

```
void doThis();
class Widget {
public:
  void operator()() const;
  void normalize(long double, int, std::vector<float>);
  …
};
std::thread t1(doThis);                // run function asynch.
Widget w;
…
std::thread t2(w);                     // "run" function object asynch.
```

To pass arguments, a lambda can be used:

```
long double ld;
int x;
std::thread t3([=]{ w.normalize(ld, x, { 1, 2, 3 }); });   // "run" closure
                                                           // asynch.
```


© 2011 Scott Meyers, all rights reserved.

Behavior with multiple threads is largely the same as classic single-threaded C/C++ behavior, with generalizations added as needed. Objects of static storage duration continue to have only one representation in a program, and although they are guaranteed to be initialized in a race-free fashion, unsynchronized access may cause races. If an exception is not caught by a thread (any thread), std::terminate is called.

If main exits and other threads are still running, they are, in Anthony Williams' words, "terminated abruptly," which essentially means you get undefined behavior.

Threads cannot be started in a suspended state, but the underlying platform-specific thread handle should be available via std::thread::native_handle.

Threads cannot be forcibly killed, but std::thread_handle may provide a platform-specific way. (Posix has no such functionality; pthread_cancel is cooperative.)

© 2011 Scott Meyers, all rights reserved.

Data Lifetime Considerations

Functions called in ST sytems know that outside data are "frozen:"

- They won't be destroyed during the call.
- Only the called function can change their value.

```
int x, y, z;
Widget *pw;
…
f(x, y);                              // call in ST system
```

➡ During f's execution:
- x, y, z, and pw will continue to exist.
- *pw will continue to exist unless f causes pw to be deleted.
- Their values will change only through f's actions.

➡ True regardless of how f declares its parameters:

```
void f(int xParam, int yParam);
void f(int& xParam, int& yParam);
void f(const int& xParam, const int& yParam);
```


© 2011 Scott Meyers, all rights reserved.

"ST" = "Single-Threaded".

© 2011 Scott Meyers, all rights reserved.

Data Lifetime Considerations

No data is inherently frozen in an asynchronous call.

```
int x, y, z;
Widget *pw;
…
call f(x, y) asynchronously (i.e., on a new thread);
```

- During f's execution:
  - x, y, z, and pw might go out of scope.
  - *pw might be deleted.
  - The values of x, y, z, pw and *pw might change.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Data Lifetime Considerations

Details depend on how f declares its parameters:

```
void f(int xParam, int yParam);                          // pass by value:
                                                         // f unaffected by
                                                         // changes to x, y

void f(int& xParam, int& yParam);                        // pass by ref:
void f(const int& xParam, const int& yParam);            // f affected by
                                                         // changes to x, y

int x, y, z;
Widget *pw;
…
call f(x, y) asynchronously;
```

No declaration insulates f from changes to z, pw, and *pw.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Data Lifetime Considerations

Conclusions:

- Data lifetime issues critical in multi-threading (MT) design.
  - ➡ A special aspect of synchronization/race issues.
    - ◆ Even shared immutable data subject to lifetime issues.
- By-reference/by-pointer parameters in asynch calls always risky.
  - ➡ Prefer pass-by-value.
    - ◆ Including via lambdas!

```
void f(int xParam);              // function to call asynchronously

{
  int x;
  …
  std::thread t1([&]{ f(x); });  // risky!  closure holds a ref to x
  std::thread t2([=]{ f(x); });  // okay, closure holds a copy of x
  …
}                                // x destroyed
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 96

In the case of t1, the lambda's closure object is created before the calling thread can continue, but the calling thread may continue before f starts executing. By the time f's parameter xParam is initialized, the closure may hold a dangling reference to x, because x has already been destroyed.

© 2011 Scott Meyers, all rights reserved.

Avoiding Lifetime Problems

Two basic strategies:

- Copy data for use by the asynchronous call.
- Ensure referenced objects live long enough.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Copying Arguments for Asynchronous Calls

std::thread's variadic constructor (conceptually) copies everything:

```
void f(int xVal, const Widget& wVal);

int x;
Widget w;
…
std::thread t(f, x, w);              // invoke copy of f on copies of x, w
```

- Copies of f, x, w, guaranteed to exist until asynch call returns.
- Inside f, wVal refers to a *copy* of w, not w itself.

Copying optimized to moving whenever possible.

- Details when we do move semantics.

© 2011 Scott Meyers, all rights reserved.

In this example, "copying" f really means copying a pointer to it, and "optimizing" this copy to a move makes no sense, because copying a pointer is cheap. The general rule, however, is that the thread constructor copies/moves its first parameter, i.e., the function to be executed asynchronously.

© 2011 Scott Meyers, all rights reserved.

Copying Arguments

Using by-value captures in closures works, too:

```
void f(int xVal, const Widget& wVal);

int x;
Widget w;
…
std::thread t([=]{ f(x, w); });        // invoke copy of f on copies of x, w
```

- Closure contains copies of x and w.
- Closure copied by thread ctor; copy exists until f returns.
  - Copying optimized to moving whenever possible.
- Inside f, wVal refers to a *copy* of w, not w itself.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Copying Arguments

Another approach is based on std::bind:

```
void f(int xVal, const Widget& wVal);

int x;
Widget w;
…
std::thread t(std::bind(f, x, w));        // invoke f with copies of x, w
```

- Object returned by bind contains copies of x and w.
- That object copied by thread ctor; copy exists until f returns.
- Inside f, wVal refers to a *copy* of w, not w itself.

We'll examine std::bind later.

- Lambdas are usually a better choice than bind.
  - Easier for readers to understand.
  - More efficient.

© 2011 Scott Meyers, all rights reserved.

[std::bind has not been introduced yet.]

© 2011 Scott Meyers, all rights reserved.

Copying Arguments

Summary:

- Options for creating argument copies with sufficient lifetimes:
  - Use variadic `thread` constructor.
  - Use lambda with by-value capture.
  - Use `bind`.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Ensuring Sufficient Argument Lifetimes

One way is to delay locals' destruction until asynch call is complete:

```
void f(int xVal, const Widget& wVal);       // as before

{
  int x;
  Widget w;
  …
  std::thread t([&]{ f(x, w); });           // wVal really refers to w
  …
  t.join();                                 // destroy w only after t
}                                           // finishes
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Mixing By-Value and By-Reference Arguments

Given

```
void f(int xVal, int yVal, int zVal, Widget& wVal);
```

what if you really want to pass w by reference?

- **Lambdas**: use by-reference capture:

```
{
  Widget w;
  int x, y, z;
  …
  std::thread t([=, &w]{ f(x, y, z, w); });    // pass copies of x, y, z;
  …                                            // pass w by reference
}
```

➡ You're responsible for avoiding data races on w.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Mixing By-Value and By-Reference Arguments

- **Variadic thread constructor or bind**: Use C++0x's std::ref:
  - Creates objects that act like references.
    - Copies of a std::ref-generated object refer to the same object.

```
void f(int xVal, int yVal, int zVal, Widget& wVal);       // as before

{
  static Widget w;
  int x, y, z;
  …
  std::thread t1(f, x, y, z, std::ref(w));              // pass copies of
  std::thread t2(std::bind(f, x, y, z, std::ref(w)));  //  x, y, z; pass w
  …                                                      // by reference
}
```

  - std::cref also exists (for ref-to-consts), but implicit ref(T) ⇒ const T& means std::ref often suffices.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Asynchronous Calls

Building blocks:

- std::async: Request asynchronous execution of a function.
- Future: token representing function's result.

Unlike raw use of std::thread objects:

- Allows values or exceptions to be returned.
  - ➡Just like "normal" function calls.

© 2011 Scott Meyers, all rights reserved.

This course neither shows nor discusses std::packaged_task or std::promise.

© 2011 Scott Meyers, all rights reserved.

async

```
double bestValue(int x, int y);                    // something callable

std::future<double> f =                            // run λ asynch.;
  std::async( []{ return bestValue(10, 20); } );   // get future for it
…                                                  // do other work
double val = f.get();                              // get result (or
                                                   // exception) from λ
```

As usual, auto reduces verbiage:

```
auto f = std::async( []{ return bestValue(10, 20); } );
…
auto val = f.get();
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 106

The idea behind bestValue is that it computes the optimal value for something given parameters x and y. Presumably, such computation takes a while, hence makes a natural separate task.

Instead of passing only a lambda, std::async may also be passed a function and its arguments (like std::thread), but I don't show any such examples.

© 2011 Scott Meyers, all rights reserved.

async Launch Policies

- **std::launch::async**: function runs on a new thread.
  - ➡ Maintains calling thread's responsiveness (e.g., GUI threads).

```
auto f = std::async(std::launch::async, doBackgroundWork);
```

- **std::launch::deferred**: function runs on calling thread.
  - ➡ Useful for debugging, performance tuning.
  - ➡ Invocation occurs upon get or a waiting call.

```
auto f = std::async(std::launch::deferred,
                    []{ return bestValue(10, 20); });
…
auto val = f.get();    // run λ synchronously here
```

By default, implementation chooses, presumably with goals:

- Take advantage of all hardware concurrency, i.e., scale.
- Avoid oversubscription.

© 2011 Scott Meyers, all rights reserved.

Threads used by std::async may (but need not) be drawn from a thread pool under the as-if rule, but implementions would have to, e.g., destroy and reinitialize thread-local variables before reusing a thread object.

Until November 2010, std::launch::deferred was named std::launch::sync.

When multiple launch policies are permitted (e.g., by specifying std::launch::async | std::launch::deferred), the decision between synchronous and asynchronous execution need not be made before std::async returns.

Motivation for async calls using std::launch::deferred executing only when get/wait is called is in N2973 under "Eager and Lazy Evaluation." Invoking wait_for or wait_until on a future for a deferred function returns the status std::future_status_deferred immediately.

Anthony Williams notes that tasks running synchronously may not use promises or conventional futures: "std::async(std::launch::deferred, *some_function*) [may create] a special type of future holding a deferred function. When you call get or wait on the future, it executes the deferred function."

A std::async-launched task that ends up running on the calling thread will modify the calling thread's thread-local data. Tasks where this is a problem should be run with the std::launch::async policy.

© 2011 Scott Meyers, all rights reserved.

Futures

Two kinds:

- std::future<T>: result may be accessed only once.
  - Suitable for most use cases.
  - Moveable, not copyable.
    - Exactly one future has right to access result.
- std::shared_future<T>: result may be accessed multiple times.
  - Appropriate when multiple threads access a single result.
  - Both copyable and moveable.
    - Multiple std::shared_futures for the same result may exist.
  - Creatable from std::future.
    - Such creation transfers ownership.

© 2011 Scott Meyers, all rights reserved.

Both std::async and std::promise return std::future objects, so the only way to create a non-null std::shared_future is to do it from a std::future object.

Until November 2009, std::future was named std::unique_future.

Regarding implementation of futures, Anthony Williams writes, “Implementations of std::future<T> must provide space for storing a T or a std::exception_ptr, and a means of counting references to the shared state. Additional storage may be required for managing the state, such as a mutex and some flags. In the case of futures arising from the use of std::async, the state must also include storage for the callable object and its arguments (for a policy of std::launch::deferred), or a handle to the new thread (for a policy of std::launch::async).”

© 2011 Scott Meyers, all rights reserved.

Futures

Result retrieval via get:

- Blocks until a return is available, then grabs it.
  - For future<T>, "grabs" ≡ "moves (if possible) or copies."
  - For shared_future<T> or *anyKindOfFuture*<T&>, "grabs" ≡ "gets reference to."
  - "Return" may be an exception (which is then propagated).

© 2011 Scott Meyers, all rights reserved.

Invoking get more than once on a std::future yields undefined behavior. Invoking get more than once on a std::shared_future yields the same result (return value or exception) each time. There is no need to copy such results or exceptions, because (1) non-exceptions are accessed by reference and (2) a copy of an exception is made only if the catch clause catching it catches by value.

© 2011 Scott Meyers, all rights reserved.

Futures

An alternative is wait:

- Blocks until a return is available.

```
std::future<double> f = std::async([]{ return bestValue(10, 20); });
…
f.wait();                          // block until λ is done
```

  - ➡ A timeout may be specified.
    - ◆ Most useful when std::launch::async specified.

```
std::future<double> f =
  std::async(std::launch::async, []{ return bestValue(10, 20); });
…
while (f.wait_for(std::chrono::seconds(0)) !=      // if result of λ
        std::future_status::ready) {               // isn't ready,
  ...                                              // do more work
}
double val = f.get();                              // grab result
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.

For unshared futures, wait_for is most useful when you know the task is running asynchronously, because if it's a deferred task (i.e., slated to run sychronously), calling wait_for will never timeout. (It will just invoke the deferred task synchronously, and you'll have to wait for it to finish.)

The enumerant future_status::ready must be qualified with future_status::, because std::future_status is an enum class, not just an enum.

There is no support for waiting for one of several futures (i.e., something akin to Windows' WaitForMultipleObjects). Anthony Williams writes: "The standard doesn't provide a means to do it, just like you cannot wait on more than one condition variable, more than one mutex or more than one thread in a 'wake me when the first one signals' kind of way. If multiple threads can provide a single result, I would use a promise and a single future. The first thread to set the promise will provide the result to the waiting thread, the other threads will get an exception when they try and set the promise. To wait for all the results, you can just wait on each in turn. The order doesn't matter, since you need to wait for all of them. The only issue is when you need to wait for the first of two unrelated tasks. There is no mechanism for that without polling. I would be tempted to add an additional flag (e.g. with a future or condition variable) which is set by either when ready — you can then wait for the flag to be set and then poll to see which task set it." As for why there is no WaitForMultipleObjects-like support, Anthony writes, "no-one proposed it for anything other than futures, and that didn't make it to the final proposal because we were so short of time. There was also lack of consensus over whether it was actually useful, or what form it should take."

There is similarly no support akin to Unix's select, but select applies only to asynchronous IO (it waits on file handles), and IO is not a part of C++0x's concurrency support.

Scott Meyers, Software Development Consultant © 2011 Scott Meyers, all rights reserved.

void Futures

Useful when callers want to know only when a callee finishes.

- Callable objects returning void.
- Callers uninterested in return value.
  - ➡ But possibly interested in exceptions.

```
void initDataStructs(int defValue);
void initGUI();

std::future<void> f1 = std::async( []{ initDataStructs(-1); } );
std::future<void> f2 = std::async( []{ initGUI(); } );

…                                  // init everything else

f1.get();                          // wait for asynch. inits. to
f2.get();                          // finish (and get exceptions,
                                   // if any)

…                                  // proceed with the program
```


© 2011 Scott Meyers, all rights reserved.

The choice between waiting to join with a thread or for a future depends on several things. First, if you have only a thread or only a future available, you have no choice. If a thread that returns a future throws an exception, that exception is available to the caller via the future, but it is silently discarded if you simply join with the thread (because the future is not read). A caller can poll to see if a future is available (via *future*::wait_for with a timeout of 0), but there is no way to poll to see if a thread is ready to be joined with.

The choice between using wait or get on a void future depends on whether you need a timeout (only wait offers that) and whether you need to know if an exception was thrown (only get offers that). wait can also be used as a signaling mechanism, i.e., to indicate to other threads that an operation has completed side effects they are waiting for. And wait can allow you to force execution of a deferred function at a point other than where you want to retrieve the result.

The example on this page uses get, because it seems likely that if an exception is thrown during asynchronous initialization, the main thread would want to know that.

© 2011 Scott Meyers, all rights reserved.

Mutexes

Four types:

- std::mutex: non-recursive, no timeout support
- std::timed_mutex: non-recursive, timeout support
- std::recursive_mutex: recursive, no timeout support
- std::recursive_timed_mutex: recursive, timeout support

Recursively locking non-recursive mutexes ⇒ undefined behavior.

© 2011 Scott Meyers, all rights reserved.

Mutex objects are neither copyable nor movable. Copying a mutex doesn't really make any sense (you'd end up with multiple mutexes for the same data). Regarding moving, Anthony Williams, in a 6 April 2010 post to comp.std.c++, explained: "Moving a mutex would be disasterous if that move raced with a lock or unlock operation from another thread. Also, the identity of a mutex is vital for its operation, and that identity often includes the address, which means that the mutex CANNOT be moved. Similar reasons apply to condition variables."

© 2011 Scott Meyers, all rights reserved.

RAII Classes for Mutexes

Mutexes typically managed by RAII classes:

- **std::lock_guard**: lock mutex in ctor, unlock it in dtor.

```
std::mutex m;                          // mutex object
{
  std::lock_guard<std::mutex> L(m);    // lock m
  …                                    // critical section
}                                      // unlock m
```

➡ No other operations.
  - No copying/moving, no assignment, no manual unlock, etc.

© 2011 Scott Meyers, all rights reserved.

RAII = "Resource Acquisition is Initialization."

In general, the terminology "lock" seems to mean an RAII or RAII-like class for managing the locking/unlocking of a mutex.

std::lock_guard is neither copyable nor movable. Again, copying makes no sense. Movability is precluded, because, as Daniel Krügler put in a 6 April 2010 comp.std.c++ posting, "lock_guard is supposed to provide the minimum necessary functionality with minimum overhead. If you need a movable lock, you should use unique_lock."

© 2011 Scott Meyers, all rights reserved.

RAII Classes for Mutexes

- **std::unique_lock**: much more flexible.
  - May lock mutex after construction, unlock before destruction.
  - Moveable, but not copyable.
  - Supports timed mutex operations:
    - Try locking, timeouts, etc.
    - Typically the best choice for timed mutexes.

```
using RCM = std::recursive_timed_mutex;      // typedef
RCM m;                                       // mutex object
{
  std::unique_lock<RCM> L(m);                // lock m
  …                                          // critical section
  L.unlock();                                // unlock m
  …
}                                            // nothing happens
```


© 2011 Scott Meyers, all rights reserved.

The name unique_lock is by analogy to unique_ptr. Originally, a "shared_lock" type was proposed (to be a reader/writer lock), but it was not adopted for C++0x.

© 2011 Scott Meyers, all rights reserved.

Additional unique_lock Functionality

```
using TM = std::timed_mutex;                       // typedef
TM m;                                              // mutex object
{
  std::unique_lock<TM> L(m, std::defer_lock);      // associate m with
                                                   // L w/o locking it
  …
  if (L.try_lock_for(std::chrono::microseconds(10))) {
    …                                              // critical section
  } else {
    …                                              // timeout w/o
  }                                                // locking m
  …
  if (L) {                                         // convert to bool
    …                                              // critical section
  } else {
    …                                              // m isn't locked
  }
}                                                  // if m locked, unlock
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Multiple Mutex Acquisition

Acquiring mutexes in different orders leads to deadlock:

```
int weight, value;
std::mutex wt_mux, val_mux;

{                                                    // Thread 1
  std::lock_guard<std::mutex> wt_lock(wt_mux);       //   wt 1st
  std::lock_guard<std::mutex> val_lock(val_mux);     //   val 2nd

   work with weight and value                        // critical section
}
{                                                    // Thread 2
  std::lock_guard<std::mutex> val_lock(val_mux);     //   val 1st
  std::lock_guard<std::mutex> wt_lock(wt_mux);       //   wt 2nd

   work with weight and value                        // critical section
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Multiple Mutex Acquisition

std::lock solves this problem:

```
{                                                      // Thread 1
  std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);
  std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);

  std::lock(wt_lock, val_lock);                        // get mutexes w/o
                                                       // deadlock

  work with weight and value                           // critical section
}

{                                                      // Thread 2
  std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);
  std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);

  std::lock(val_lock, wt_lock);                        // get mutexes w/o
                                                       // deadlock

  work with weight and value                           // critical section

}
```


© 2011 Scott Meyers, all rights reserved.

How std::lock avoids deadlock is unspecified. It could canonically order the locks, use a back-off algorithm, etc.

If std::lock is called with a lock object that is already locked, an exception is thrown. If std::lock is called with mutex objects and one of the mutex objects is already locked, behavior may be undefined. (It depends on the details of the mutex type.)

© 2011 Scott Meyers, all rights reserved.

Condition Variables

Allow threads to communicate about changes to shared data.

- Consumers wait until producers notify about changed state.

Rules:

- Call wait while holding locked mutex.
- wait unlocks mutex, blocks thread, enqueues it for notification.
- At notification, thread is unblocked and moved to mutex queue.
  - ➡ "Notified threads awake and run with the mutex locked."

Condition variable types:

- condition_variable: wait on std::unique_lock<std::mutex>.
  - ➡ Most efficient, appropriate in most cases.
- condition_variable_any: wait on any mutex type.
  - ➡ Possibly less efficient, more flexible.

© 2011 Scott Meyers, all rights reserved.

In concept, condition variables simply make it possible for one thread to notify another when some event occurs, but the fact that condition variables are inheritly tied to mutexes suggests that shared data is always involved. Pure notification could be achieved via semaphores, but there are no semaphores in C++0x.

There are no examples of condition_variable_any in this course.

As noted in the mutex discussion, condition variables are neither copyable nor movable.

© 2011 Scott Meyers, all rights reserved.

Condition Variables

wait parameters:

- Mutex for shared data (required).
- Timeout (optional).
- Predicate that must be true for thread to continue (optional).
  - Allows library to handle spurious wakeups.
  - Often specified via lambda.

Notification options:

- notify_one waiting thread.
  - When all waiting threads will do and only one needed.
    - No guarantee that only one will be awakened.
- notify_all waiting threads.

© 2011 Scott Meyers, all rights reserved.

All threads waiting on a condition variable must specify the same mutex. In general, violations of this constraint can not be statically detected, so programs violating it will compile (and have undefined behavior).

The most common use case for notify_all seems to be after a producer adds multiple elements to a work queue, at which point multiple consumers can be awakened.

© 2011 Scott Meyers, all rights reserved.

waiting Examples

```
std::atomic<bool> readyFlag(false);
std::mutex m;
std::condition_variable cv;

{
  std::unique_lock<std::mutex> lock(m);

  while (!readyFlag)                        // loop for spurious wakeups
    cv.wait(lock);                          // wait for notification

  cv.wait(lock, []{ return readyFlag; });     // ditto, but library loops

  if (cv.wait_for(lock,                       // if (notification rcv'd
                  std::chrono::seconds(1),    // or timeout) and
                  []{ return readyFlag; })) { // predicate's true…
    …                                         // critical section
  }
  else {
    …                                         // timed out w/o getting
  }                                           // into critical section
}
```


© 2011 Scott Meyers, all rights reserved.

[std::atomic<bool> has not yet been introduced.]

The copy constructor in std::atomic<bool> is deleted, so direct initialization syntax or brace initialization syntax must be used; copy initialization won't compile.

Atomic types (e.g., std::atomic<bool>) are defined in <atomic>.

The waiting functions are wait, wait_for, and wait_until. The only difference between wait_for and wait_until is that the former takes a duration as a timeout (how long to wait), while the latter takes an absolute time (when to wait until). Waiting times are absolute (e.g., the example above will wait for a total of 1 second, regardless of how many spurious wakeups occur).

The examples on this page assume that readyFlag, m, and cv are nonlocal variables, e.g., at global or namespace scope. That's why the lambdas can refer to readyFlag without capturing it.

© 2011 Scott Meyers, all rights reserved.

Notification Examples

```
std::atomic<bool> readyFlag(false);            // as before
std::condition_variable cv;
{
  …                                        // make things "ready"
  readyFlag = true;
  cv.notify_one();                         // wake ~1 thread
}                                          // blocked on cv
{
  …                                        // make things "ready"
  readyFlag = true;
  cv.notify_all();                         // wake all threads
}                                          // blocked on cv (all but
                                           // 1 will then block on m)
```

notify_all moves all blocked threads from the condition variable queue to the corresponding mutex queue.

© 2011 Scott Meyers, all rights reserved.

The examples make no mention of a mutex, because notifiers need not hold a mutex in order to signal a condition.

© 2011 Scott Meyers, all rights reserved.

Thread-Local Data

Variables eligible for static storage duration may be thread_local.

- I.e., global/file/namespace-scoped vars; class-statics, file-statics.

```
thread_local std::string threadName;           // e.g., at global or
                                               // namespace scope

static thread_local
  std::chrono::microseconds timeUsed(0);       // e.g., at file scope

void f(int x)
{
  static thread_local unsigned timesCalled;
  …
}

class Widget {
private:
  thread_local static std::unordered_map<std::u16string, int> cache;
  …
};
```


© 2011 Scott Meyers, all rights reserved.

The threadName variable, for example, could be set by the function that the thread is started running in (i.e., that's passed to the std::thread constructor).

The standard does not require that unused thread-locals be constructed, so under good implementations, threads should pay for construction/destruction of only those thread-locals they use. This is a difference from global objects, which must be constructed/destructed unless the implementation can establish that they have no side effects.

© 2011 Scott Meyers, all rights reserved.

Thread-Local Data

Some details:

- thread_locals may be dynamically initialized.
  - Their constructors may be arbitrarily complex.
- thread_local may be combined with extern.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Other Concurrency Features

- Thread-safe initialization of objects of static storage duration.
- Thread-safe one-time function invocation via std::call_once and std::once_flag.
- Thread detachment when no join is needed.
- Separation of task setup and invocation via std::packaged_task.
- Support for mutex and lock UDTs via standard interfaces.
- Atomic types (e.g., std::atomic<int>) with memory ordering options.
- Operations on current thread, e.g., yield and sleep.
- Query number of hardware-supported threads.
- Library thread safety guarantees (e.g., for std::cin/std::cout, STL containers, std::shared_ptr, etc.)
- Many other features for threads, locks, condition variables, etc.,
  ➡ This was an *overview*.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.
Slide **124**

There is also a standard API for getting at the platform-specific handles behind threads, mutexes, condition variables, etc.. These handles are assumed to be the mechanism for setting thread priorities, setting stack sizes, etc. (Regarding setting stack sizes, Anthony Williams notes: "Of those OSs that support setting the stack size, they all do it differently. If you're coding for a specify platform (such that use of the native_handle would be OK), then you could use that platform's facilities to switch stacks. e.g. on POSIX you could use makecontext and swapcontext along with explicit allocation of a stack, and on Windows you could use Fibers. You could then use the platform-specific facilities (e.g. Linker flags) to set the default stack size to something really tiny, and then switch stacks to something bigger where necessary.“)

“UDT” = “User Defined Type”.

The best way to find C++0x’s library thread safety guarantees is to search draft standard chapters 17ff for “data race”. Relevant sections of N3290 are 17.6.5.9 (general rules), 18.6.1.4 (memory allocators), 23.2.2 and 21.4/3 (STL containers and string), and 27.4.1/4 (streams). Sometimes you have to read between the lines, e.g., 17.6.5.9/7 of N3290 is, I believe, the standard’s way of saying that reference count manipulations (e.g., in shared_ptr, promise, shared_future, etc.) must be thread-safe.

© 2011 Scott Meyers, all rights reserved.

Concurrency Support Summary

- Threads run callable objects, support joining and detaching.
  - ➡ Callers must avoid argument lifetime problems.
- std::async and futures support asynchronous calls.
- Mutexes may do timeouts or recursion; typical use is via locks.
  - ➡ std::lock_guard often suffices, std::unique_lock is more flexible.
- std::lock locks multiple mutexes w/o deadlock.
- Condition variables do timeouts, predicates, custom mutex types.
- Data eligible for static storage duration may be thread-local.
- Many concurrency support details aren't treated in this talk.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Summary of Features for Everybody

- “>>” at close of nested templates eliminates a syntactic pothole.
- auto variables have the type of their initializing expression.
- Range-based for loops ease iteration over containers, arrays, etc.
- nullptr avoids int/pointer confusion and aids perfect forwarding.
- Unicode string encodings support UTF-8, UCS-16, and UTF-32.
- Uniform initialization syntax and std::initializer_list makes brace initialization lists valid everywhere.
- Lambda expressions create function objects at their point of use.
- Template aliases allow “template typedefs” to be created.
- Concurrency support includes mutexes, locks, condition variables, thread-local data, asynchronous calls, and more.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

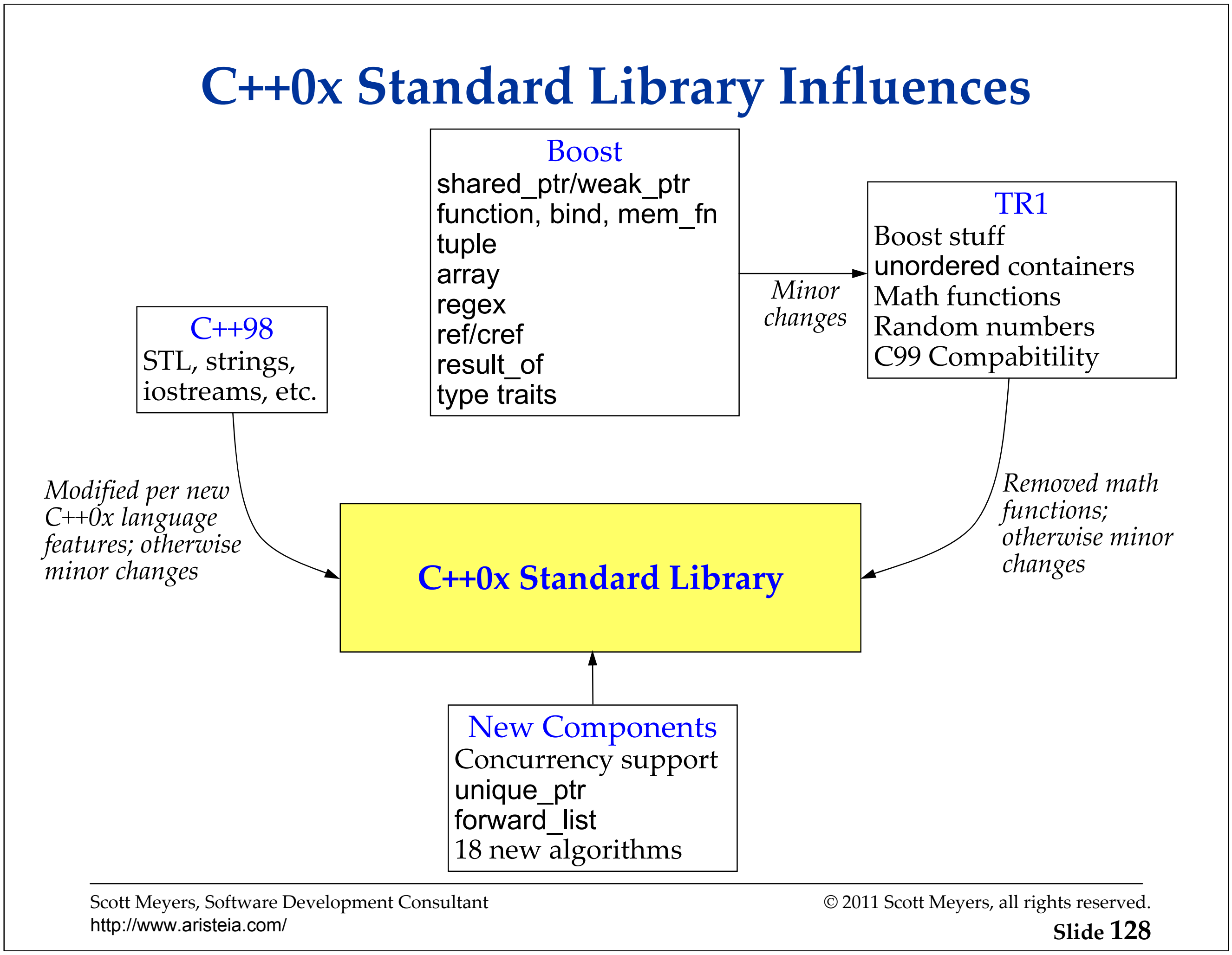
Overview

- Introduction
- Features for Everybody
- **Library Enhancements**
- Features for Class Authors
- Features for Library Authors
- Yet More Features
- Further Information

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.
Slide 127

In general, the material on library enhancements is terser than the rest of the material, because I assume many attendees will be familiar with the STL and possibly even TR1, hence there is less need to provide background information.

© 2011 Scott Meyers, all rights reserved.



Although the C++98 box is smallest, it had the strongest influence on the C++0x standard library.

© 2011 Scott Meyers, all rights reserved.

New Features for Standard Containers

General:

- Initializer list support.
- Move semantics support to avoid unnecessary copying.
- Improved const_iterator support:
  - ➡ cbegin/cend/crbegin/crend generate const_iterators/const_reverse_iterators.
  - ➡ const_iterators instead of iterators to specify locations.
- emplace/emplace_hint for copy/move-free in-place construction:

```
std::vector<Widget> vw;
…
vw.push_back(Widget(10, 20));      // create temp Widget, copy
                                   // (or move) it into vw,
                                   // destroy temp
vw.emplace_back(10, 20);           // create Widget in vw
                                   // using given ctor args
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 129

Emplacement operations can't be called with brace initialization lists, because brace initialization lists can't be perfect-forwarded.

© 2011 Scott Meyers, all rights reserved.

New Features for Standard Containers

Specific containers:

- vector::shrink_to_fit, deque::shrink_to_fit, string::shrink_to_fit
  - All *request* removal of unused capacity.
- vector::data member function (akin to string's).
- map::at member function that throws if key not present.
- set and multiset elements now officially immutable.
  - Originally agreed on in 2001…
  - Loopholes: mutable members, const_cast.
    - Mutations affect sort order ⇒ undefined behavior.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 130

Regarding vector::shrink_to_fit, N3290 says only that "shrink_to_fit is a non-binding request to reduce capacity() to size()." The description for string::shrink_to_fit is similar. Presumably one can make no assumptions about memory allocation, copying or moving of elements, exceptions, etc.

The motivation for deque::shrink_to_fit is that the array of block pointers can become arbitrarily large, depending on the maximum size of the deque over its lifetime. Details at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2795.html#850 .

© 2011 Scott Meyers, all rights reserved.

TR1

- Standard C++ Committee Library "Technical Report 1."
- Basis for most new library functionality in C++0x.
- Largely derived from Boost libraries.
- TR1 functionality in namespace std::tr1.
- C++0x TR1-derived functionality in std.
  - Not identical to that in TR1.
    - Uses new C++0x features.
    - Tweaks some APIs based on experience.
  - APIs mostly backwards-compatible

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

From TR1 to C++0x

Common C++0x enhancements:

- Variadic templates eliminate number-of-parameter restrictions.
- New container conventions adopted.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

TR1 Functionality

New Functionality	Summary
Reference Wrapper	Objects that act like references
Smart Pointers	Reference-counting smart pointers
Return Type Determination	Useful for template programming
Enhanced Member Pointer Adapter	2nd-generation mem_fun/mem_fun_ref
Enhanced Binder	2nd-generation bind1st/bind2nd
Generalized Functors	Generalization of function pointers
Type Traits	Compile-time type reflection
Random Numbers	Supports customizable distributions
Mathematical Special Functions	Laguerre polynomials, beta function, etc.
Tuples	Generalization of pair
Fixed Size Array	Like vector, but no dynamic allocation
Hash Tables	Hash table-based set/multiset/map/multimap
Regular Expressions	Generalized regex searches/replacements
C99 Compatibility	64-bit ints, <cstdint>, new format specs, etc.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 133

Libraries in blue are also in C++0x. Libraries in bold are covered in this course (to at least some degree).

Regarding random numbers, C supports only rand, which is expected to produce a uniform distribution. C++0x supports both *engines* and *distributions*. An engine produces a uniform distribution, while a distribution takes the result of an engine and produces an arbitrary distribution from it. C++0x specifies default versions for the engine and distributions, but it also allows for customized versions of both.

© 2011 Scott Meyers, all rights reserved.

From TR1 to C++0x

TR1 Functionality	C++0x Functionality Changes
Reference Wrapper	None.
Smart Pointers	Support for allocators and unique_ptr. Minor new functionality (details shortly).
Return Type Determination	Inherent C++98 restrictions lifted.
Enhanced Member Pointer Adapter	None.
Enhanced Binder	Inherent C++98 restrictions lifted.
Generalized Functors	Support for allocators. Added assign.
Type Traits	Inherent C++98 restrictions lifted. Some additions/renamings.
Random Numbers	Revised engines/distributions. Removal of variate_generator.
Mathematical Special Functions	Not in C++0x. (To be a separate standard.)
Tuples	Added tuple_cat.
Fixed Size Array	Renamed assign ⇒ fill.
Hash Tables	Support for operators == and !=.
Regular Expressions	String literals often okay (not just std::strings).
C99 Compatibility	fabs(complex<T>) ⇒ abs(complex<T>).

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 134

Of the 23 proposed mathematical special functions in TR1, 21 are preserved in the separate draft standard, "Extensions to the C++ Library to Support Mathematical Special Functions." The two missing functions are confluent hypergeometric functions and hypergeometric functions.

"Inherent C++98 restrictions lifted" means that restrictions inherent in library functionality based on C++98 were removed from the corresponding C++0x specification. From Stephan T. Lavavej: "In C++0x, result_of is powered by decltype and thus always gets the right answer without TR1's cumbersome and incomplete library machinery. Similarly, bind is powered by rvalue references, lifting its restriction on rvalues. Type traits are guaranteed to use compiler hooks and always get the right answers." Practically speaking, it means that many TR1 edge cases are no longer edge cases.

© 2011 Scott Meyers, all rights reserved.

From TR1: shared_ptr and weak_ptr

Motivation:

- Smart pointers simplify resource management.
  - E.g., prevention of leaks when exceptions are thrown.
- auto_ptr is constraining:
  - Designed for exclusive-ownership.
  - Has strange copy semantics.
    - No containers of auto_ptr.
- A standard shared-ownership smart pointer needed:
  - Should offer "normal" copy semantics.
    - Hence may be stored in containers.
  - Many versions have been created/deployed.
    - Typically based on reference counting.

© 2011 Scott Meyers, all rights reserved.

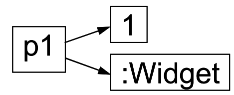
The "From TR1" in the title indicates that this is a C++0x feature based on TR1 functionality.

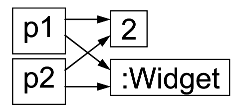
© 2011 Scott Meyers, all rights reserved.

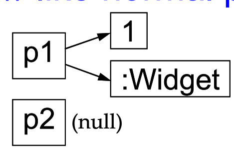
shared_ptr

- Declared in <memory>.
- A reference-counting smart pointer.
- Pointed-to resources are released when the ref. count (RC) → 0.

```
{
  std::shared_ptr<Widget> p1(new Widget);


  std::shared_ptr<Widget> p2(p1);
   …
  p1->doThis();                           // use p1 and p2
  if (p2) p2->doThat();                   // like normal ptrs
  …
  p2 = nullptr;
  …
}                                         // RC = 0; Widget
                                          // deleted
```







© 2011 Scott Meyers, all rights reserved.

p2 = nullptr; is essentially the same as p2.reset();.

"RC" = "Reference Count".

© 2011 Scott Meyers, all rights reserved.

shared_ptr Constructors

- Default, copy, from raw pointer.

```
std::shared_ptr<Widget> pw1;
std::shared_ptr<Widget> pw2(pw1);
std::shared_ptr<Widget> pw3(new Widget);       // typical use
```

  - Latter is explicit:

```
std::shared_ptr<Widget> pw4 = new Widget;  // error!
```

- From compatible unique_ptr, auto_ptr, shared_ptr, or weak_ptr.

```
std::unique_ptr<Widget> makeUP();              // factory funcs
std::auto_ptr<Widget> makeAP();
std::shared_ptr<Widget> pw5(makeUP());         // from unique_ptr
std::shared_ptr<Widget> pw6(makeAP());         // from auto_ptr
std::shared_ptr<const Widget> pw7(pw3);        // add const
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 137

[std::unique_ptr has not been introduced yet.]

"Compatible" pointer types takes into account derived-to-base conversions (e.g., shared_ptr<base> from shared_ptr<derived>.

Conversion from unique_- and auto_ptrs is supported only for sources that are non-const rvalues (as shown in the examples). Initializing a shared_ptr with an lvalue auto_- or unique_ptr requires use of std::move.

© 2011 Scott Meyers, all rights reserved.

shared_ptr Constructors

- From this:
  - It's a raw pointer, but other shared_ptrs might already exist!

```
std::shared_ptr<ISomething>
Widget::getISomething()
{                                                    // dangerous!
  return std::shared_ptr<ISomething>(this);          // could create a
}                                                    // new ref count!

std::shared_ptr<ISomething>
Widget::getISomething()
{                                                    // okay, no chance
  return shared_from_this();                         // of a new RC
}
```

  - Inheritance from enable_shared_from_this is required:

```
class Widget: public ISomething,
              public std::enable_shared_from_this<Widget> {
  …
};
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 138

"RC" = "Reference Count".

shared_from_this can't be used to create the first shared_ptr to an object.

Using shared_from_this in constructors, e.g., to register an object during its construction, is not reliable. A brief discussion of the problem can be found at http://www.boost.org/libs/smart_ptr/sp_techniques.html#in_constructor.

© 2011 Scott Meyers, all rights reserved.

Some shared_ptr Features

- Access to underlying raw pointer:
  - Useful for communicating with legacy APIs.

```
void oldAPI(Widget *pWidget);
std::shared_ptr<Widget> spw(new Widget);
oldAPI(spw.get());
```

- Access to reference count:

```
if (spw.unique()) …                        // always efficient
std::size_t refs = spw.use_count();        // may be inefficient
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Some shared_ptr Features

- Operators:
  - static_pointer_cast, dynamic_pointer_cast, const_pointer_cast

```
void someFunc(std::shared_ptr<Widget> spw)
{
  if (std::shared_ptr<Gadget> spg =
      std::dynamic_pointer_cast<Gadget>(spw)) {
  …                                    // spw really points to a Gadget
  }
}
```

  - Relationals: ==, !=, <
  - Output: <<

```
std::shared_ptr<Widget> spw;
…
std::cout << spw;
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 140

There is no reinterpret_pointer_cast for shared_ptrs. N1450 (the proposal document for adding shared_ptr to TR1, which is the precursor to shared_ptr in C++0x) says, "reinterpret_cast and const_cast equivalents have been omitted since they have never been requested by users (although it's possible to emulate a reinterpret_pointer_cast by using an intermediate shared_ptr<void> and a static_pointer_cast)." Both TR1 and C++0x include const_pointer_cast but lack reinterpret_pointer_cast, so presumably during standardization uses cases were found for the former but not for the latter.

© 2011 Scott Meyers, all rights reserved.

shared_ptr and Incomplete Types

Unlike auto_ptr (but like unique_ptr), shared_ptr supports incomplete types:

```
class Widget;                    // incomplete type
std::auto_ptr<Widget> ap;        // undefined behavior!
std::shared_ptr<Widget> sp;      // fine
std::unique_ptr<Widget> up;      // also fine
```

shared_ptr thus allows common coupling-reduction strategies.

- E.g., pimpl.

© 2011 Scott Meyers, all rights reserved.

In C++03, auto_ptr's undefined behavior when used with incomplete types is a fallout of 17.4.3.6/2, which says that instantiating any standard library template with an incomplete type yields undefined behavior. (The corresponding section in draft C++0x is [res.on.functions]/2 (17.6.4.8/2 in N3290).)

© 2011 Scott Meyers, all rights reserved.

shared_ptr and Inheritance Conversions

auto_ptr fails to support some inheritance-based conversions that shared_ptr offers:

```
class Base { … };
class Derived: public Base { … };

std::auto_ptr<Derived> produce();        // func. returning auto_ptr<Der>
void consume(std::auto_ptr<Base>);       // func. taking auto_ptr<Base>

consume(produce());                      // error!  won't compile


std::shared_ptr<Derived> produce();      // same code, but with
void consume(std::shared_ptr<Base>);     // shared_ptr

consume(produce());                      // fine
```

Note: the auto_ptr-based code (erroneously) compiles on some platforms.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Custom Deleters

By default, shared_ptrs use delete to release resources, but this can be overridden:

```
Widget* getWidget();                  // API to acquire/release a
void releaseWidget(Widget*);          // resource


{
  std::shared_ptr<Widget> pw(getWidget(), releaseWidget);
  …
}                                     // releaseWidget called
```

The default deleter is a function invoking delete.

- Out of the box, the cross-DLL delete problem goes away!

Deleters are really *releasers* (as above):

- E.g., a deleter could release a lock.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

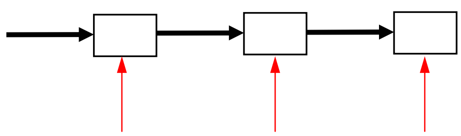
weak_ptr

weak_ptrs are like raw pointers, but they know when they dangle:

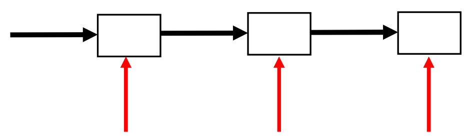
- When a resource's RC → 0, its weak_ptrs *expire.*
  - ➡ The shared_ptr releasing a resource expires all weak_ptrs:

```
std::shared_ptr<Widget> spw(new Widget);          // RC = 1
std::weak_ptr<Widget> wpw(spw);                   // RC remains 1
…
if (!wpw.expired()) …                             // if RC >= 1 …
```

- Useful for "observing" data structures managed by others.



Pointer observers – Risky



weak_ptr observers – Less Risky

© 2011 Scott Meyers, all rights reserved.

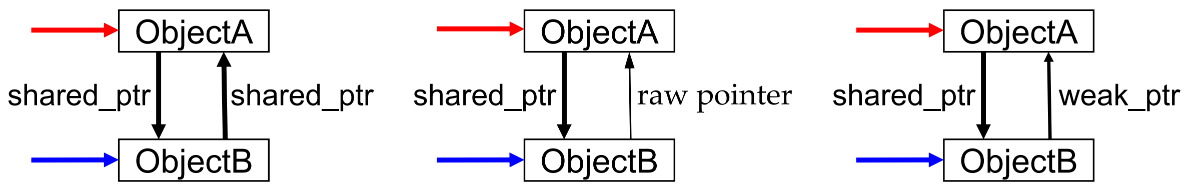
Calling expired may be faster than calling use_count, because use_count may not be constant-time. Calling unique is not an alternative, because unique does not exist for weak_ptrs.

"RC" = "Reference Count".

© 2011 Scott Meyers, all rights reserved.

weak_ptr

- Also to facilitate cyclic structures that would otherwise foil RC:
  - ➡ Consider reassigning the red pointer, then later the blue one.



RC-Unfriendly

RC-Friendly,
but Risky

RC-Friendly,
less Risky

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Using only shared_ptrs, we have an uncollectable cycle after both pointers are reassigned. Using a raw back pointer, ObjectB has no way to tell that its raw pointer dangles after the red pointer is assigned. (The blue pointer keeps ObjectB alive and referenceable.) Using a weak_ptr as a back pointer, ObjectB can detect if its back pointer dangles.

“RC” = “Reference Count”.

© 2011 Scott Meyers, all rights reserved.

weak_ptr

weak_ptrs aren't really smart pointers!

- No dereferencing operators (no operator-> or operator*).
- No implicit nullness test (conversion to something boolish).

To use a weak_ptr as a pointer, create a shared_ptr from it:

```
std::weak_ptr<Widget> wpw(spw);
wpw->doSomething();                         // risky and won't compile
if (!wpw.expired()) wpw->doSomething();     // won't compile

std::shared_ptr<Widget> pw1(wpw);           // create shared_ptr;
                                            // throws if wpw's expired
pw1->doSomething();                         // fine (if pw1 constructed)

std::shared_ptr<Widget> pw2(wpw.lock());    // pw2 is null if wpw's
                                            // expired
if (pw2) pw2->doSomething();                // fine
```

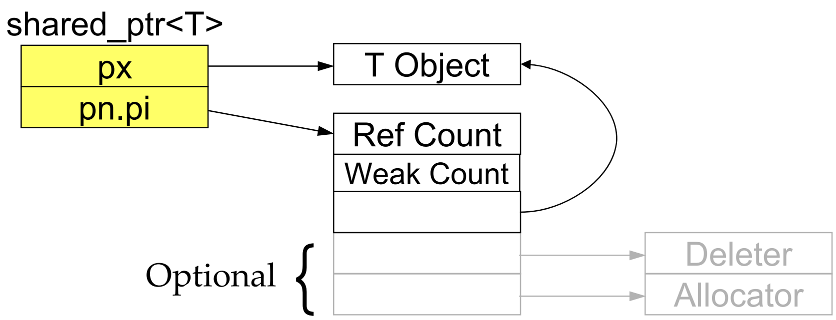

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Cost of shared_ptrs

Sample implementation (Boost 1.41):

- 2 words in size (pointer to object, pointer to RC).



- Uses dynamically allocated memory for the RC.
- Resource release (i.e., deletion) via a virtual function call ⇒ vtbls.
- Incurs cost for weak_ptr count even if no weak_ptrs are used.

© 2011 Scott Meyers, all rights reserved.

"RC" = "Reference Count".

The Boost implementaton allocates space for a custom deleter or a custom allocator only if the smart pointer is constructed with them. If the default deleter/allocator is used, no memory is used to store pointers to them.

The weak count keeps track of how many weak pointers exist for the object. When the RC becomes 0, the object itself is destroyed, but the RC block continues to exist until the weak count becomes 0. Weak pointers can tell whether they have expired by checking to see if the RC == 0. If so, they have.

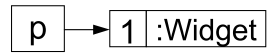
Memory allocation for the RC is avoided if std::make_shared (discussed on next page) is used.

Both px and the object pointer in *pn.pi point to the RC'd object, but the pointer values may be different. From N1450 (the proposal to add smart pointers to TR1): "The original pointer passed at construction time needs to be remembered for shared_ptr<X> to be able to correctly destroy the object when X is incomplete or void, ~X is inaccessible, or ~X is not virtual."

© 2011 Scott Meyers, all rights reserved.

From TR1 to C++0x

- make_shared<T> and allocate_shared<T> allocate object and RC with one allocation:




```
auto p =
  std::make_shared<Widget>(Widget ctor args);
```

```
class MyAllocator { … };
MyAllocator a;
```

```
auto p = std::allocate_shared<Widget>(a, Widget ctor args);
```

- Supports two "p1 and p2 point to same object" semantics:
  - **Value**: p1.get() == p2.get()
  - **Ownership**: p1 and p2 affect the RC of the same object
    - Good for shared_ptr<void>s pointing to MI-based types.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Use of make_shared/allocate_shared precludes specification of custom deleters, because there would be no way to differentiate those parameters from those for the object (e.g., Widget) constructor.

Operators == and < on shared_ptrs use value "points to the same object" semantics. Ownership semantics are available via std::shared_ptr::owner_before.

"RC" = "Reference Count".

© 2011 Scott Meyers, all rights reserved.

TR1-Derived Smart Pointers Summary

- shared_ptrs use reference counting to manage resource lifetimes.
- They support incomplete types, inheritance-based conversions, custom deleters, and C++-style casts.
- weak_ptrs can detect dangling pointers and help break cycles.
- shared_ptrs bigger/slower than built-in pointers.
- make_shared and allocate_shared avoid dedicated memory allocations for reference counts.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

unique_ptr

Successor to auto_ptr (which C++0x deprecates).

- Declared in <memory>.
- Like auto_ptr, supports *moving* values instead of *copying*.
  - But avoids "copy syntax that really moves."
- More general than auto_ptr:
  - Safe in containers and arrays.
  - Supports inheritance conversions and custom deleters.
  - May point to arrays.
- More efficient than shared_ptr for factory function returns.
- May be larger than auto_ptr.
  - gcc 4.5: auto_ptr holds 1 ptr, unique_ptr holds 2.
  - VC10: both typically hold 1.

© 2011 Scott Meyers, all rights reserved.

Unlike VC10, gcc 4.5 stores data in a unique_ptr for a deleter, even if the default deleter is being used. This is why gcc's unique_ptr is bigger than an auto_ptr. The comment about VC10 "typically" holding only 1 pointer is based on the assumption that "typical" use involves no custom deleter.

© 2011 Scott Meyers, all rights reserved.

unique_ptr

```
class Base {
public:
  virtual ~Base();                  // so polymorphic deletes work
  virtual void doIt();              // some virtual
};
class Derived: public Base {
public:
  virtual void doIt();              // overridden virtual function
};
...
{
  std::unique_ptr<Derived> pd(new Derived);
  std::unique_ptr<Base> pb(pd);     // error! can't copy lvalue

  std::unique_ptr<Base>             // ownership xfer; note
    pb(std::move(pd));              // Derived ⇒ Base
  …                                 // conversion

  pb->doIt();                       // calls Derived::doIt
  …
}                                   // delete pb.get()
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide **151**

[This slide mentions lvalues for the first time.]

© 2011 Scott Meyers, all rights reserved.

unique_ptr

```
using UPS = std::unique_ptr<std::string>;           // same as typedef
std::vector<UPS> vsp;                                // fine, container of
                                                     // unique_ptrs

for (int i = 0; i < n; ++i) {                        // move "copied"
  vsp.push_back(UPS(new std::string("Hello")));      // rvalue unique_ptrs
}                                                    // into vsp
…
std::sort(vsp.begin(), vsp.end(),                    // fine due to library
          [](const UPS& p1, const UPS&  p2)          // guarantee that sort
          { return *p1 < *p2; });                    // moves (not copies)
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.

[This slide mentions rvalues for the first time.]

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

unique_ptr and Custom Deleters

Unlike shared_ptr, deleters for unique_ptr are part of its type.

```
Widget* getWidgetFromPool();                    // Widget Pool API
void returnWidgetToPool(const Widget*);

std::unique_ptr<Widget, void(*)(const Widget*)>
getWidget()                                     // factory function
{
  return { getWidgetFromPool(), returnWidgetToPool };
}

...

{
  auto pw = getWidget();
  …
}                                   // invoke
                                    // returnWidgetToPool(pw.get())
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

unique_ptr and Arrays

Unlike shared_ptr, unique_ptr may point to an array.

- Its behavior then correspondingly modified:
  - No inheritance conversions.
  - No dereferencing (* or ->) operations.
  - Indexing added.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

unique_ptr and Arrays

```
class Base { … };

class Derived: public Base {
public:
  void doIt();
  …
};

std::unique_ptr<Derived[]> upda1(new Derived[10]);
…
std::unique_ptr<Derived[]> upda2;

upda2 = upda1;                            // error!  lvalue unique_ptr
                                          // not copyable

upda2 = std::move(upda1);                 // okay (upda1 now null)

std::unique_ptr<Base[]> upba =            // error! no inheritance
  std::move(upda2);                       // conversions with arrays

upda2->doIt();                            // error! no op-> or op*

for (int i = 0; i < 10; ++i) upda2[i].doIt();   // okay
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

unique_ptr vs. shared_ptr

Already noted:

- Deleter type part of unique_ptr type, not shared_ptr type.
- unique_ptr supports arrays, shared_ptr doesn't.
- Both support incomplete types.

In addition:

- shared_ptr supports static_pointer_cast, const_pointer_cast, dynamic_pointer_cast; unique_ptr doesn't.
- No unique_ptr analogue to make_shared/allocate_shared.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.
Slide 156

unique_ptr's support for incomplete types has one caveat. Given a std::unique_ptr<T> p, the type T must be complete at the point where p's destructor is invoked. Violation of this constraint requires a diagnostic, i.e., code failing to fulfill it will typically not compile. This constraint applies only to unique_ptrs using the default deleter; unique_ptrs using custom deleters are not so constrained.

© 2011 Scott Meyers, all rights reserved.

unique_ptr Summary

- Small/fast smart pointer for unique ownership; replaces auto_ptr.
- Safe for use in containers/arrays.
- Supports custom deleters and arrays.
- API "different" from shared_ptr API.

© 2011 Scott Meyers, all rights reserved.

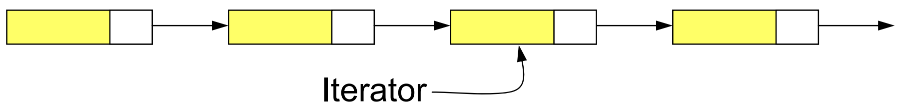
unique_ptr and shared_ptr do different things, so their APIs can't be the same, but in some cases they are different for no apparent reason.

© 2011 Scott Meyers, all rights reserved.

forward_list

A singly-linked list.

- Declared in <forward_list>
- Goal: zero time/space overhead compared to hand-written C.
- STL container conventions sacrificed to achieve goal:
  - insert_after/emplace_after/erase_after instead of insert/emplace/erase.
    - Normal "insert/emplace/erase before" behavior costly.



  - before_begin returns iterator preceding *begin.
    - Needed to insert/emplace/erase at front of list.
  - No size or push_back.
    - Store size or end (footprint) or run in O(n) time (surprising)?
- Offers only forward iterators.

© 2011 Scott Meyers, all rights reserved.

Having iterators point to the node prior to the one they reference would allow for an interface that was more like the rest of the STL, but at the cost of additional indirection per dereference, something contrary to the goal of as-good-as-hand-written-C performance.

Iterator invalidation rules for forward_list are essentially the same as for list: insertions invalidate nothing, erasures invalidate only iterators to erased elements.

© 2011 Scott Meyers, all rights reserved.

forward_list

Example:

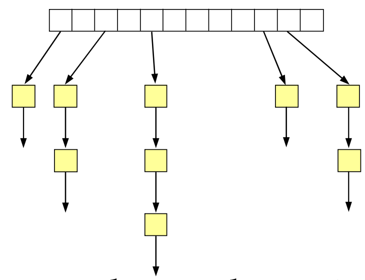
```
#include <forward_list>
#include <algorithm>

std::forward_list<int> fli { 1, 2, 3, 4, 5 };

auto it = std::find(fli.cbegin(), fli.cend(), 3);

fli.erase(it);                        // error!  no erase

--it;                                 // error!  forward iterator

fli.erase_after(it);                  // fli = 1, 2, 3, 5

fli.push_back(10);                    // error!  no push_back

fli.push_front(0);                    // fli = 0, 1, 2, 3, 5

fli.erase_after(fli.before_begin());  // fli = 1, 2, 3, 5
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

From TR1: Hash Tables

- Declared in <unordered_set> and <unordered_map>.
  - Default hashing functionality declared in <functional>.
- Designed not to conflict with pre-TR1/C++0x implementations.
  - I.e., hash_set, hash_map, hash_multiset, hash_multimap.
    - Interfaces vary – hence the need for standardization.
    - Standard names are unordered_set, unordered_map, etc.
  - Compatible with hash_* interfaces where possible.
- Each bucket has its own chain of elements:



Conceptual diagram! Implementations vary!

- Bucket count can change dynamically.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Containers' Characteristics

- Usual members exist:
  - iterator/const_iterator and other typedefs.
  - begin/end/cbegin/cend, insert/erase, size, swap, etc.
- Also 3 associative container functions: find, count, equal_range.
  - lower_bound/upper_bound are absent.
- unordered_map/unordered_multimap offer operator[] and at.
- Most relationals *not* supported: no <, <=, >=, >
  - Indeterminate ordering makes these too expensive.
  - == and != do exist: result based on *content*, not ordering.
    - Expected complexity O(n); worse-case is $O(n^2)$.
- Only forward iteration is provided.
  - No reverse_iterators, no rbegin/rend/crbegin/crend.

© 2011 Scott Meyers, all rights reserved.

When equal_range finds no elements, it returns (container.end(), container.end()). This makes it a bit easier to swallow the failure to include upper_- and lower_bound in the containers' interfaces.

© 2011 Scott Meyers, all rights reserved.

Hash Table Parameters

Hashing and equality-checking types are template parameters:

```
template<class Value,
         class Hash = std::hash<Value>,
         class Pred = std::equal_to<Value>,
         class Alloc = std::allocator<Value>>
class unordered_set { … };

template<class Key,
         class T,
         class Hash = std::hash<Key>,
         class Pred = std::equal_to<Key>,
         class Alloc = std::allocator<std::pair<const Key, T>>>
class unordered_map { … };
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Hashing Functions

Defaults are provided for built-in, string, and smart pointer types:

```
class Widget { … };

std::unordered_set<int> si;                    // all use default hash
std::unordered_multiset<double> md;            // func for shown types
std::unordered_map<std::wstring, int>mwi;
std::unordered_multimap<Widget*, std::string> mm1;
std::unordered_multimap<std::unique_ptr<Widget>, std::string> mm2;
```

Also for these less commonly used types:

- std::vector<bool>
- std::bitset
- std::thread::id
- std::error_code
- std::type_index

© 2011 Scott Meyers, all rights reserved.

Keys in associative containers (both ordered and unordered) are immutable; modifying elements in an associative container yields undefined behavior. Changing a key could affect the sort order (for sorted containers) or the hashed location (for unordered containers).

In [syserr.errcode.overview] (19.5.2.1/1 of N3290), draft C++0x describes std::error_code this way: "The class error_code describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. ... Class error_code is an adjunct to error reporting by exception."

std::type_index is a wrapper for std::type_info objects that's designed for storage in associative containers (ordered or unordered).

© 2011 Scott Meyers, all rights reserved.

Hashing Functions

To override a default or hash a UDT, specialize hash<T> or create a custom functor:

```
template<>
struct std::hash<Widget>: public std::unary_function<Widget,
                                                     std::size_t> {
  std::size_t operator()(const Widget& w) const { … };
};
std::unordered_set<Widget> sw;


struct IntHasher: public std::unary_function<int, std::size_t> {
  std::size_t operator()(int i) const { … };
};
std::unordered_map<int, std::string, IntHasher> mis;
```


© 2011 Scott Meyers, all rights reserved.

"UDT" = "User Defined Type".

Function pointers can also be used as hashing objects, but only the function pointer type would be specified as part of the type of the container. To actually use a function for hashing, the container would have to be constructed with a pointer to the specific hashing function.

© 2011 Scott Meyers, all rights reserved.

Operations for Bucket Count and Load Factor

Constructors allow a floor on bucket count (B) to be specified:

```
std::unordered_set<int> s1;          // B chosen by implementation
std::unordered_set<int> s2(53);      // B >= 53.  (Other ctor forms
                                     // support bucket floor, too.)
```

A table's *load factor (z)* is the average number of elements/bucket:

- z = *container*.size()/B.
- z can be queried, and a ceiling for it can be "hinted" (requested):

```
float z = s1.load_factor();          // get current load factor
s1.max_load_factor(0.75f);           // request ceiling for z;
                                     // future insertions may
                                     // increase B so that z <= .75,
                                     // then rehash s to use new B
float z_max = s1.max_load_factor();  // get current z_max (defaults to 1)
```

Because max_load_factor(*z*) is only a request, it's possible that *container*.load_factor() > *container*.max_load_factor().

© 2011 Scott Meyers, all rights reserved.

The variables z and z_max on this page could use auto in their declarations, but I'm using float to show that that's the precision used for load factors.

Empty buckets are included in an unordered_* container's load factor calculation.

© 2011 Scott Meyers, all rights reserved.

Rehashing

Explicit rehashing can also change the bucket count and load factor.

- Specify number of desired buckets via rehash.
- Specify number of expected elements via reserve.

```
std::unordered_set<int> s;
...
auto newB = computeNewB();
s.rehash(newB);          // reorganize s so that B >= newB
                         // and s.size()/B <= z_max
...
auto expElems = computeExpectedElements();
s.reserve(expElems);     // reorganize s so that expElems/B <= z_max
```

Rehashing (implicitly or explicitly) invalidates iterators.

- But not pointers or references.

© 2011 Scott Meyers, all rights reserved.

From what I can tell from the iterator invalidation rules, rehashing can happen only when insert or rehash is called.

For multi containers, rehashing preserves the relative order of equivalent elements.

© 2011 Scott Meyers, all rights reserved.

Iterating Over Bucket Contents

Useful for e.g., monitoring performance of hashing functions.

```
std::unordered_set<std::string> s;
…
auto numBuckets = s.bucket_count();            // # buckets
for (std::size_t b = 0; b < numBuckets; ++b) {
  std::cout << "Bucket " << b << " has "
            << s.bucket_size(b)                // # elems in bucket b
            << " elements: ";
  std::copy(
    s.cbegin(b), s.cend(b),                    // iters for bucket b
    std::ostream_iterator<std::string>(std::cout, " ")
  );
  std::cout << '\n';
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Hash Tables Summary

- Unordered containers based on hash tables with open hashing.
- Only forward iteration is supported.
- Maximum load factor can be dynamically altered.
- There is support for iterating over individual buckets.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

From TR1: Tuples

Motivation:

- pair should be generalized.
- Tuple utility demonstrated by other languages.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

TR1 Tuples

- tuple declared in <tuple>, helper templates in <utility>.
- Offers fixed-size heterogeneous "containers:"
  - Fixed-size ⇒ no dynamic memory ⇒ no allocator.

```
class Name { ... };
class Address { … };
class Date { … };

std::tuple<Name, Address, Date>                 // function to return
employeeInfo(unsigned employeeID);              // employee name,
…                                               // address, hire date

unsigned eid;
…
std::tuple<Name, Address, Date>                 // initialize tuple with
  info(employeeInfo(eid));                      // value from
                                                // employeeInfo
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 170

"Containers" is in quotes, because tuple doesn't, in general, adhere to the container interface.

© 2011 Scott Meyers, all rights reserved.

get

Tuple elements are accessed via get:

- Takes a compile-time index; indices start at 0:

```
Name empName(std::get<0>(info));
Address empAddr(std::get<1>(info));
Date empHDate(std::get<2>(info));
```

- A compile-time index!
  - get is a template, and the index is a *template argument*.

```
int nameIdx = 0;

Name empName(std::get<nameIdx>(info));          // error!
```

  - for/do/while loops over tuple contents aren't possible.

© 2011 Scott Meyers, all rights reserved.

TMP can be used to generate code to iterate over the contents of a tuple.

© 2011 Scott Meyers, all rights reserved.

get

Using named indices makes for more readable code:

```
enum { EmpName, EmpAddr, EmpHireDate };

Name empName(std::get<EmpName>(info));
Address empAddr(std::get<EmpAddr>(info));
Date empHDate(std::get<EmpHireDate>(info));
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

tie

tie can perform the work of multiple gets:

```
std::tie(empName, empAddr, empHDate) =     // assign to all 3
  employeeInfo(eid);                        // variables
```

ignore can be used within tie to get only selected elements:

```
std::tie(empName, empAddr, std::ignore) =  // assign only name
  employeeInfo(eid);                        // and address

std::tie(std::ignore,                       // assign address only.
         empAddr,                           // (Here, using get
         std::ignore) = employeeInfo(eid);  // would be easier.)
```


© 2011 Scott Meyers, all rights reserved.

std::tie can be used with std::pair objects, because std::tie returns a tuple, and std::tuple has a constructor that takes a std::pair:

```
std::pair<Name, Address> empNameAddr(unsigned employeeID);
std::tie(empName, empAddr) = empNameAddr(eid);
```


© 2011 Scott Meyers, all rights reserved.

make_tuple

A generalization of make_pair:

```
class Employee {
public:
  Name name() const;
  Address address() const;
  Date hireDate() const;
  …
};
Employee findByID(unsigned eid);


std::tuple<Name, Address, Date>
employeeInfo(unsigned employeeID)
{
  Employee e(findByID(employeeID));
  return std::make_tuple(e.name(), e.address(), e.hireDate());
}
```


© 2011 Scott Meyers, all rights reserved.

The final return statement in the example can't be written as

```
return { e.name(), e.address(), e.hireDate() };
```

because the relevant tuple constructors are either explicit (hence not usable here) or are templates (also not usable here, because templates can't deduce a type for brace initialization lists).

© 2011 Scott Meyers, all rights reserved.

Reflection

There's support for compile-time reflection:

```
template<typename Tuple>
void someFunc(Tuple t)
{
  std::size_t numElems =                              // # elems
    std::tuple_size<Tuple>::value;                    // in Tuple

  typedef
    typename std::tuple_element<0, Tuple>::type       // type of
    FirstType;                                        // 1st elem
  …
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Other tuple Functionality

The usual STL container relationals (<, <=, ==, !=, >=, >):

- == and != tests use elementwise ==
- Other relational tests are lexicographical using only <:
  - Values are considered equal if they're equivalent (based on <)

pair<T1, T2> can often be used as a tuple<T1, T2>:

- A 2-element tuple can be created or assigned from a compatible pair.
- get<0> and get<1> both work on pairs.
- So do tuple_size and tuple_element.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Tuples Summary

- Tuples are a generalization of `std::pair`.
- Element access is via compile-time index using `get` or via `tie`.
- Compile-time reflection is supported. It works on `std::pair`s, too.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

From TR1: Fixed-Size Arrays

Motivation:

- Built-in arrays aren't STL containers:
  - ➡ No `begin`, `end`, etc.
  - ➡ They don't know their size.
  - ➡ They decay into pointers.
- `vector` imposes overhead:
  - ➡ Dynamic memory allocation.
- Need an STL container with performance of built-in arrays.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Fixed-Size Arrays

- Declared in <array>.
- Offers conventional members:
  - iterator/const_iterator/reverse_iterator and other typedefs
  - begin/end/cbegin/cend, empty, swap, relational operators, etc.
    - But swap runs in *linear* (not constant) time.
- Also vectoresque members: operator[], at, front, back
- Contents layout-compatible with C arrays (and vector).
  - Get a pointer to elements via data (as with vector and string):

```
std::array<int, 5> arr;              // create array
…
int *pElements = arr.data();         // get pointer to elements
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 179

For std::array objects with a size of 0, results of invoking data are “unspecified.”

© 2011 Scott Meyers, all rights reserved.

Fixed-Size Arrays

Because arrays are fixed-size,

- No insert, push_back, erase, clear, etc.
- No dynamic memory allocation.
  - ➡ Hence no allocator.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

arrays are Aggregates

A array is an *aggregate*, so:

- No initializer for built-in element types ⇒ default initialization.
  - For stack or heap arrays ⇒ "random values":

```
std::array<int, 5> arr1;      // if arr1 on stack or heap,
                              // element values undefined
```

  - For arrays with static storage duration ⇒ zeros:

```
std::array<int, 5> arr2;      // if arr2 in static storage,
                              // element values are zero
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

arrays are Aggregates

- Too few initializers ⇒ remaining objects are *value-initialized*:
  - Built-in types initialized to 0.
  - UDTs with constructors are default-constructed.
  - UDTs without constructors: members are value-initialized.

```
std::array<short, 5> arr3 { 1, 2, 3, 4, 5 };
std::array<int, 5> arr4 {10, arr1[3], 30 };     // last 2 values
                                                // init'd to 0
std::array<float, 1> arr5 { 1, 2, 3, 4, 5 };    // error!  won't
                                                // compile
```

- Too many initializers ⇒ error.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 182

Value initialization is defined in [dcl.init] (8.5/7 of N3290).

“UDT” = “User Defined Type”.

© 2011 Scott Meyers, all rights reserved.

arrays are Aggregates

For built-in types, no initializer ≠ too few initializer values:

```
std::array<int, 5> arr1;          // no initializer:
                                  // - on stack or heap ⇒ random values
                                  // - static storage ⇒ all zeros
std::array<int, 5> arr2 {};       // too few initializers ⇒ use zeros
std::array<int, 5> arr3 = {};     // too few initializers ⇒ use zeros
```

Types with constructors always have constructors called:

```
class Widget {
public:
  explicit Widget(int = -1);
  ...
};

std::array<Widget, 5> arr4;        // construct all Widgets from -1

std::array<Widget, 5> arr5 {};     // construct all Widgets from 0
```

All behavior above is same as for built-in arrays.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 183

The aggregate initialization rules for std::arrays are the same as for built-in arrays.

© 2011 Scott Meyers, all rights reserved.

arrays are Aggregates

Because array is an aggregate:

- All members are public!
- Only default, copy, and move construction is supported.
  - These constructors are compiler-generated.
  - Range construction is unavailable:

```
std::vector<int> v;
…
std::array<int, 10>                    // error!  array supports only
  arr(v.begin(), v.begin()+10);        // default and copy construction
```


© 2011 Scott Meyers, all rights reserved.

Technically, aggregates may have non-public members that are static.

© 2011 Scott Meyers, all rights reserved.

arrays as Tuples

array<T, n> can sometimes be treated like tuple<T, T, …, T>:

```
class Widget { … };
const std::size_t arraySz = 10;
typedef std::array<Widget, arraySz> WidgetArray;

WidgetArray arr;                    // 10 Widgets default-constructed

…

std::size_t numElements = std::tuple_size<WidgetArray>::value;

std::size_t elemSize =
  sizeof(std::tuple_element<0, WidgetArray>::type);

const auto& value = std::get<0>(arr);

std::cout << "arr has " << numElements
          << " elements, each of size " << elemSize
          << ".  The first element's value is " << value
          << '\n';
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

array vs. vector

- array is fixed-size, vector is dynamically sized.
- array uses no dynamic memory, vector does.
- array::swap is linear-time and may throw, vector::swap is constant-time and can't throw.
- array can be treated like a tuple, vector can't.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

array vs. C Arrays

- array objects know their size, C arrays don't
- array allows 0 elements, C arrays don't
- array requires an explicit size, C arrays can deduce it from their initializer
- array supports assignment, C arrays don't
- array can be treated like a tuple, C arrays can't

Given array, vector, and string, there is little reason to use C-style arrays any longer.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Fixed-Size Arrays Summary

- array objects are STLified C arrays.
- They support brace-initialization, but not range initialization.
- They support some tuple operations.
- Given array, std::vector, and std::string, there is little reason to use C-style arrays.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

From TR1: Regular Expressions

Motivation:

- Regular expression (RE) functionality is widely useful.
- Many programming languages and tools support it.
- C RE libraries support only char*-based strings.
  - ➡ C++ should support wchar_t* strings and string objects, too.

© 2011 Scott Meyers, all rights reserved.

Conceptually, C++0x regex support works not just with std::string, but with all std::basic_string instantiations (e.g., std::wstring, std::u16string, std::u32string). However, library specializations and overloads exist only for strings based on char*, wchar_t*, std::string, and std::wstring. How difficult it would be to use the library's regex components with other string types, I don't know.

© 2011 Scott Meyers, all rights reserved.

TR1 Regular Expressions

- Declared in <regex>.
- RE objects modeled on string objects:
  - Support char, wchar_t, Unicode encodings, locales.
- RE syntax defaults to modified ECMAScript.

```
std::regex capStartRegex("[A-Z][[:alnum:]]*");      // alnum substr.
                                                    // starting with a
                                                    // capital letter
std::regex SSNRegex(R"(\d{3}-\d{2}-\d{4})");        // looks like a SSN
                                                    // (ddd-dd-dddd)
```

  - Alternatives: POSIX Basic, POSIX Extended, awk, grep, egrep.
  - Raw string literals very useful in RE specifications.
- Offers control over state machine behavior:

```
std::regex filenameRegex(               // regex for some
  R"(\w+\.((txt)|(dat)|(log)))",        // .txt, .dat, and .log files;
  std::regex::icase |                   // ignore case during search;
  std::regex::optimize                  // match speed more important
);                                      // than regex ctor speed
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 190

ECMAScript is essentially a standardized version of Perl RE syntax.

"SSN" is short for "Social Security Number", which is a government-issued ID number in the USA.

\w means word characters (i.e., letters, digits, and underscores).

Regarding the optimize flag, Pete Becker's *The C++ Standard Library Extensions* (see end of notes for full reference) says: "This optimization typically means converting a nondeterministic FSA into a deterministic FSA. There are well-understood algorithms for doing this. Unfortunately, this conversion can sometimes be very complex; hence, very slow. So don't ask for it if you don't need it."

© 2011 Scott Meyers, all rights reserved.

Fundamental Functionality

- regex_match: Does the RE match the complete string?
- regex_search: Does the RE occur in the string?
- regex_replace: Replace text matching RE with other text.
  - Replacement isn't in-place: new text is returned.

Matches are held in match_results objects. Iteration is supported:

- regex_iterator: Iterate over matches for a string.
- regex_token_iterator: Iterate over matches and match subfields.

These are templates. You normally use named instantiations:

- For strings: smatch/sregex_iterator/sregex_token_iterator
- For wstrings: wsmatch/wsregex_iterator/wsregex_token_iterator
- For char*s: cmatch/cregex_iterator/cregex_token_iterator
- For wchar_t*s: wcmatch/wcregex_iterator/wcregex_token_iterator

© 2011 Scott Meyers, all rights reserved.

regex_replace can be configured with flags to (1) replace only the first match and/or to (2) not write out unmatched text, but by default, it behaves as summarized in these slides. I'm not familiar with use cases for these options.

Regex iterators iterate only over nonoverlapping matches. Iteration over overlapping matches must be done manually and must take into account the issues described in Becker's book (mentioned on a subsequent slide).

I don't know why there are no typedefs for char16_t- and char32_t-based types (e.g., char16_t*s and u16strings).

© 2011 Scott Meyers, all rights reserved.

Examples: regex_match, regex_search

Does text look like an SSN?

```
const std::regex SSNRegex(R"(\d{3}-\d{2}-\d{4})");

bool looksLikeSSN(const std::string& text)
{
  return std::regex_match(text, SSNRegex);
}
```

Does text contain a substring that looks like an SSN?

```
bool mayContainSSN(const std::string& text)
{
  return std::regex_search(text, SSNRegex);
}
```


© 2011 Scott Meyers, all rights reserved.

Information on matches found can be retrieved through an optional match_results parameter. The next slide gives an example.

© 2011 Scott Meyers, all rights reserved.

Example: regex_search

Collect all (non-overlapping) substrings that look like SSNs:

```
void possibleSSNs1(const std::string& text, std::list<std::string>& results)
{
  auto b(text.cbegin()), e(text.cend());
  std::smatch match;

  while (std::regex_search(b, e, match, SSNRegex)) {
    results.push_back(match.str());
    b = match[0].second;
  }
}
```

This works, but iterative calls to regex_search are suspect:

- REs allowing empty matches can cause infinite loops.
- REs with ^ and \b specifiers problematic after first iteration.

Details in chapter 19 of Becker's *The C++ Standard Library Extensions*.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 193

An empty match is one matching no text, e.g., the regex "(abc)*"can match zero characters, because "*" means "zero or more."

\b is the beginning-of-word specifier.

This loop finds only nonoverlapping matches. To allow overlapping matches, change b's assigment to

```
b = ++match[0].first;
```


© 2011 Scott Meyers, all rights reserved.

Example: regex_iterator

A better approach:

```
void possibleSSNs2(const std::string& text, std::list<std::string>& results)
{
  std::sregex_iterator b(text.cbegin(), text.cend(), SSNRegex);
  std::sregex_iterator e;

  for (auto it = b; it != e; ++it) {
    results.push_back(it->str());
  }
}
```

regex_iterator (and regex_token_iterator) handle tricky cases.

- Use them instead of loops over regex_search calls.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Example: regex_replace

Replace all substrings that look like SSNs with dashes:

```
void dashifySSNs(std::string& text)
{
  const std::string dashes("-----------");
  text = std::regex_replace(text, SSNRegex, dashes);
}
int main()
{
  std::string data("123-45-6789x777-77-7777abc");
  std::cout << data;          // 123-45-6789x777-77-7777abc
  dashifySSNs(data);
  std::cout << data;          // -----------x-----------abc
}
```


© 2011 Scott Meyers, all rights reserved.

Note that the assignment to text in dashifySSNs is a move assignment.

There is no conditional replacement, i.e., no "regex_replace_if". If you don't want a global substitution of the regex or a replacement for only the first match, you have to iterate from match to match and construct the modified string yourself.

© 2011 Scott Meyers, all rights reserved.

Capture Groups

Count (non-overlapping) word repetitions in a string (e.g., "the the"):

```
std::size_t repWords( std::string::const_iterator b,
                      std::string::const_iterator e)
{
  std::regex wordRepeatRgx(R"(\b)"               // word boundary
                           R"(([A-Za-z_]\w*))"   // word
                           R"(\s+)"              // whitespace
                           R"(\1)"               // same word text
                           R"(\b)"               // word boundary
                          );
  std::size_t repCount = 0;

  for (std::sregex_iterator i(b, e, wordRepeatRgx), end;
       i != end;
       ++i) {
    ++repCount;
  }

  return repCount;
}
```


© 2011 Scott Meyers, all rights reserved.

With this regex, the repeated word must begin with a letter or an underbar. This avoids matching repeated numbers, which we'd get if we just used "\w\w*".

With std::regex_replace, capture groups can be referred to in the replacement pattern. For a nice example, consult Marius Bancila's article in the Further Information section.

© 2011 Scott Meyers, all rights reserved.

Capture Groups

Alternative (for loop-haters and lambda-lovers):

```
std::size_t repWords(std::string::const_iterator b,
                     std::string::const_iterator e)
{
  // same regex as before
  std::regex wordRepeatRgx(R"(\b([A-Za-z_]\w*)\s+\1\b)");

  std::size_t repCount = 0;

  std::for_each(std::sregex_iterator(b, e, wordRepeatRgx),
                std::sregex_iterator(),
                [&](const std::smatch&) { ++repCount; }
               );

  return repCount;
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Regular Expressions Summary

- Several RE syntaxes and string representations are supported.
- Search functions are regex_match and regex_search.
- regex_replace does global search/replace; result is a new string.
- Match iteration done via regex_iterator/regex_token_iterator.
- Capture groups are supported.

© 2011 Scott Meyers, all rights reserved.

Again, regex_replace can be configured with flags to (1) replace only the first match and/or to (2) not write out the result of the replacements it performs (i.e., not return any new text), but by default, it behaves as summarized in these slides.

© 2011 Scott Meyers, all rights reserved.

From TR1: Generalized Functors

Motivation:

- Function pointers and member function pointers are rigid:
  - ➡ Exact parameter/return types and ex. specs. must be specified.
  - ➡ Can't point to nonstatic member functions.
  - ➡ Can't point to function objects.
- Useful to be able to refer to ***any callable entity compatible with a given calling interface***.
  - ➡ Convenient for developers (especially for callbacks).
  - ➡ Can help limit code bloat from template instantiations.

© 2011 Scott Meyers, all rights reserved.

Regarding code bloat, instead of instantiating a template for many types with the same calling interface, the template can be instantiated only once for the function type that specifies that interface. (Under the hood, the implementation machinery for std::function will be instantiated once for each actual type, but the template taking a std::function parameter will be instantiated only once for all types compatible with the std::function type.)

© 2011 Scott Meyers, all rights reserved.

Callable Entities

Something that can be called like a function:

- Functions, function pointers, function references:

```
void f(int x);                // function
void (*fp)(int) = f;          // function pointer
int val;
…
f(val);                       // call f
fp(val);                      // call *fp
```


© 2011 Scott Meyers, all rights reserved.

The term "callable entity" is mine and slightly more restricted than the C++0x notion of a "callable object," because callable objects include member data pointers. Callable objects also include pointers to nonstatic member functions, which I don't discuss in conjunction with std::function. (I do discuss them in conjunction with std::bind and lambdas, both of which produce function objects that can be stored in std::function objects.)

© 2011 Scott Meyers, all rights reserved.

Callable Entities

- Objects implicitly convertible to one of those:

```
class Widget {
public:
  using FuncPtr = void (*)(int);
  operator FuncPtr() const;     // conversion to function ptr
  …
};

Widget w;                       // object with conversion to func ptr

int val;
…
w(val);                         // "call" w, i.e.,
                                // invoke (w.operator FuncPtr())()
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Callable Entities

- Function objects (including closures):

```
class Gadget {
public:
  void operator()(int);    // function call operator
  …
};
Gadget g;                  // object supporting operator()
int val;
…
g(val);                    // "call" g, i.e., invoke w.operator()

auto f = [](int x) { return x < currentThreshhold(); };
if (f(val)) …              // "call" closure
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

std::function Basics

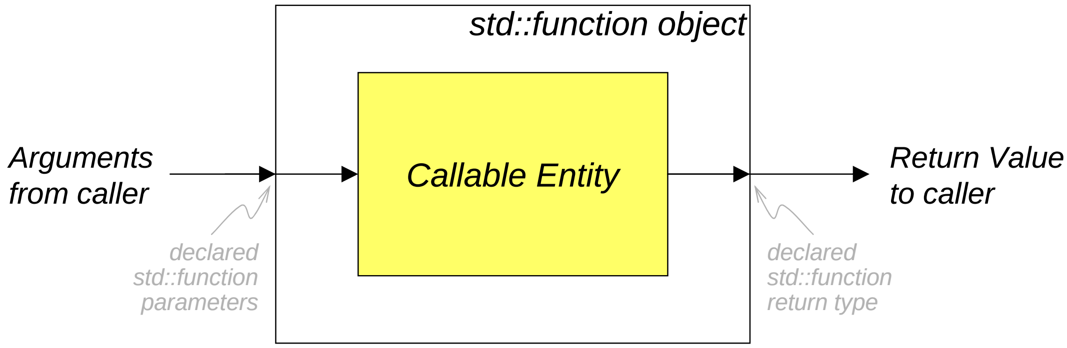
- Declared in <functional>.
- std::functions are type-safe wrappers for callable entities:

```
std::function<int(std::string&)> f;     // f refers to callable entity
                                        // compatible with given sig.

int someFunc(std::string&);             // some function
f = someFunc;                           // f refers to someFunc
f = [](std::string &s)->unsigned
    { s += "!"; return s.size(); };     // f refers to λ's closure
class Gadget {
public:
  int operator()(std::string&);         // function call operator
  …
};
Gadget g;
f = g;                                  // f refers to g
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Fundamental Behavior




```
int someFunc(std::string&);                        // some function
f = someFunc;
std::string str;
...
int result = f(str);                               // calls someFunc
f = [](std::string& s)->unsigned { s += "!"; return s.size(); };
...
result = f(str);                                   // calls λ's closure
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.

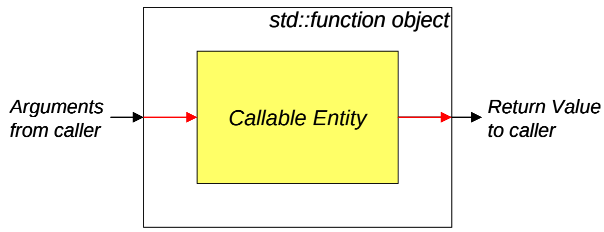
The outer box represents the std::function object, the inner box the callable entity it wraps (i.e., forwards calls to).

Scott Meyers, Software Development Consultant © 2011 Scott Meyers, all rights reserved.

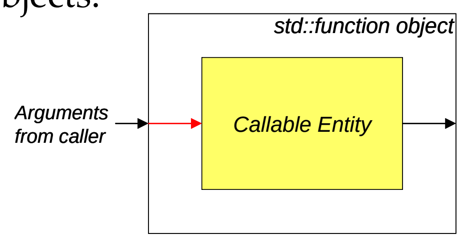
Compatible Signatures

A callable entity is *compatible* with a function object if:

- The function object's parameter types can be converted to the entity's parameter types.
- The entity's return type can be converted the function object's.



➡ All entity return types are compatible with void-returning function objects.



© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

function Callback Example

A Button class supporting click-event callbacks:

- The callback parameter indicates a down- or up-click.

```
class Button: public SomeGUIFrameworkBaseClass {
public:
  …
  using CallbackType = std::function<void(short)>;

  void setCallback(const CallbackType& cb)
  {
    clickHandler = cb;
  }

  virtual void onClick(short upOrDown)     // invoked by base class
  {
    clickHandler(upOrDown);                // invoke function object
  }

private:
  CallbackType clickHandler;
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

function Callback Example

```
void buttonClickHandler(int eventType);    // non-member function
class ButtonHandler {
public:
  …
  static int clicked(short upOrDown);      // static member function
};
void (*clicker)(int) = buttonClickHandler; // function pointer
Button b;
…
b.setCallback(buttonClickHandler);         // pass non-member func
b.setCallback(ButtonHandler::clicked);     // pass static member func
b.setCallback(clicker);                    // pass function ptr
```

Note the (compatible) type mismatches:

- buttonClickHandler and clicker take int, not short
- ButtonHandler::clicked returns int, not void

© 2011 Scott Meyers, all rights reserved.

For static member functions, the use of "&" before the name is optional when taking their address (i.e., same as non-member functions).

© 2011 Scott Meyers, all rights reserved.

function Callback Example

```
class ButtonClickCallback {                // class generating
public:                                    // function objects
   void operator()(short upOrDown) const;
};

Button b;
…
ButtonClickCallback bccb;

b.setCallback(bccb);                       // pass function object

void logClick(short upOrDown);
...
b.setCallback([](int v) { logClick(v); }); // pass closure; note
                                           // (compatible) param.
                                           // type mismatch
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

function Callback Example

```
class ButtonHandler {
public:
  …
  int clicked(short upOrDown) const;      // as before, but non-static
};
Button b;
ButtonHandler bh;
…
b.setCallback(std::bind(&ButtonHandler::clicked,
                        bh, _1));      // pass non-static member func;
                                       // info on std::bind coming soon
```


© 2011 Scott Meyers, all rights reserved.

ButtonHandler::clicked is declared const, because that avoids my having to mention mutable lambdas when I later contrast lambdas and bind.

_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

For non-static member functions, the use of "&" before the name is *not* optional when taking their address.

© 2011 Scott Meyers, all rights reserved.

Other function Characteristics

- Declared in <functional>
- Supports nullness testing:

```
std::function<signature> f;
…
if (f) …                                  // fine
if (f == nullptr) …                       // also fine
```

- Disallows equality and inequality testing
  - ➡ Nontrivial to determine whether two function objects refer to equal callable entities.

```
std::function<signature> f1, f2;
…
if (f1 == f2) …                           // error!
if (f1 != f2) …                           // error!
```


© 2011 Scott Meyers, all rights reserved.

operator== and operator!= are deleted functions (which have not yet been introduced).

© 2011 Scott Meyers, all rights reserved.

function Summary

- function objects are generalizations of function pointers.
- Can refer to any callable entity with a compatible signature.
- Especially useful for callback interfaces.
- Explicitly disallow tests for equality or inequality.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

From TR1: bind

Motivation:

- bind1st and bind2nd are constrained:
  - Bind only first or second arguments.
  - Bind only one argument at a time.
  - Can't bind functions with reference parameters.
  - Require adaptable function objects.
    - Often necessitates ptr_fun, mem_fun, and mem_fun_ref.

bind1st and bind2nd are deprecated in C++0x.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

bind

- Declared in <functional>.
- Produces a function object from:
  - A callable entity.
  - A specification of which arguments are to be bound.

```
functionObject std::bind(callableEntity,
                         1stArgBinding,
                         2ndArgBinding,
                         …
                         nthArgBinding);
```

  - *Placeholders* allow mapping from arguments for bind's return value to callable object arguments.
    - _*n* specifies the *n*th argument passed to the function object returned by bind.

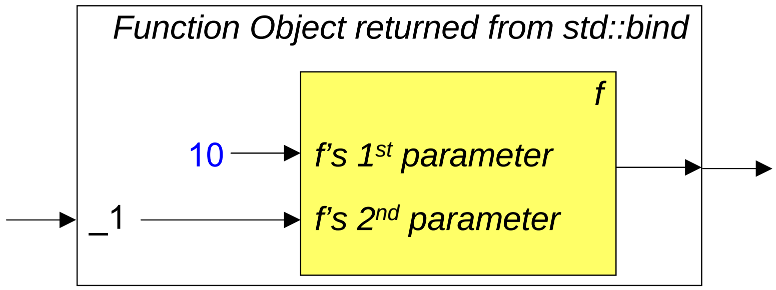
© 2011 Scott Meyers, all rights reserved.

The information on this page is likely to make sense only after examples have been presented.

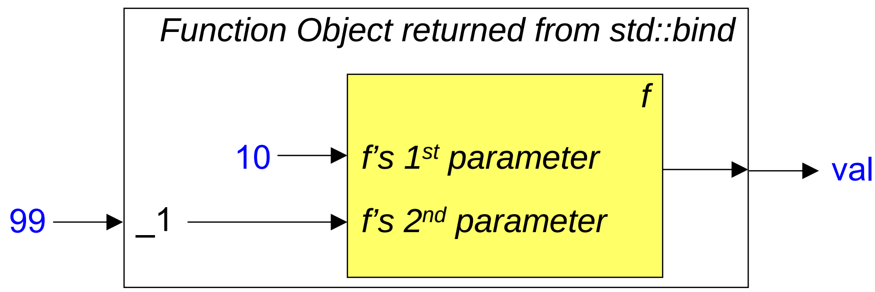
© 2011 Scott Meyers, all rights reserved.

bind Basics

```
int f(int x, int y);
auto bf = std::bind(f, 10, _1);          // same as std::bind1st(f, 10)
```




```
int val = bf(99);                         // call to bound function
```



© 2011 Scott Meyers, all rights reserved.

Placeholders or formal parameter names are shown in black just inside the box that represents the callable entity they apply to. Hence the outer box (representing the function object returned by std::bind) has a placeholder name of _1.

_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

bf is not the same object as the one returned from bind, but in all likelihood, it's been move-constructed from the rvalue returned by bind.

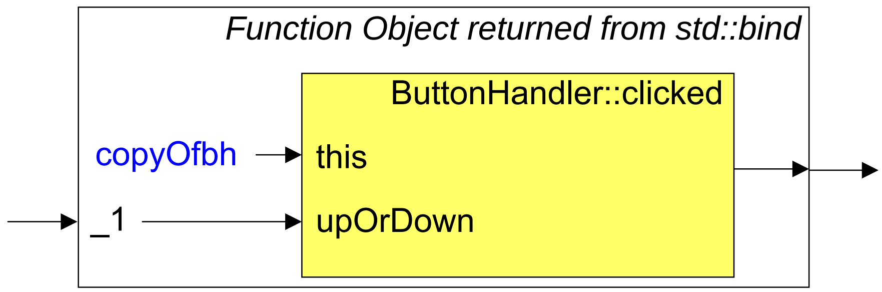
© 2011 Scott Meyers, all rights reserved.

Binding Non-Static Member Functions

For non-static member functions, this comes from the first argument:

- Just like for bind1st and bind2nd.

```
class ButtonHandler {                  // from the std::function example
public:
  …
  int clicked(short upOrDown) const;
};

Button b;
ButtonHandler bh;
…
b.setCallback(std::bind(&ButtonHandler::clicked, bh, _1));
```



© 2011 Scott Meyers, all rights reserved.

_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

std::bind copies the arguments it binds, hence the use of the name copyOfbh inside the function object returned by bind.

© 2011 Scott Meyers, all rights reserved.

Binding Non-Static Member Functions

bind supports this specified via:

- **Object:**

```
ButtonHandler bh;
b.setCallback(std::bind(&ButtonHandler::clicked, bh, _1));
```

- **Pointer:**

```
ButtonHandler *pbh;
b.setCallback(std::bind(&ButtonHandler::clicked, pbh, _1));
```

- **Smart Pointer:**

```
std::shared_ptr<ButtonHandler> sp;
b.setCallback(std::bind(&ButtonHandler::clicked, sp, _1));

std::unique_ptr<ButtonHandler> up;
b.setCallback(std::bind(&ButtonHandler::clicked, std::ref(up), _1));

MyCustomSmartPtr<ButtonHandler> mcsp;
b.setCallback(std::bind(&ButtonHandler::clicked, mcsp, _1));
```


© 2011 Scott Meyers, all rights reserved.

std::unique_ptr must be wrapped by std::ref when bound, because std::unique_ptr isn't copyable. (It's only movable.)

Any smart pointer will work with bind as long as it defines operator* in the conventional manner, i.e., to return a reference to the pointed-to object. (This implies that std::weak_ptr won't work with bind.)

© 2011 Scott Meyers, all rights reserved.

Binding Beyond the 2nd Argument

Binding beyond the 2nd argument is easy:

```
class Point {
public:
  …
  void translate(int deltaX, int deltaY);
};
std::vector<Point> vp;
…
std::for_each(                                    // translate points
  vp.begin(), vp.end(),                           // in vp by (10, 20);
  std::bind(&Point::translate, _1, 10, 20)        // note that deltaY
);                                                // is 3rd arg
```

- bind's placeholder arguments passed by reference, so this loop modifies Points in vp, not copies of them.

© 2011 Scott Meyers, all rights reserved.

Arguments corresponding to bind placeholders are passed using perfect forwarding.

_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

© 2011 Scott Meyers, all rights reserved.

bind and Adapters

Unlike bind1st and bind2nd, bind needs no adapters:

```
class Point {
public:
  …
  void setColor(Color c);
};
std::forward_list<Point> flp;
…
// without bind
std::for_each(
  flp.begin(), flp.end(),
  std::bind2nd(std::mem_fun_ref(&Point::setColor), green)
);
// with bind
std::for_each(flp.begin(), flp.end(),
              std::bind(&Point::setColor, _1, green));
```


© 2011 Scott Meyers, all rights reserved.

_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

© 2011 Scott Meyers, all rights reserved.

bind and function

bind's result often stored in a function object:

```
class Button: public SomeGUIFrameworkBaseClass {       // from
public:                                                  // std::function
   typedef std::function<void(short)> CallbackType;      // discussion

   void setCallback(const CallbackType& cb)
   { clickHandler = cb; }
   …
private:
  CallbackType clickHandler;
};

class ButtonHandler {                                    // from
public:                                                  // earlier
  int clicked(short upOrDown) const;
};

Button b;
ButtonHandler bh;
…
b.setCallback(std::bind(&ButtonHandler::clicked, bh, _1));
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 219

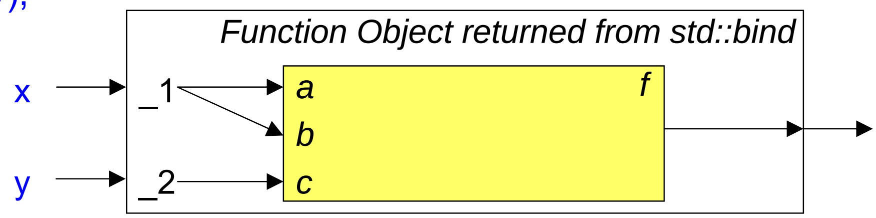
_1 is actually in namespace std::placeholders, so the call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

© 2011 Scott Meyers, all rights reserved.

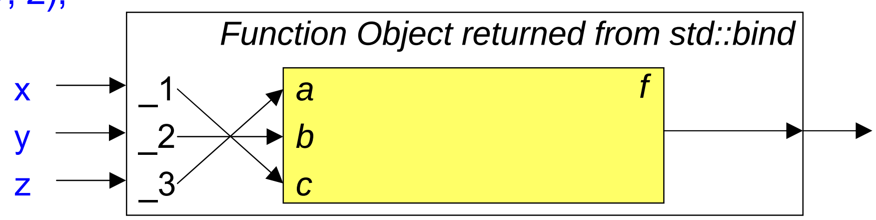
Fun With bind

bind allows reordering and duplicating arguments:

```
void f(int a, int b, int c);
int x, y, z;
…
std::function<void(int, int)> f1 = std::bind(f, _1, _1, _2);
f1(x, y);
```




```
std::function<void(int, int, int)> f2 = std::bind(f, _3, _2, _1);
f2(x, y, z);
```



© 2011 Scott Meyers, all rights reserved.

These diagrams are a little misleading, because they don't show the std::function objects that wrap the objects returned by std::bind.

_1, _2, and _3 are actually in namespace std::placeholders, so the calls to bind on this page won't compile as shown unless std::placeholders::_1 (and similarly for _2 and _3) have been made visible (e.g., via a using declaration). In practice, this is virtually always done in code that uses bind.

© 2011 Scott Meyers, all rights reserved.

Lambdas vs. bind

Both lambdas and bind create function objects:

```
std::for_each( vp.begin(), vp.end(),
               std::bind(&Point::translate, _1, 10, 20));
std::for_each(vp.begin(), vp.end(),
              [](Point& p) { p.translate(10, 20); });

b.setCallback(std::bind(&ButtonHandler::clicked, bh, _1));
b.setCallback([=](short upOrDown) { bh.clicked(upOrDown); });

std::function<void(int, int, int)> f2 = std::bind(f, _3, _2, _1);
std::function<void(int, int, int)> f2 = [](int a, int b, int c) { f(c, b, a); };
```

Many people find lambdas clearer.

- No _1, _2, etc.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 221

All the examples on this page are taken from the foregoing bind discussion.

© 2011 Scott Meyers, all rights reserved.

Lambdas vs. bind

Lambdas always clearer when more than simple binding needed:

```
class Person {
public:
  std::size_t age() const;
  …
};
std::vector<Person> vp;
…
std::partition(vp.begin(), vp.end(),
               [](const Person& p) { return p.age() < 21 || p.age() > 65; }
              );

std::partition( vp.begin(), vp.end(),
                std::bind(std::logical_or<bool>(),
                          std::bind(std::less<std::size_t>(),
                                    …_1…,                        // ???
                                    21),
                          std::bind(std::greater<std::size_t>(),
                                    …_1…,                        // ???
                                    65)
                         )
              );
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 222

As far as I know, there is no way to use bind with the call to partition, because there is no way to specify that _1 for the outer call to bind maps to _1 for the other two calls to bind.

© 2011 Scott Meyers, all rights reserved.

Lambdas vs. bind

Lambdas typically generate better code.

- Calls through bind involve function pointers ⇒ no inlining.
- Calls through closures allow full inlining.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

bind Summary

- Generalizes bind1st and bind2nd (which are now deprecated).
- No need for ptr_fun, mem_fun, mem_fun_ref, or std::mem_fn.
- Results often stored in function objects.
- Lambdas typically preferable.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

New Algorithms for C++0x

R is a range, e is an element, p is a predicate:

all_of	is p true for all e in R?
any_of	is p true for any e in R?
none_of	is p true for no e in R?
find_if_not	find first e in R where p is false
copy_if	copy all e in R where p is true
copy_n	copy first n elements of R
iota	assign all e in R increasing values starting with v
minmax	return pair(minVal, maxVal) for given inputs
minmax_element	return pair(min_element, max_element) for R

- min/max/minmax return values.
- min_element/max_element/minmax_element return iterators.

© 2011 Scott Meyers, all rights reserved.

The descriptions for minmax and minmax_element are different, because minmax is overloaded to take individual objects or an initializer_list, but not a range. minmax_element accepts only ranges.

© 2011 Scott Meyers, all rights reserved.

New Algorithms for C++0x

R is a range, e is an element, p is a predicate, v is a value:

partition_copy	copy all e in R to 1 of 2 destinations per p(e)
is_partitioned	is R partitioned per p?
partition_point	find first e in R where p(e) is false
is_sorted	is R sorted?
is_sorted_until	find first out-of-order e in R
is_heap	do elements in R form a heap?
is_heap_until	find first out-of-heap-order e in R
move	like copy, but each e in R is moved
move_backward	like copy_backward, but each e in R is moved

- std::move_iterator turns copying algorithms into moves, e.g.:

```
std::copy_if( std::move_iterator<It>(b),    // ≡ std::copy_if(b, e, p),
              std::move_iterator<It>(e),    // but moves instead of
              p);                           // copies
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 226

There is no actual need for std::move and std::move_backward, because their effects can be achieved with copy, copy_backward, and move_iterators, but, per a comp.std.c++ posting by Howard Hinnant, the committee felt that these two algorithms "might be used so often, move versions of them should be provided simply for notational convenience."

© 2011 Scott Meyers, all rights reserved.

Extended C++98 Algorithms in C++0x

swap	New overload taking arrays
min	New overloads taking initializer lists
max	New overloads taking initializer lists

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Summary of Library Enhancements

- Initializer lists, emplacement, and move semantics added to C++98 containers.
- TR1 functionality except mathematical functions adopted.
- forward_list is a singly-linked list.
- unique_ptr replaces auto_ptr.
- 18 new algorithms.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 228

There are some more subtle library changes in moving from C++03 to C++0x, e.g., function objects used with STL algorithms in C++03 are generally prohibited from having side effects, while in C++0x, some side effects are allowed. For example, in C++03, the specification for accumulate says that "binary_op shall not cause side effects," but in [accumulate] (26.7.2/2 of N3290), the corresponding wording is "In the range [first,last], binary_op shall neither modify elements nor invalidate iterators or subranges."

© 2011 Scott Meyers, all rights reserved.

Overview

- Introduction
- Features for Everybody
- Library Enhancements
- **Features for Class Authors**
- Features for Library Authors
- Yet More Features
- Further Information

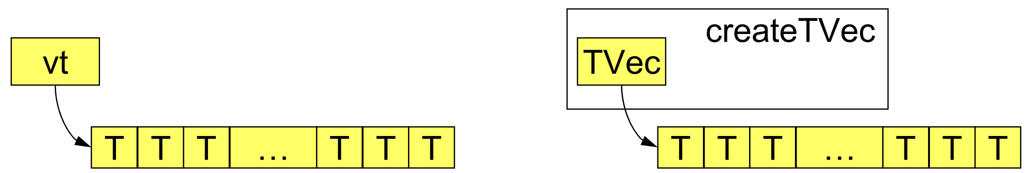
© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Move Support

C++ sometimes performs unnecessary copying:

```
typedef std::vector<T> TVec;
TVec createTVec();              // factory function
TVec vt;
…
vt = createTVec();              // copy return value object to vt,
                                // then destroy return value object
```



© 2011 Scott Meyers, all rights reserved.

The diagrams on this slide make up a PowerPoint animation.

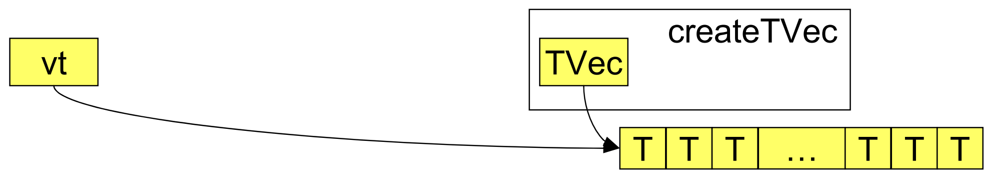
Throughout this discussion, I use a container of T, rather than specifying a particular type, e.g., container of string or container of int. The motivation for move semantics is largely independent of the types involved, although the larger and more expensive the types are to copy, the stronger the case for moving over copying.

© 2011 Scott Meyers, all rights reserved.

Move Support

Moving values would be cheaper:

```
TVec vt;
…
vt = createTVec();                // move data in return value object
                                  // to vt, then destroy return value
                                  // object
```



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 231

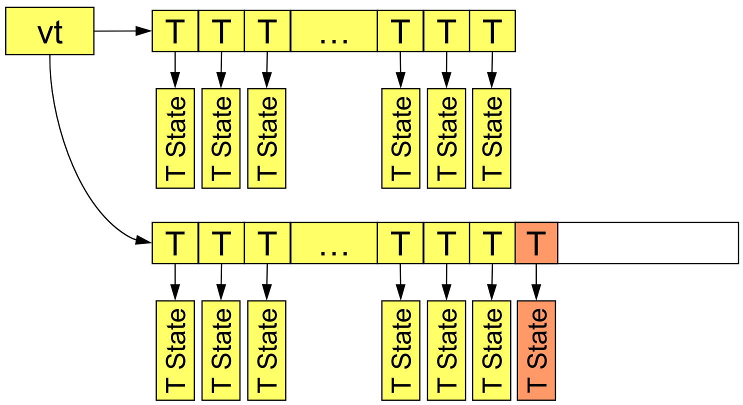
The diagrams on this slide make up a PowerPoint animation.

© 2011 Scott Meyers, all rights reserved.

Move Support

Appending to a full vector causes much copying before the append:

```
std::vector<T> vt;
...
vt.push_back(T object);                // assume vt lacks
                                       // unused capacity
```



© 2011 Scott Meyers, all rights reserved.

The diagrams on this slide make up a PowerPoint animation.

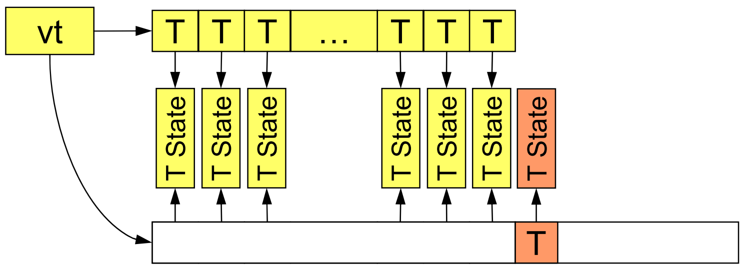
The new element has to be added to the new storage for the vector before the old elements are destroyed, because it's possible that the new element is a copy of an existing element, e.g. vt.emplace_back(vt[0]).

© 2011 Scott Meyers, all rights reserved.

Move Support

Again, moving would be more efficient:

```
std::vector<T> vt;
...
vt.push_back(T object);                  // assume vt lacks
                                         // unused capacity
```



Other vector and deque operations could similarly benefit.

- insert, emplace, resize, erase, etc.

© 2011 Scott Meyers, all rights reserved.

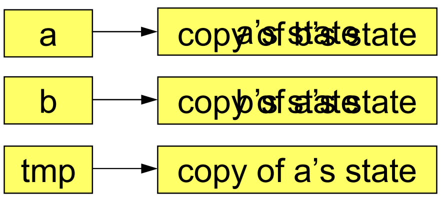
The diagrams on this slide make up a PowerPoint animation.

© 2011 Scott Meyers, all rights reserved.

Move Support

Still another example:

```
template<typename T>              // straightforward std::swap impl.
void swap(T& a, T& b)
{
  T tmp(a);                       // copy a to tmp (⇒ 2 copies of a)
  a = b;                          // copy b to a (⇒ 2 copies of b)
  b = tmp;                        // copy tmp to b (⇒ 2 copies of tmp)
}                                 // destroy tmp
```



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

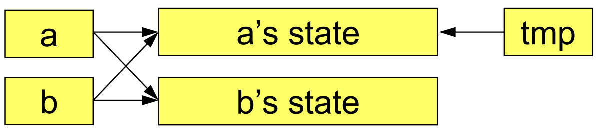
Slide 234

The diagrams on this slide make up a PowerPoint animation. That's why there appears to be overlapping text.

© 2011 Scott Meyers, all rights reserved.

Move Support

```
template<typename T>              // straightforward std::swap impl.
void swap(T& a, T& b)
{
  T tmp(std::move(a));            // move a's data to tmp
  a = std::move(b);               // move b's data to a
  b = std::move(tmp);             // move tmp's data to b
}                                 // destroy (eviscerated) tmp
```



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 235

The diagrams on this slide make up a PowerPoint animation.

std::move is defined in <utility>.

© 2011 Scott Meyers, all rights reserved.

Move Support

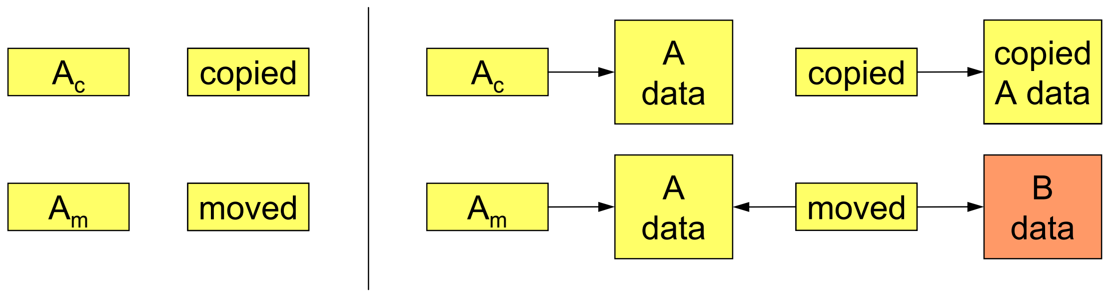
Moving most important when:

- Object has data in separate memory (e.g., on heap).
- Copying is deep.

Moving copies only object memory.

- Copying copies object memory **+ separate memory**.

Consider copying/moving A to B:



Moving never slower than copying, and often faster.

© 2011 Scott Meyers, all rights reserved.

The diagrams on this slide make up a PowerPoint animation. The upper line depicts copying objects with and without separate memory, the lower line depicts moving such objects.

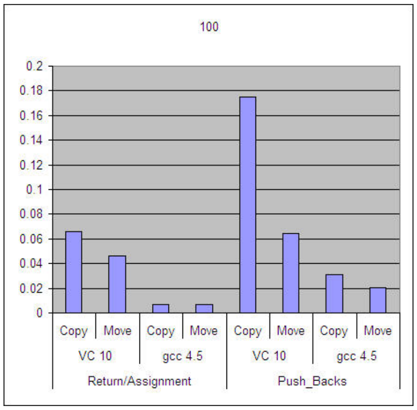
© 2011 Scott Meyers, all rights reserved.

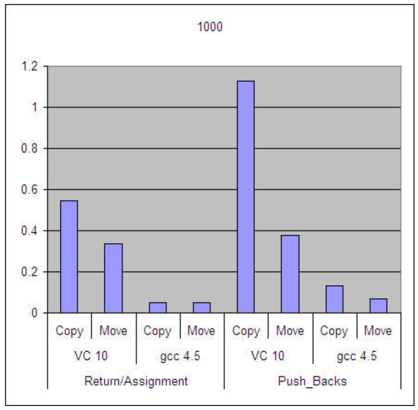
Performance Data

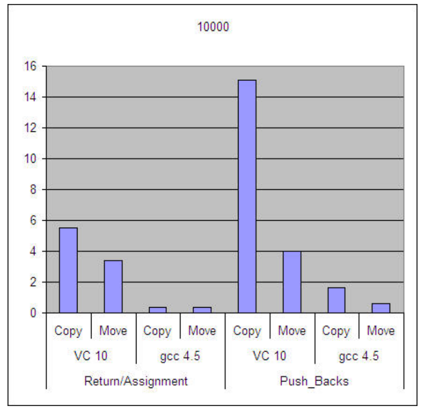
Consider these use cases again:

```
vt = createTVec();            // return/assignment
vt.push_back(T object);       // push_back
```

Copy-vs-move performance differences notable:







© 2011 Scott Meyers, all rights reserved.

All data are for a std::vector<Widget> of length n (where n = 100, 1000, or 10000, as indicated), where a Widget contains a single std::string data member with a value that's 29 characters in length. Data was collected on a Lenovo Z61t laptop.

© 2011 Scott Meyers, all rights reserved.

Move Support

Lets C++ recognize move opportunities and take advantage of them.

- How recognize them?
- How take advantage of them?

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Lvalues and Rvalues

Lvalues are generally things you can take the address of:

- Named objects.
- Lvalue references.
  - More on this term in a moment.

Rvalues are generally things you can't take the address of.

- Typically unnamed temporary objects.

Examples:

```
int x, *pInt;                    // x, pInt, *pInt are lvalues
std::size_t f(std::string str);  // str is lvalue, f's return is rvalue
f("Hello");                      // temp string created for call
                                 // is rvalue
std::vector<int> vi;             // vi is lvalue
…
vi[5] = 0;                       // vi[5] is lvalue
```

- Recall that vector<T>::operator[] returns T&.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2011 Scott Meyers, all rights reserved.
Slide 239

The this pointer is a named object, but it's defined to be an rvalue expression.

Per [expr.prim.general] (5.1.1/1 in N3290) literals (other than string literals) are rvalues, too, but those types don't define move operations, so they are not relevant for purposes of this discussion. User-defined literals yield calls to literal operator functions, and the temporaries returned from such functions are rvalues, so user-defined literals are rvalues, too, but not rvalues any different from any other temporary returned from a function, so they don't require any special consideration.

Because f takes its std::string parameter by value, a copy or move constructor should be called to initialize it. The call to f with "Hello" is thus supposed to generate a temporary, which is then used to initialize the parameter str. In practice, the copy or move operation will almost certainly be optimized away, and str will be initialized via std::string's constructor taking a const char*, but that does not change the analysis: f("Hello") generates a temporary std::string object, at least conceptually.

© 2011 Scott Meyers, all rights reserved.

Moving and Lvalues

Value movement generally not safe when the source is an lvalue.

- The lvalue object continues to exist, may be referred to later:

```
TVec vt1;
…
TVec vt2(vt1);                 // author expects vt1 to be
                               // copied to vt2, not moved!

…use vt1…                      // value of vt1 here should be
                               // same as above
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 240

In some cases, it's known that an lvalue object will never be referenced again, and in those cases, C++0x permits lvalues to be implicitly moved from. Such objects are known in (draft) C++0x as *xvalues*: lvalues that may be treated as rvalues. Probably the most common manifestation of an xvalue is an object being returned from a function, where C++0x permits the function's return value to be move-constructed from an lvalue `return` expression. As another example, an exception object may be move-constructed from an lvalue `throw` operand.

© 2011 Scott Meyers, all rights reserved.

Moving and Rvalues

Value movement is safe when the source is an rvalue.

- Temporaries go away at statement's end.
  - ➡No way to tell if their value has been modified.

```
TVec vt1;
vt1 = createTVec();             // rvalue source: move okay
auto vt2 { createTVec() };      // rvalue source: move okay
vt1 = vt2;                      // lvalue source: copy needed
auto vt3(vt2);                  // lvalue source: copy needed

std::size_t f(std::string str); // as before
f("Hello");                     // rvalue (temp) source: move okay
std::string s("C++0x");
f(s);                           // lvalue source: copy needed
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 241

In the example declaring/defining vt2, the move could be optimized away (as could the copy in C++98), but that doesn't change the fact that the source is an rvalue and hence a move could be used instead of a copy.

© 2011 Scott Meyers, all rights reserved.

Rvalue References

C++0x introduces **rvalue references**.

- Syntax: `T&&`
- "Normal" references now known as **lvalue references**.

Rvalue references behave similarly to lvalue references.

- Must be initialized, can't be rebound, etc.

Rvalue references identify objects that may be moved from.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Reference Binding Rules

Important for overloading resolution.

As always:

- Lvalues may bind to lvalue references.
- Rvalues may bind to lvalue references to const.

In addition:

- Rvalues may bind to rvalue references to non-const.
- Lvalues may *not* bind to rvalue references.
  - ➡ Otherwise lvalues could be accidentally modified.

© 2011 Scott Meyers, all rights reserved.

General rules governing reference binding are in [dcl.init.ref] (8.5.3 in N3290), and rules governing the interaction of reference binding and overloading resolution are in [over.ics.ref] (13.3.3.1.4 in N3290) and [over.ics.rank] (13.3.3.2 in N3290, especially 13.3.3.2/3 which states that in case of a tie between binding to an lvalue reference or an rvalue reference, rvalues preferentially bind to rvalue references. A tie can occur only when one function takes a parameter of type const T& and the other a type of const T&&, because rvalues can't bind at all to non-const T& parameters, and a non-const rvalue would prefer to bind to a T&& parameter over a const T& parameter, because the former would not require the addition of const.

There was a time in draft C++0x when lvalues were permitted to bind to rvalue references, and some compilers (e.g., gcc 4.3 and 4.4 (but not 4.5), VC10 beta 1 (but not beta 2 or subsequent releases)) implemented this behavior. This is sometimes known as "version 1 of rvalue references." Motivated by N2812, the rules were changed such that lvalues may not bind to rvalue references, sometimes called "version 2 of rvalue references." Developers need to be aware that some older compilers supporting rvalue references may implement the "version 1" rules instead of the version 2 rules.

© 2011 Scott Meyers, all rights reserved.

Rvalue References

Examples:

```
void f1(const TVec&);          // takes const lvalue ref
TVec vt;
f1(vt);                        // fine (as always)
f1(createTVec());              // fine (as always)

void f2(const TVec&);          // #1: takes const lvalue ref
void f2(TVec&&);               // #2: takes non-const rvalue ref
f2(vt);                        // lvalue ⇒ #1
f2(createTVec());              // both viable, non-const rvalue ⇒ #2

void f3(const TVec&&);         // #1: takes const rvalue ref
void f3(TVec&&);               // #2: takes non-const rvalue ref
f3(vt);                        // error!  lvalue
f3(createTVec());              // both viable, non-const rvalue ⇒ #2
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Rvalue References and const

C++ remains const-correct:

- const lvalues/rvalues bind only to references-to-const.

But rvalue-references-to-const are essentially useless.

- Rvalue references designed for two specific problems:
  - ➡ Move semantics
  - ➡ Perfect forwarding
- C++0x language rules carefully crafted for these needs.
  - ➡ rvalue-refs-to-const not considered in these rules.
- const T&&s are legal, but not designed to be useful.
  - ➡ Uses already emerging :-)

© 2011 Scott Meyers, all rights reserved.

The crux of why rvalue-references-to-const are not useful is the special handling accorded T&& parameters in [temp.deduct.call] (14.8.2.1/3 in N3290): "If P is an rvalue reference to a cv-unqualified template parameter [i.e. T&&] and the argument is an lvalue, the type 'lvalue reference to A' [i.e., T&] is used in place of A for type deduction." This hack applies only to T&& parameters, not const T&& parameters.

The emerging use for const T&& function template parameters is to allow binding lvalues while prohibiting binding rvalues, e.g., from [function.objects] (20.8/2 in N3290):

```
template <class T> reference_wrapper<T> ref(T&) noexcept;
template <class T> reference_wrapper<const T> cref(const T&) noexcept;

template <class T> void ref(const T&&) = delete;
template <class T> void cref(const T&&) = delete;
```


© 2011 Scott Meyers, all rights reserved.

Rvalue References and const

Implications:

- Don't declare const T&& parameters.
  - You wouldn't be able to move from them, anyway.
  - Hence this (from a prior slide) rarely makes sense:

```
void f3(const TVec&&);            // legal, rarely reasonable
```

- Avoid creating const rvalues.
  - They can't bind to T&& parameters.
  - E.g., avoid const function return types:
    - This is a change from C++98.

```
class Rational { … };

const Rational operator+( const Rational&,      // legal, but
                          const Rational&);     // poor design

Rational operator+(const Rational&,             // better design
                   const Rational&);
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Distinguishing Copying from Moving

Overloading exposes move-instead-of-copy opportunities:

```
class Widget {
public:
  Widget(const Widget&);                  // copy constructor
  Widget(Widget&&);                       // move constuctor

  Widget& operator=(const Widget&);       // copy assignment op
  Widget& operator=(Widget&&);            // move assignment op
  …
};

Widget createWidget();                    // factory function

Widget w1;

Widget w2 = w1;                           // lvalue src ⇒ copy req'd

w2 = createWidget();                      // rvalue src ⇒ move okay

w1 = w2;                                  // lvalue src ⇒ copy req'd
```

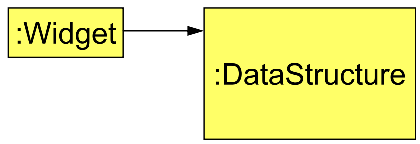

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implementing Move Semantics

Move operations take source's value, but leave source in valid state:

```
class Widget {
public:
  Widget(Widget&& rhs)
  : pds(rhs.pds)                    // take source's value
  { rhs.pds = nullptr; }            // leave source in valid state
  Widget& operator=(Widget&& rhs)
  {
    delete pds;                     // get rid of current value
    pds = rhs.pds;                  // take source's value
    rhs.pds = nullptr;              // leave source in valid state
    return *this;
  }
  …
private:
  struct DataStructure;
  DataStructure *pds;
};
```



Easy for built-in types (e.g., pointers). Trickier for UDTs…

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 248

A move operation needs to do three things: get rid of the destination's current value, move the source's value to the destination, and leave the source in a valid state. For UDTs, memberwise move is the way to achieve all three. For types managing primitive types (e.g., pointers, semaphores, etc.), their move operations have to do these things manually.

A generic, "clever" (i.e., suspicious) way to implement move assignment for a type T is

```
T& operator=(T&& rhs) { T(rhs).swap(*this); return *this; }
```

This has the effect of swapping the contents of *this and rhs. The idea is that because rhs is an rvalue reference, it's bound to an rvalue, and that rvalue will be destroyed at the end of the statement containing the assignment. When it is, the data formerly associated with *this will be destroyed (e.g., resources will be released). The problem is that rhs may actually correspond to an lvalue that has been explicitly std::move'd, and in that case, the lvalue may not be destroyed until later than expected. That can be problematic. Details can be found at http://thbecker.net/articles/rvalue_references/section_04.html and http://cpp-next.com/archive/2009/09/your-next-assignment/ .

"UDT" = "User Defined Type".

© 2011 Scott Meyers, all rights reserved.

Implementing Move Semantics

Widget's move operator= fails given move-to-self:

```
Widget w;
w = std::move(w);                    // undefined behavior!
```

It may be harder to recognize, of course:

```
Widget *pw1, *pw2;
…
*pw1 = std::move(*pw2);              // undefined if pw1 == pw2
```

C++0x likely to condone this.

- In contrast to copy operator=.

A fix is simple, if you are inclined to implment it:

```
Widget& Widget::operator=(Widget&& rhs)
{
   if (this == &rhs) return *this;       // or assert(this != &rhs);
     …
}
```


© 2011 Scott Meyers, all rights reserved.

The condoning of "self-move-assignment yields undefined behavior" is found at http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#1204. A discussion of the issue can be found in the comments at the end of http://cpp-next.com/archive/2009/09/making-your-next-move/ .

© 2011 Scott Meyers, all rights reserved.

Implementing Move Semantics

Part of C++0x's string type:

```
string::string(const string&);          // copy constructor
string::string(string&&);               // move constructor
```

An incorrect move constructor:

```
class Widget {
private:
  std::string s;

public:
  Widget(Widget&& rhs)                  // move constructor
  : s(rhs.s)                            // compiles, but copies!
  { … }
  …
};
```

- rhs.s an **lvalue**, because it has a name.
  - ➡ Lvalueness/rvalueness orthogonal to type!
    - ◆ ints can be lvalues or rvalues, and rvalue references can, too.
  - ➡ s initialized by string's *copy* constructor.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Implementing Move Semantics

Another example:

```
class WidgetBase {
public:
  WidgetBase(const WidgetBase&);          // copy ctor
  WidgetBase(WidgetBase&&);               // move ctor
  …
};

class Widget: public WidgetBase {
public:
  Widget(Widget&& rhs)                    // move ctor
  : WidgetBase(rhs)                       // copies!
  { … }
  …
};
```

- rhs is an **lvalue**, because it has a name.
  - ➡ Its declaration as Widget&& not relevant!

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Explicit Move Requests

To request a move on an lvalue, use std::move:

```
class WidgetBase { … };
class Widget: public WidgetBase {
public:
  Widget(Widget&& rhs)                          // move constructor
  : WidgetBase(std::move(rhs)),                 // request move
    s(std::move(rhs.s))                         // request move
  { … }
  Widget& operator=(Widget&& rhs)               // move assignment
  {
    WidgetBase::operator=(std::move(rhs));      // request move
    s = std::move(rhs.s);                       // request move
    return *this;
  }
  …
};
```

std::move turns lvalues into rvalues.

- The overloading rules do the rest.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 252

The move assignment operator on this page fails to worry about move-to-self.

© 2011 Scott Meyers, all rights reserved.

Why move Rather Than Cast?

std::move uses implicit type deduction. Consider:

```
template<typename It>
void someAlgorithm(It begin, It end)
{
  // permit move from *begin to temp, static_cast version
  auto temp1 =
    static_cast<typename std::iterator_traits<It>::value_type&&>(*begin);

  // same thing, C-style cast version
  auto temp2 = (typename std::iterator_traits<It>::value_type&&)*begin;

  // same thing, std::move version
  auto temp3 = std::move(*begin);

  ...

}
```

What would you rather type?

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implementing std::move

std::move is simple – in concept:

```
template<typename T>
T&&                              // return as an rvalue whatever
move(MagicReferenceType obj)     // is passed in; must work with
{                                // both lvalue/rvalues
  return obj;
}
```

Between concept and implementation lie arcane language rules.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Reference Collapsing in Templates

In C++98, given

```
template<typename T>
void f(T& param);

int x;

f<int&>(x);                     // T is int&
```

f is initially instantiated as

```
void f(int& & param);           // reference to reference
```

C++98's reference-collapsing rule says

- T& & ⇒ T&

so f's instantiation is actually:

```
void f(int& param);             // after reference collapsing
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Reference Collapsing in Templates

C++0x's rules take rvalue references into account:

- T& & ⇒ T& // from C++98
- T&& & ⇒ T& // new for C++0x
- T& && ⇒ T& // new for C++0x
- T&& && ⇒ T&& // new for C++0x

Summary:

- Reference collapsing involving a & is always T&.
- Reference collapsing involving only && is T&&.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 256

These rules are defined by [dcl.ref] (8.3.2/6 in N3290).

© 2011 Scott Meyers, all rights reserved.

std::move's Return Type

To guarantee an rvalue return type, std::move does this:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(MagicReferenceType obj)
{
  return obj;
}
```

- Recall that a T& return type would be an lvalue!

Hence:

```
int x;
std::move<int&>(x);  // calls remove_reference<int&>::type&& std::move(/*…*/)
                     // ⇒ int&& std::move(/*…*/)
```

- Without remove_reference, move<int&> would return int&.

© 2011 Scott Meyers, all rights reserved.

std::remove_reference is part of the type traits functionalityin C++0x. It turns both T& and T&& types into T.

© 2011 Scott Meyers, all rights reserved.

std::move's Parameter Type

Must be a non-const reference, because we want to move its value.

An lvalue reference doesn't work.

- Rvalues can't bind to them:

```
TVec createTVec();                  // as before

TVec&& std::move(TVec& obj);        // possible move
                                    // instantiation

std::move(createTVec());            // error!
```


© 2011 Scott Meyers, all rights reserved.

Note that this page shows move as a function, not a template.

© 2011 Scott Meyers, all rights reserved.

std::move's Parameter Type

An rvalue reference doesn't, either.

- Lvalues can't bind to them.

```
TVec&& std::move(TVec&& obj);              // possible move
                                           // instantiation
TVec vt;
std::move(vt);                             // error!
```

What std::move needs:

- For lvalue arguments, a parameter type of T&.
- For rvalue arguments, a parameter type of T&&.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.
Slide 259

Note that this page shows move as a function, not a template.

© 2011 Scott Meyers, all rights reserved.

Why Not Just Overload?

Overloading could solve the problem:

```
template<typename T>
typename std::remove_reference<T>::type&& move(T& lvalue)
{ return static_cast<T&&>(lvalue); }

template<typename T>
typename std::remove_reference<T>::type&& move(T&& rvalue)
{ return static_cast<T&&>(rvalue); }
```

But the perfect forwarding problem would remain:

- How forward *n* arguments to another function?
  - ➡ We'd need 2^n overloads!

Rvalue references aimed at both std::move and perfect forwarding.

© 2011 Scott Meyers, all rights reserved.

This slide assumes the C++98/C++03 rules for template argument deduction, i.e., that no distinction is drawn between lvalue and rvalue arguments for purposes of determining T.

© 2011 Scott Meyers, all rights reserved.

T&& Parameter Deduction in Templates

Given

```
template<typename T>
void f(T&& param);              // note non-const rvalue reference
```

T's deduced type depends on what's passed to param:

- **Lvalue** ⇒ T is an lvalue reference (T&)
- **Rvalue** ⇒ T is a non-reference (T)

In conjunction with reference collapsing:

```
int x;
f(x);                 // lvalue: generates f<int&>(int& &&),
                      // calls f<int&>(int&)
f(10);                // rvalue: generates/calls f<int>(int&&)
TVec vt;
f(vt);                // lvalue: generates f<TVec&>(TVec& &&),
                      // calls f<TVec&>(TVec&)
f(createTVec());      // rvalue: generates/calls f<TVec>(TVec&&)
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implementing std::move

std::move's parameter is thus T&&:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj)
{
  return obj;
}
```

This is almost correct. Problem:

- obj is an lvalue. (It has a name.)
- move's return type is an rvalue reference.
- Lvalues can't bind to rvalue references.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implementing std::move

A cast eliminates the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj)
{
  using ReturnType =
    typename std::remove_reference<T>::type&&;

  return static_cast<ReturnType>(obj);
}
```

This is a correct implementation.

© 2011 Scott Meyers, all rights reserved.

I believe (but have not yet confirmed) that it's possible to avoid repeating std::move's return type in the body of the function by apply decltype to move itself:

```
template<typename T>
type std::remove_reference<T>::type&&
move(T&& obj)
{
  using ReturnType = decltype(move(obj));

  return static_cast<ReturnType>(obj);
}
```


© 2011 Scott Meyers, all rights reserved.

T&& Parameters in Templates

Note that function templates with a T&& parameter need not generate functions taking a T&& parameter!

```
template<typename T>
void f(T&& param);          // as before

int x;

f(x);                       // still calls f<int&>(int&),
                            // i.e., f(int&)

f(10);                      // still calls f<int>(int&&),
                            // i.e., f(int&&)

TVec vt;

f(vt);                      // still calls f<TVec&>(TVec&),
                            // i.e., f(TVec&)

f(createTVec());            // still calls f<TVec>(TVec&&),
                            // i.e., f(TVec&&)
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

T&& Parameters in Templates

T&&-taking function templates should be read as "takes anything:"

```
template<typename T>
void f(T&& param);              // takes anything:  lvalue or rvalue,
                                // const or non-const
```

- Lvalues can't bind to rvalue references, but param may bind to an lvalue.
  - ➡ After instantiation, param's type may be T&, not T&&.
- Important for perfect forwarding (described shortly).

T&& as a "takes anything" parameter applies only to templates!

- For functions, a && parameter binds only to non-const rvalues:

```
void f(Widget&& param);       // takes only non-const rvalues
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

auto&& ≡ T&&

auto type deduction ≡ template type deduction, so an auto&& variable's type may be an lvalue reference:

```
int calcVal();

int x;

auto&& v1 = calcVal();          // deduce type from rvalue ⇒
                                // v1's type is int&&

auto&& v2 = x;                  // deduce type from lvalue ⇒
                                // v2's type is int&
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Move is an Optimization of Copy

Move requests for copyable types w/o move support yield copies:

```
class Widget {                          // class w/o move support
public:
  Widget(const Widget&);                // copy ctor
};

class Gadget {                          // class with move support
public:
  Gadget(Gadget&& rhs)                  // move ctor
  : w(std::move(rhs.w))                 // request to move w's value
  { … }

private:
  Widget w;                             // lacks move support
};
```

rhs.w is *copied* to w:

- std::move(rhs.w) returns an rvalue of type Widget.
- That rvalue is passed to Widget's copy constructor.

© 2011 Scott Meyers, all rights reserved.

Move requests on types that are not copyable but also lack move support will fail to compile.

© 2011 Scott Meyers, all rights reserved.

Move is an Optimization of Copy

If Widget adds move support:

```
class Widget {
public:
  Widget(const Widget&);          // copy ctor
  Widget(Widget&&);               // move ctor
};

class Gadget {                    // as before
public:
  Gadget(Gadget&& rhs)
  : w(std::move(rhs.w)) { … }     // as before

private:
  Widget w;
};
```

rhs.w is now *moved* to w:

- std::move(rhs.w) still returns an rvalue of type Widget.
- That rvalue now passed to Widget's move constructor.
  - Via normal overloading resolution.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Move is an Optimization of Copy

Implications:

- Giving classes move support can improve performance even for move-unaware code.
  - ➡ Copy requests for rvalues may silently become moves.
- Move requests safe for types w/o explicit move support.
  - ➡ Such types perform copies instead.
    - ◆ E.g., all built-in types.

In short:

- **Give classes move support.**
- **Use std::move for lvalues that may safely be moved from.**

© 2011 Scott Meyers, all rights reserved.

Both move and copy operations may throw, and the issues associatied with exceptions in move functions are essentially the same as those associated with copy functions. E.g., both must implement at least the basic guarantee, both should document the guarantee they offer, clients must take into account that such functions might throw, etc.

N2983 recommends that generic (i.e., template-based) code wishing to offer the strong guarantee but that uses a move operation on an unknown type T offer a *conditional* guarantee: the generic code offers the strong guarantee only if T's version of the move operation offers the strong (or nothrow) guarantee. This approach is viable for generic code using any unknown operation, however; there is nothing move-specific about it.

N2983 also explains how std::move_if_noexcept can be used in the tiny corner case of (1) legacy code offering the strong guarantee (2) that is being revised to replace copy operations known to offer the strong guarantee (3) with move operations not known to offer that guarantee. std::move_if_noexcept on an object of type T is like std::move on that object, except it performs a copy instead of a move unless the relevant T move operation is known to not throw.

 © 2011 Scott Meyers, all rights reserved.

Implicitly-Generated Move Operations

Move constructor and move operator= are "special: "

- Generated by compilers under appropriate conditions.

Conditions:

- All data members and base classes are movable.
  - Implicit move operations move everything.
  - Most types qualify:
    - All built-in types (move ≡ copy).
    - Most standard library types (e.g., all containers).
- Generated operations likely to maintain class invariants.
  - No user-declared copy or move operations.
    - Custom semantics for any ⇒ default semantics inappropriate.
    - Move is an optimization of copy.
  - No user-declared destructor.
    - Often indicates presence of implicit class invariant.

© 2011 Scott Meyers, all rights reserved.

Library types that aren't movable tend to be infrastructure-related, e.g., (to quote from a Daniel Krügler post in the comp.std.c++ thread at http://tinyurl.com/3afblkw) "type_info, error_category, all exception classes, reference_wrapper, all specializations from the primary allocator template, weak_ptr, enable_shared_from_this, duration, time_point, all iterators / iterator adaptors I am aware of, local::facet, locale::id, random_device, seed_seq, ios_base, basic_istream<charT,traits>::sentry, basic_ostream<charT,traits>::sentry, all atomic types, once_flag, all mutex types, lock_guard, all condition variable types."

© 2011 Scott Meyers, all rights reserved.

Destructors and Implicit Class Invariants

```
class Widget {
private:
  std::vector<int> v;
  std::set<double> s;
  std::size_t sizeSum;

public:
  ~Widget() { assert(sizeSum == v.size()+s.size()); }
  ...
};
```

If Widget had implicitly-generated move operations:

```
{
  std::vector<Widget> vw;
  Widget w;
  ...                               // put stuff in w's containers
  vw.push_back(std::move(w));       // move w into vw
  ...                               // no use of w
}                                   // assert fires!
```

User-declared dtor ⇒ no compiler-generated move ops for Widget.

© 2011 Scott Meyers, all rights reserved.

The assertion would fire, because the moved-from w would have empty containers (presumably), but sizeSum would continue to have a value corresponding to the containers' pre-move sizes.

© 2011 Scott Meyers, all rights reserved.

Implicitly-Generated Move Operations

Examples:

```
class Widget1 {                        // copyable & movable type
private:
  std::u16string name;                 // copyable/movable type
  long long value;                     // copyable/movable type
public:
  explicit Widget1(std::u16string n);
};                                     // implicit copy/move ctor;
                                       // implicit copy/move operator=

class Widget2 {                        // copyable type; not movable
private:
  std::u16string name;
  long long value;
public:
  explicit Widget2(std::u16string n);
  Widget2(const Widget2& rhs);         // user-declared copy ctor
};                                     // ⇒ no implicit move ops;
                                       // implicit copy operator=
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Custom Moving ⇒ Custom Copying

Declaring a move operation prevents generation of copy operations.

- Custom move semantics ⇒ custom copy semantics.
  - Move is an optimization of copy.

```
class Widget3 {                        // movable type; not copyable
private:
  std::u16string name;
  long long value;

public:
  explicit Widget3(std::u16string n);
  Widget3(Widget3&& rhs);              // user-declared move ctor
                                       // ⇒ no implicit copy ops;
  Widget3&                             // user-declared move op=
    operator=(Widget3&& rhs);          // ⇒ no implicit copy ops
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implicit Copy Operations Revisited

Rules for implicit copy operations can lead to trouble:

```
class ProblemSince1983 {                        // copyable class
public:
  ~ProblemSince1983() { delete p; }             // implicit invariant:
                                                // p owns *p
   ...                                          // no copy ops
                                                // declared
private:
  int *p;
};

{                                               // some scope
  ProblemSince1983 prob1;
  ...
  ProblemSince1983 prob2(prob1);
  ...
}                                               // double delete!
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implicit Copy Operations Revisited

Ideally, rules for copying would mirror rules for moving, i.e.,

- Declaring a custom move op ⇒ no implicit copy ops.
  - Already true.
- Declaring any copy op ⇒ no implicit copy ops.
  - Too big a change for C++0x.
- Declaring a destructor ⇒ no implicit copy ops.
  - Too big a change for C++0x.

However:

- Implicit copy ops **deprecated** in classes with user-declared copy, move, or dtor operations.
  - Compilers may issue warnings.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Implicit Copy Operations Revisited

```
class ProblemSince1983 {                    // as before
public:
  ~ProblemSince1983() { delete p; }
  ...                                       // no copy ops
private:
  int *p;
};

{                                           // as before
  ProblemSince1983 prob1;
  ...
  ProblemSince1983 prob2(prob1);            // generation of copy
                                            // ctor deprecated
  ...
}                                           // still double delete
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Copying, Moving, and Concurrency

Conceptually, copying an object reads it, but moving also writes it:

```
Widget w1;
Widget w2(w1);                    // read w1, but don't modify it
Widget w3(std::move(w1));         // both read and modify w1
```

Conceptually, in an MT environment:

- Concurrent copying of an object is safe.
- Concurrent moving of an object is *not* safe.

Concurrent copies/moves possible only with lvalues:

- Rvalues visible only in thread where they're created.
- Concurrent moves entail use of std::move in multiple threads.
  - std::moves on shared objects require manual synchronization.
    - E.g., use of std::lock_guard or std::unique_lock.

© 2011 Scott Meyers, all rights reserved.

MT = "Multi-threaded".

© 2011 Scott Meyers, all rights reserved.

Copying, Moving, and Concurrency

Conceptual reality is simplistic:

- Copying an object may modify it.
  - mutable data members.
  - Copy constructors with a non-const param (e.g., std::auto_ptr).
  - Copying shared objects may require manual synchronization.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Move Operations

- May exist even if copy operations don't.
  - E.g., std::thread and std::unique_ptr moveable, but not copyable.
    - "Move-only types"
- Types should provide when moving cheaper than copying.
  - Libraries use moves whenever possible (e.g., STL, Boost, etc.).
- May lead to races in MT environments.
  - Synchronization your responsibility.
  - Applies to some copy operations, too.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Beyond Move Construction/Assignment

Move support useful for other functions, e.g., setters:

```
class Widget {
public:
  ...
  void setName(const std::string& newName)
  { name = newName; }                              // copy param

  void setName(std::string&& newName)
  { name = std::move(newName); }                   // move param

  void setCoords(const std::vector<int>& newCoords)
  { coordinates = newCoords; }                     // copy param

  void setCoords(std::vector<int>&& newCoords)
  { coordinates = std::move(newCoords); }          // move param
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Construction and Perfect Forwarding

Constructors often copy parameters to data members:

```
class Widget {
public:
  Widget(const std::string& n, const std::vector<int>& c)
  :  name(n),                        // copy n to name
     coordinates(c)                  // copy c to coordinates
  {}
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Construction and Perfect Forwarding

Moves for rvalue arguments would be preferable:

```
std::string lookupName(int id);
int widgetID;
...
std::vector<int> tempVec;                    // used only for Widget ctor
...
Widget w(lookupName(widgetID),               // rvalues args, but Widget
         std::move(tempVec));                // ctor copies to members
```

Overloading Widget ctor for lvalue/rvalue combos ⇒ 4 functions.

- Generally, n parameters requires 2^n overloads.
  - Impractical for large n.
  - Boring/repetitive/error-prone for smaller n.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Construction and Perfect Forwarding

Goal: one function that "does the right thing:"

- Copies lvalue args, moves rvalue args.

Solution is a **perfect forwarding** ctor:

- Templatized ctor forwarding T&& params to members:

```
class Widget {
public:
  template<typename T1, typename T2>
  Widget(T1&& n, T2&& c)
  :  name(std::forward<T1>(n)),            // forward n to string ctor
     coordinates(std::forward<T2>(c))       // forward c to vector ctor
  {}
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 283

As noted on a later slide, this doesn't behave precisely like the non-template constructor, because perfect forwarding isn't perfect.

© 2011 Scott Meyers, all rights reserved.

Construction and Perfect Forwarding

Once again:

- A *templatized ctor forwarding* ***T&&*** *params* to members:

```
class Widget {
public:
  template<typename T1, typename T2>
  Widget(T1&& n, T2&& c)
  : name(std::forward<T1>(n)),           // forward n to string ctor
    coordinates(std::forward<T2>(c))     // forward c to vector ctor
  {}
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```

Effect:

- Lvalue arg passed to n ⇒ std::string ctor receives lvalue.
- Rvalue arg passed to n ⇒ std::string ctor receives rvalue.
- Similarly for for c and std::vector ctor.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding Beyond Construction

Useful for more than just construction, e.g., for setters:

```
class Widget {                                          // revised
public:                                                 // example
  ...
  template<typename T>
  void setName(T&& newName)                             // forward
  { name = std::forward<T>(newName); }                  // newName

  template<typename T>
  void setCoords(T&& newCoords)                         // forward
  { coordinates = std::forward<T>(newCoords); }         // newCoords
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```


© 2011 Scott Meyers, all rights reserved.

As noted on a later slide, this doesn't behave precisely like the non-template setters, because perfect forwarding isn't perfect.

© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding Beyond Construction

Despite T&& parameter, code fully type-safe:

- Type compatibility verified upon instantiation.
  - E.g., only std::string-compatible types valid in setName.

More flexible than a typed parameter.

- Accepts/forwards all compatible parameter types.
  - E.g., std::string, char*, const char* for setName.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding Beyond Construction

Flexibility can be removed via static_assert (described soon):

```
template<typename T>
void setName(T&& newName)
{
  static_assert(std::is_same< typename std::decay<T>::type,
                              std::string
                            >::value,
                "T must be a [const] std::string"
  );
  name = std::forward<T>(newName);
};
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 287

[static_assert has not been introduced yet.]

std::decay<T>::type is, for non-array and non-function types, equivalent to std::remove_cv<std::remove_reference<T>::type>::type.

std::enable_if could also be used, but static_assert seems simpler and clearer in this case. std::enable_if would remove setName from the overload set, while static_assert would be evaluated only after setName had been selected as the overload to be called.

© 2011 Scott Meyers, all rights reserved.

std::forward

Consider again:

```
template<typename T>
void setName(T&& newName)
{ name = std::forward<T>(newName); }
```

- T a reference (i.e., T is T&) ⇒ lvalue was passed to newName.
  - ➡ std::forward<T>(newName) should return lvalue.
- T a non-reference (i.e., T is T) ⇒ rvalue was passed to newName.
  - ➡ std::forward<T>(newName) should return rvalue.

Reference-collapsing rules makes implementation easy:

```
template<typename T>                    // For lvalues (T is T&),
T&& std::forward(T&& param)             // take/return lvalue refs.
{                                       // For rvalues (T is T),
  return static_cast<T&&>(param);       // take/return rvalue refs.
}
```

Real implementations more sophisticated; see Further Information.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 288

Production implementations of std::forward prevent misuse by disabling implicit argument deduction, thus forcing specification of T at the call site. That forces clients to write

```
std::forward<T>(param)
```

instead of

```
std::forward(param)
```

The latter expression would always return an lvalue, because param has a name.

The usual std::forward implementation is:

```
template<typename T>
struct identity {
  typedef T type;
};

template<typename T>
T&& forward(typename identity<T>::type&& param)
{ return static_cast<identity<T>::type&&>(param); }
```


© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding

- Applicable only to function templates.
- Preserves arguments' lvalueness/rvalueness/constness when forwarding them to other functions.
- Implemented via std::forward.

© 2011 Scott Meyers, all rights reserved.

Perfect forwarding isn't really perfect. There are several kinds of arguments that cannot be perfectly forwarded, including (but not necessarily limited to):

- 0 as a null pointer constant.
- Names of function templates (e.g., std::endl and other manipulators).
- Braced initializer lists.
- In-class initialized const static data members lacking an out-of-class definition.
- Bit fields.

For details consult the comp.std.c++ discussion, " Perfect Forwarding Failure Cases," referenced in the Further Information section of the course.

© 2011 Scott Meyers, all rights reserved.

default Member Functions

The "special" member functions are implicity generated if used:

- Default constructor
  - ➡ Only if no user-declared constructors.
- Destructor
- Copy operations (copy constructor, copy operator=)
  - ➡ Only if move operations not user-declared.
- Move operations (move constructor, move operator=)
  - ➡ Only if copy operations not user-declared.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

default Member Functions

Generated versions are:

- Public
- Inline
- Non-explicit

defaulted member functions have:

- **User-specified declarations with the usual compiler-generated implementations.**

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

default Member Functions

Typical use: "unsuppress" implicitly-generated functions:

```
class Widget {
public:
  Widget(const Widget&);              // copy ctor prevents implicitly-
                                      // declared default ctor and
                                      // move ops

  Widget() = default;                 // declare default ctor, use
                                      // default impl.

  Widget(Widget&&) = default;         // declare move ctor, use
                                      // default impl.
  ...
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

default Member Functions

Or change "normal" accessibility, explicitness, virtualness:

```
class Widget {
public:
  virtual ~Widget() = default;                    // declare as virtual

  explicit Widget(const Widget&) = default;       // declare as explicit

private:
  Widget& operator=(Widget&&) = default;          // declare as private
  ...
};
```


© 2011 Scott Meyers, all rights reserved.

Declaring a copy constructor explicit changes its behavior in odd ways, e.g., in the code above, functions would not be permitted to return Widget objects by value, nor would callers be allowed to bind rvalues to parameters of type const Widget&. I am unaware of any practical uses for explicit copy constructors.

The class on this page is strange in another way. The declaration of the copy constructor will suppress generation of the move operations, and the declaration of the move assignment operator will suppress generation of the copy operations. I do not know of any use for such a type.

© 2011 Scott Meyers, all rights reserved.

delete Functions

deleted functions are declared, but can't be used.

- Most common application: prevent object copying:

```
class Widget {
  Widget(const Widget&) = delete;                   // declare and
  Widget& operator=(const Widget&) = delete;        // make uncallable
  …
};
```

- Note that Widget isn't movable, either.
  - *Declaring* copy operations suppresses implicit move operations!
  - It works both ways:

```
class Gadget {
  Gadget(Gadget&&) = delete;                        // these also
  Gadget& operator=(Gadget&&) = delete;             // suppress copy
  …                                                 // operations
};
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 294

"=delete" functions can't be used in any way: they can't be called, can't have their address taken, can't be used in a sizeof expression, etc.

Template functions may be deleted. For example, this is how construction from rvalues is prevented for std::reference_wrappers (e.g., as returned from std::ref).

A virtual function may be deleted, but if it is, all base and derived versions of that virtual must also be deleted. That is, either all declarations of a virtual in a hierarchy are deleted or none are.

© 2011 Scott Meyers, all rights reserved.

delete Functions

Not limited to member functions.

- Another common application: control argument conversions.
  - ➡ deleted functions are declared, hence participate in overload resolution:

```
void f(void*);                  // f callable with any ptr type
void f(const char*) = delete;   // f uncallable with [const] char*

auto p1 = new std::list<int>;   // p1 is of type std::list<int>*
extern char *p2;
…
f(p1);                          // fine, calls f(void*)

f(p2);                          // error!  f(const char*) unavailable

f("Fahrvergnügen");             // error!

f(u"Fahrvergnügen");            // fine (char16_t* ≠ char*)
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Default Member Initialization

Default initializers for non-static data members may now be given:

```
class Widget {
private:
  int x = 5;
  std::string id = defaultID();
};

Widget w1;                          // w1.x initialized to 5,
                                    // w1.id initialized per defaultID.
```

Uniform initialization syntax is also allowed:

```
class Widget {                      // semantically identical to above
  …
private:
  int x {5};                        // "=" is not required,
  std::string id = {defaultID()};   // but is allowed
};

Widget w2;                          // same as above
```


© 2011 Scott Meyers, all rights reserved.

Direct initialization syntax (using parentheses) is not permitted for default member initialization.

Default member initialization values may depend on one another:

```
class Widget {
private:
  int x { 15 };
  int y { 2 * x };
  ...
};
```

Per N2756, everything valid as an initializer in a member initialization list should be valid as a default initializer. In particular, non-static member function calls are valid, e.g., in the initialization of Widget::id above, defaultID may be either a static or a non-static member function. If a non-static member function is used, there could be issues of referring to data members that have not yet been initialized.

In-class initialization of static data members continues to be valid only for const objects with static initializers (i.e., in-class dynamic initialization is not valid). However, all “literal” types – not just integral types – may be so initialized in C++0x. (Literal types are defined in [basic.types] (3.9/10 in N3290).)

© 2011 Scott Meyers, all rights reserved.

Default Member Initialization

Constructor initializer lists override defaults:

```
class Widget {
public:
  Widget() = default;
  explicit Widget(int xVal): x(xVal) {}
private:
  int x = 5;
  std::string id = defaultID();
};

Widget w3;                 // w3.x == 5, w3.id == defaultID()

Widget w4(-99);            // w4.x == -99, w4.id == defaultID()
```

Default member initialization most useful when initialization independent of constructor called.

- Eliminates redundant initialization code in constructors.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 297

Use of a default member initializer renders the class/struct a non-aggregate, so, e.g.:

```
struct Widget {
  int x = 5;
};
Widget w { 10 };          // error!  Attempt to call a constructor taking an int,
                          // but Widget has no such constructor
```


© 2011 Scott Meyers, all rights reserved.

Delegating Constructors

Consider a class with several constructors:

```
class Base {
public:
  explicit Base(int);
  …
};
class Widget: public Base {                    // 4 constructors
public:
  Widget();
  explicit Widget(double fl);
  explicit Widget(int sz);
  Widget(const Widget& w);
private:
  static int calcBaseVal();
  static const double defaultFlex = 1.5;
  const int size;
  long double flex;
};
```


© 2011 Scott Meyers, all rights reserved.

Java has delegating constructors.

Base's constructor in this (and subsequent) examples is explicit, just to show good default style. None of the examples depends on it.

© 2011 Scott Meyers, all rights reserved.

Delegating Constructors

Often, implementations include redundancy:

```
Widget::Widget()
: Base(calcBaseVal()), size(0), flex(defaultFlex)
{
  registerObject(this);
}
Widget::Widget(double fl)
: Base(calcBaseVal()), size(0), flex(fl)
{
  registerObject(this);
}
Widget::Widget(int sz)
: Base(calcBaseVal()), size(sz), flex(defaultFlex)
{
  registerObject(this);
}
Widget::Widget(const Widget& w)
: Base(calcBaseVal()), size(w.size), flex(w.flex)
{
  registerObject(this);
}
```


© 2011 Scott Meyers, all rights reserved.

The first two constructors are also redundant, in that they both contain "size(0)". This redundancy is removed in the forthcoming code using delegating constructors.

© 2011 Scott Meyers, all rights reserved.

Delegating Constructors

Delegating constructors call other constructors:

```
class Base { … };                                       // as before
class Widget: public Base {
public:
  Widget(): Widget(defaultFlex) {}                      // #1 (calls #2)
  explicit Widget(double fl): Widget(0, fl) {}          // #2 (calls #5)
  explicit Widget(int sz): Widget(sz, defaultFlex) {}   // #3 (calls #5)
  Widget(const Widget& w): Widget(w.size, w.flex) {}    // #4 (calls #5)
private:
  Widget(int sz, double fl)                             // #5 (this is new)
  : Base(calcBaseVal()), size(sz), flex(fl)
  { registerObject(this); }
  static int calcBaseVal();
  static const double defaultFlex = 1.5;
  const int size;
  long double flex;
};
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 300

A constructor that delegates to another constructor may not do anything else on its member initialization list.

A constructor that delegates to itself (directly or indirectly) yields an “ill-formed” program.

© 2011 Scott Meyers, all rights reserved.

Delegating Constructors

Delegation independent of constructor characteristics.

- Delegator and delegatee may each be inline, explicit, public/protected/private, etc.
- Delegatees can themselves delegate.
- Delegators' code bodies execute when delegatees return:

```
class Widget: public Base {
public:
  Widget(const Widget& w): Widget(w.size, w.flex)
  {
    makeLogEntry("Widget copy constructor");
  }
  ...
private:
  Widget(int sz, double fl);                  // as before
  ...
};
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Inheriting Constructors

using declarations can now be used with base class constructors:

```
class Base {
public:
  explicit Base(int);
  void f(int);
};

class Derived: public Base {
public:
  using Base::f;          // okay in C++98 and C++0x

  using Base::Base;       // okay in C++0x only; causes implicit
                          // declaration of Derived::Derived(int),
                          // which, if used, calls Base::Base(int)

  void f();               // overloads inherited Base::f
  Derived(int x, int y);  // overloads inherited Base ctor
};

Derived d1(44);           // okay in C++0x due to ctor inheritance

Derived d2(5, 10);        // normal use of Derived::Derived(int, int)
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 302

using declarations for constructors only declare inherited constructors, they don't define them. Such constructors are defined only if used.

If the derived class declares a constructor with the same signature as a base class constructor, that specific base class constructor is not inherited. This is the same rule for non-constructors.

Inherited constructors retain their exception specifications and whether they are explicit or constexpr.

© 2011 Scott Meyers, all rights reserved.

Inheriting Constructors

"Inheritance" ⇒ new implicit constructors calling base class versions.

- The resulting code must be valid.

```
class Base {
private:
  explicit Base(int);
};

class Derived: public Base {
public:
  using Base::Base;
};

Derived d(10);               // error! calls Derived(int), which calls
                             // Base(int), which is private
```


© 2011 Scott Meyers, all rights reserved.

The error is diagnosed at the point of use of the inheriting constructor (i.e., the declaration of d).

© 2011 Scott Meyers, all rights reserved.

Inheriting Constructors

Inheriting constructors into classes with data members risky:

```
class Base {
public:
  explicit Base(int);
};

class Derived: public Base {
public:
  using Base::Base;

private:
  std::u16string name;
  int x, y;
};


Derived d(10);                  // compiles, but d.name is
                                // default-initialized, and
                                // d.x and d.y are uninitialized
```


© 2011 Scott Meyers, all rights reserved.

It's not quite true that Derived::x and Derived::y are uninitialed. Rather, they are treated as if they are not mentioned on the member initialization list of the inherited constructor. If the Derived object is of static or thread storage duration, its x and y data members would be initialized to zero.

© 2011 Scott Meyers, all rights reserved.

Inheriting Constructors

Default member initializers can mitigate the risk:

```
class Base { … };                           // as before

class Derived: public Base {
public:
  using Base::Base;

private:
  std::u16string name = "Uninitialized";
  int x = 0, y = 0;
};

Derived d(10);                             // d.name == "Uninitialized",
                                           // d.x == d.y == 0
```


© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Summary of Features for Class Authors

- Rvalue references facilitate move semantics and perfect forwarding.
- =default yields default body for user-declared special functions.
- =delete makes declared functions unusable.
- All data members may have default initialization values.
- Delegating constructors call other constructors.
- Inherited constructors come from base classes.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Overview

- Introduction
- Features for Everybody
- Library Enhancements
- Features for Class Authors
- **Features for Library Authors**
- Yet More Features
- Further Information

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Static Assertions

Generate user-defined diagnostics when compile-time tests fail:

```
static_assert(sizeof(void*)==sizeof(long),
              "Pointers and longs are different sizes");
```

Valid anywhere a declaration is:

- Global/namespace scope.
- Class scope.
- Function/block scope.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Static Assertions

Especially useful with templates:

```
template<typename T>
void f(const T& obj)
{
  static_assert(std::is_base_of<Widget,T>::value,
                "T doesn't inherit from Widget");
  …
}

template<short val>
class Gadget {
  static_assert(1 <= val && val <= 10,
                "val out of range (must be 1-10)");
  …
};
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Static Assertions

Diagnostics may be any kind of string literal:

```
static_assert(CHAR_BIT == 8,                                  // ordinary
              "chars don't have 8 bits \u2620");              // string
static_assert(CHAR_BIT == 8,
              L"chars don't have 8 bits \u2620");             // wide string
static_assert(CHAR_BIT == 8,
              u8"chars don't have 8 bits \u2620");            // UTF-8
static_assert(CHAR_BIT == 8,
              u"chars don't have 8 bits \u2620");             // UTF-16
static_assert(CHAR_BIT == 8,
              U"chars don't have 8 bits \u2620");             // UTF-32
```

Raw string literals are also valid.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 310

Code point 2620 is the skull and crossbones symbol (☠).

When a code point specified via an escape sequence is part of a narrow string literal (e.g., the first example on this page), the resulting string literal contains as many bytes as is needed for that code point. So if representing \2620 requires 2 bytes, 2 bytes will be included as part of the narrow string literal.

Some static_assert conditions are so self-explanatory, it may be desirable to use them as the diagnostic message, i.e., to default the diagnostic to being the text of the condition. Such behavior can be offered via a suitable macro:

```
#define STATIC_ASSERT(condition) static_assert(condition, #condition)
```


© 2011 Scott Meyers, all rights reserved.

explicit Conversion Functions

explicit now applicable to conversion functions:

```
class Widget {
public:
  explicit Widget(int i);                    // C++98 and C++0x
  …
  explicit operator std::string() const;     // C++0x only
};
```

Behavior analogous to that of constructors:

```
void fw(const Widget& w);
int i;
…
fw(i);                                       // error!
fw(static_cast<Widget>(i));                  // okay

void fs(const std::string& s);
Widget w;
…
fs(w);                                       // error!
fs(static_cast<std::string>(w));             // okay
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 311

This slide shows uses of static_cast, but other cast syntaxes (i.e, C-style and functions-style) would behave the same way.

© 2011 Scott Meyers, all rights reserved.

explicit Conversion Functions

explicit operator bool functions treated specially.

- Implicit use okay when "safe"(i.e., in "contextual conversions"):

```
template<typename T>
class SmartPtr {
public:
  …
  explicit operator bool() const;
};

SmartPtr<std::string> ps;

if (!ps) …                                      // okay

long len = ps ? ps->size() : -1;                // okay

SmartPtr<Widget> pw;

if (ps == pw) …                                 // error!

int i = ps;                                     // error!
```


© 2011 Scott Meyers, all rights reserved.

The "explicitness" of an operator bool function is ignored in cases where the standard calls for something being "contextually converted" to bool.

© 2011 Scott Meyers, all rights reserved.

Variadic Templates

Templates may now take arbitrary numbers and types of parameters:

```
template <class... Types>                   // std::tuple is in C++0x
class tuple;

template<class T, class... Args>            // std::make_shared is
shared_ptr<T>                               // in C++0x
  make_shared(Args&&... params);
```

Non-type parameters also okay:

```
template<typename T, std::size_t... Dims>   // this template is
class MultiDimensionalArray;                // not in C++0x
```

Whitespace around "**...**" not significant:

```
template <class ... Types>                  // Same meaning as
class tuple;                                // above

template<class T, class ...Args>            // Ditto
shared_ptr<T>
  make_shared(Args&&... params);
```


© 2011 Scott Meyers, all rights reserved.

The declarations for tuple and make_shared are copied from draft C++0x, which is why they use "class" instead of my preferred "typename" for template type parameters. In C++0x, the function parameter pack is named "args", but I've renamed it to "params" to make it easier to distinguish orally from the template parameter "Args" (which is in draft C++0x).

© 2011 Scott Meyers, all rights reserved.

Parameter Packs

Two kinds:

- **Template**: hold variadic template parameters.
- **Function**: hold corresponding function parameters.

```
template <class... Types>                    // template param. pack
class tuple { … };

template<class T, class... Args>             // template param. pack
shared_ptr<T>
  make_shared(Args&&... params);             // function param. pack

std::tuple<int, int, std::string> t1;        // Types = int, int, std::string

auto p1 = std::make_shared<Widget>(10);      // Args/params = int/int&&

int x;
const std::string s("Variadic Fun");
…
auto p2 = std::make_shared<Widget>(x, s);    // Args/params =
                                             //   int&, const std::string&/
                                             //   int&, const std::string&
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 314

A function parameter pack declaration is a function parameter declaration containing a template parameter pack expansion. It must be the last parameter in the function parameter list.

Class templates may have at most one parameter pack, which must be at the end of the template parameter list, but function templates, thanks to template argument type deduction, may have multiple parameter packs, e.g. (from draft C++0x),

```
template<class... TTypes, class... UTypes>
bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```


© 2011 Scott Meyers, all rights reserved.

Parameter Packs

Manipulation based on recursive "first"/"rest" manipulation:

- Primary operation is **unpack** via **...**:

```
template<typename... Types>                  // declare list-
struct Count;                                // walking template

template<typename T, typename... Rest>       // walk list
struct Count<T, Rest...> {
  const static int value = Count<Rest...>::value +1;
};

template<> struct Count<>                    // recognize end of
{                                            // list
  const static int value = 0;
};

auto count1 = Count<int, double, char>::value;  // count1 = 3

auto count2 = Count<>::value;                 // count2 = 0
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Parameter Packs

Count purely an exercise; C++0x's sizeof**...** does the same thing:

```
template<typename… Types>
struct VerifyCount {
  static_assert(Count<Types…>::value == sizeof...(Types),
                "Count<T>::value != sizeof…(T)");
};
```

Unpack (**...**) and sizeof**...** only two operations for parameter packs.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Variadic Function Templates

Example: type-safe printing of arbitrary objects:

```
void print() { std::cout << '\n'; };          // print 0 objects

template<typename T,                          // type of 1st object
         typename... TRest>                   // types of the rest
void print(const T& obj,                      // 1st object
           const TRest&... rest)              // the rest of them
{
  std::cout << obj << " ";                    // print 1st object
  print(rest...);                             // print the rest
}

double p = 3.14;
std::string s("Vari");

print(-22, p, &p, s, "adic");          // -22 3.14 0x22ff40 Vari adic
```

- Gregor's/Järvi's article shows a compile-time-checked printf.
  - ➡ Ensures format string consistent with passed arguments.

© 2011 Scott Meyers, all rights reserved.

This example passes everything by const T&, but perfect forwarding would probably be a better approach.

© 2011 Scott Meyers, all rights reserved.

Unpacking Patterns

Unpacking uses the **pattern** of the expression being unpacked:

```
template<class T, class... Args>
shared_ptr<T>                                    // add "&&" to each
  make_shared(Args&&... params);                 // unpacked elem.

template<typename T,  typename... TRest>         // add "const" and
void print(const T& obj, const TRest&... rest)   // "&" to each
{
  std::cout << obj << " ";
  print(rest...);                                // add nothing to
}                                                // each elem's name
```

Call to print expands to:

print(rest$_1$, rest$_2$, ..., rest$_n$);

© 2011 Scott Meyers, all rights reserved.

The ellipsis is always at the end of the pattern.

© 2011 Scott Meyers, all rights reserved.

Unpacking Patterns

```
template<typename T>                          // return a normalized
T normalize(T&& obj);                         // copy of obj

template<typename F, typename… PTypes>
void normalizedCall( F func,
                     PTypes&&… params)        // as before
{
  func(normalize(params)…);                   // call normalize on
}                                             // each unpacked elem
```

Call to func expands to:

func(normalize(params$_1$), normalize(params$_2$), ..., normalize(params$_n$));

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Variadic Class Templates

Foundational for TMP (template metaprogramming). Examples:

- Numerical computations similar to Count:
  - Max size of types in a list (e.g., for a discriminated union).
- Type computations:
  - <type_traits> has template <class... T> struct common_type;
- Object structure generation:
  - std::tuple<T_1, T_2, …, T_n> needs n fields, each of correct type.
    - std::tuple<T_1, T_2, …, T_n> inherits from std::tuple<T_2, T_3,…, T_n>

© 2011 Scott Meyers, all rights reserved.

Given a list of types, std::common_type returns the type in the list to which all types in the list can be converted. If there is no such type, or if there is more than one such type, the code won't compile. For built-in types, the usual promotion and conversion rules apply in their usual order, so, e.g., std::common_type<int, double>::type is double, because int→double is preferable to double→int, although both are possible.

© 2011 Scott Meyers, all rights reserved.

Sketch of std::tuple

```
template <class... Types>                 // declare primary template
class tuple;

template<> class tuple<>{};               // for empty tuples

template<typename T,                      // class with data member
         typename… TRest>                 // for 1st T in pack
class tuple<T, TRest…>:                   // inherits from class for
  private tuple<TRest…> {                 // rest of pack
private:
  T data;                                 // data member of type T

public:
  tuple()                                 // default ctor; all types
  :  data() {};                           // must be default-
                                          // constructible

  …                                       // non-default ctors, etc.
};
```


© 2011 Scott Meyers, all rights reserved.

Doing "data()" on the member initialization line ensure that built-in types are initialized to zero (and pointers to null).

The implementation published by Douglas Gregor and Jaakko Järvi (see Further Information section) declares data protected, but no justification is given, and real implementations (e.g., in VC 10, gcc 4.5) declare it private. Hence my use of private here.

© 2011 Scott Meyers, all rights reserved.

Generated Hierarchy

```
std::tuple<int, char, std::string> t;
```

```
class tuple<> {};
```

```
class tuple<std::string>:
  private tuple<> {
private:
  std::string data;
};
```

```
class tuple<char, std::string>:
  private tuple<std::string> {
private:
  char data;
};
```

```
class tuple<int, char, std::string>:
  private tuple<char, std::string> {
private:
  int data;
};
```


© 2011 Scott Meyers, all rights reserved.

The fact that data is private raises the question of how std::get is implemented. Among tuple member functions not listed in this example is a public one returning a reference to data. In Gregor's and Järvi's implementation, this function is called head. std::get<n> on a tuple<T, TRest...> recursively walks up TRest, decreasing n at leach level until it's 0. It then returns the result of head for that class.

© 2011 Scott Meyers, all rights reserved.

decltype

Yields the type of an expression without evaluating it.

```
int x, *ptr;

decltype(x) i1;                              // i1's type is int
decltype(ptr) p1;                            // p1's type is int*

std::size_t sz = sizeof(decltype(ptr[44]));  // sz = sizeof(int);
                                             // ptr[44] not evaluated
```

Fairly intuitive, but some quirks, e.g., parentheses can matter:

```
struct S { double d; };
const S* p;
...
decltype(p->d) x1;                           // double

decltype((p->d)) x2;                         // const double&
```

- Quirks rarely relevant (and can be looked up when necessary).

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

decltype

Primary use: template return types that depend on parameter types.

- Common for forwarding templates:

```
template<typename F, typename… Ts>                // logAndInvoke
auto logAndInvoke(std::ostream& os,               // returns what
                  F&& func, Ts&&… args) ->        // func(args...) does.
  decltype(func(args…))                           // not quite right
{
  os << std::chrono::system_clock::now();         // from new time lib
  return func(args…);                             // not quite right
}
```

- Also in math-related libraries:

```
template<typename T1, typename T2>                // mult's return type
auto mult(T1&& a, T2&& b) ->                      // is same as a*b's.
  decltype(a * b)                                 // not quite right
{
  return a * b;                                   // not quite right
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 324

For the code on this page to be correct, we need to add uses of std::forward in various places. Hence the comments that say "not quite right". The correct code is shown shortly.

There is no operator<< for std::chrono::time_point objects (the return type from std::chrono::system_clock::now) in the standard library, so the statement involving std::chrono::system_clock::now will not compile unless such an operator<< has been explicitly declared.

© 2011 Scott Meyers, all rights reserved.

The Forwarding Problem

```
template<typename F, typename... Ts>          // as before
auto logAndInvoke(std::ostream& os,
                  F&& func, Ts&&... args) ->
  decltype(func(args...))                     // not quite right
{
  os << std::chrono::system_clock::now();
  return func(args...);                       // not quite right
}
```

args... are lvalues, but logAndInvoke's caller may have passed rvalues:

- Templates can distinguish rvalues from lvalues.
- logAndInvoke might call the wrong overload of func.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

The Forwarding Problem

Example:

```
class FontProcessor {
public:
  void operator()(const Font&);       // takes lvalue
  void operator()(Font&&);            // takes rvalue
};

Font getFont();                       // function returning rvalue

logAndInvoke(std::cout,
             FontProcessor(),
             getFont());              // caller passes rvalue, but
                                      // logAndInvoke calls
                                      // FontProcessor::operator()
                                      // taking lvalue
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding Redux

Solution is perfect forwarding:

```
template<typename F, typename… Ts>                    // return type is
auto logAndInvoke(std::ostream& os,                   // same as func's
                  F&& func, Ts&&… args) ->            // on original args
  decltype(func(std::forward<Ts>(args)…))
{
  os << std::chrono::system_clock::now();
  return func(std::forward<Ts>(args)...);
}
```


© 2011 Scott Meyers, all rights reserved.

In the expression "std::forward<Ts>(args)…", the pattern being unpacked is "std::forward<Ts>(args)", so "std::forward<Ts>(args)…" is equivalent to "std::forward<Ts_1>($args_1$), std::forward<Ts_2>($args_2$), … , std::forward<Ts_n>($args_n$)". This is a parameter pack pattern that involves the simultaneous unpacking of two parameter packs: one from the template parameter list (Ts) and one from the function parameter list (args).

© 2011 Scott Meyers, all rights reserved.

Perfect Forwarding Redux

A correct version of mult:

```
template<typename T1, typename T2>
auto mult(T1&& a, T2&& b) ->
  decltype(std::forward<T1>(a) * std::forward<T2>(b))
{
  return std::forward<T1>(a) * std::forward<T2>(b);
}
```


© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

decltype vs. auto

To declare objects, decltype can replace auto, but more verbosely:

```
std::vector<std::string> vs;
…
auto i = vs.begin();
decltype(vs.begin()) i = vs.begin();
```

Only decltype solves the template-function-return-type problem.

auto is for everybody. decltype is primarily for template authors.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Summary of Features for Library Authors

- static_assert checks its condition during compilation.
- explicit conversion functions restrict their implicit application.
- Variadic templates accept an unlimited number of arguments.
- decltype helps declare template functions whose return type depends on parameter types.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Overview

- Introduction
- Features for Everybody
- Library Enhancements
- Features for Class Authors
- Features for Library Authors
- **Yet More Features**
- Further Information

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

More C++0x Features

- Enum enhancements:
  - Forward declaration
  - Specification of underlying type
  - Enumerant names scoped to the enum
  - No implicit conversion to `int`
- Unrestricted unions (members may be any non-reference type).
- Time library supportings clocks, durations, points in time.
- Local types allowed as template arguments.
- C99 compatibility, e.g., `long long`, `__func__`, etc.
- Inline namespaces facilitate library versioning.
- Scoped allocators allow containers and their elements to use different allocators, e.g., `vector<string>`.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2011 Scott Meyers, all rights reserved.

Slide 332

The primary motivation for the time library was to be able to specify timeouts for the concurrency API (e.g., sleep durations, timeouts for lock acquisition, etc.).

© 2011 Scott Meyers, all rights reserved.

Still More C++0x Features

- Generalized constant expressions (constexpr).
- User-defined literals (e.g., 10_km, 30_sec).
- Relaxed POD type definition; new standard layout types.
- extern templates for control of implicit template instantiation.
- sizeof applicable to class data members alone (e.g., sizeof(C::m)).
- & and && member functions.
- Relaxed rules for in-class initialization of static data members.
- Contextual keywords for alignment control and constraining virtual function overrides.
- Attributes express special optimization opportunities and provide a standard syntax for platform-specific extensions.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Removed and Deprecated Features

- auto as a storage class has been removed.
- export as a language feature has been removed.
  - export remains a keyword (with no semantics).
- register as a storage class has been deprecated.
- Exception specifications have been deprecated.
  - noexcept conceptually replaces the "throw()" specification.
- auto_ptr is deprecated. (Use unique_ptr instead.)
- bind1st/bind2nd are deprecated. (Use bind or lambdas instead.)

© 2011 Scott Meyers, all rights reserved.

If an exception attempts to propagate beyond a noexcept(true) function, terminate is called. This is different from what happens in C++03 if a "throw()" specifier is violated. In that case, unexpected is invoked after the stack has been unwound.

From Herb Sutter's 8 December 2010 blog post, "Trip Report: November 2010 C++ Standards Meeting:" "Destructor and delete operators [are] noexcept by default. ... Briefly, every destructor will be noexcept by default unless a member or base destructor is noexcept(false); you can of course still explicitly override the default and write noexcept(false) on any destructor. "

Herb Sutter argues that the primary advantage of noexcept over throw() is that noexcept offers compilers additional optimization opportunities. From a 30 March 2010 comp.std.c++ posting: "noexcept enables optimizations not only in the caller but also in the callees, so that the optimizer can assume that functions called in a noexcept function and not wrapped in a try/catch are themselves noexcept without being declared such (e.g., C standard library functions are not so annotated). "

© 2011 Scott Meyers, all rights reserved.

Overview

- Introduction
- Features for Everybody
- Library Enhancements
- Features for Class Authors
- Features for Library Authors
- Yet More Features
- **Further Information**

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

FDIS for C++0x:

- *Programming Languages — C++*, Pete Becker (Ed.), 2011-04-11, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3290.pdf.

C++0x in General:

- "C++0x," *Wikipedia*.
- *C++0x - the next ISO C++ standard*, Bjarne Stroustrup, http://www.research.att.com/~bs/C++0xFAQ.html.
- "C++0X: The New Face of Standard C++," Danny Kalev, *informIT.com*, http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=216.
  - ➨ Click "Guide Contents," scroll to C++0X section, select topic.

© 2011 Scott Meyers, all rights reserved.

Many sources listed in this section have no URLs, because they are easy to find via search engine. The fewer URLs I publish, the fewer will be broken when target sites reorganize.

© 2011 Scott Meyers, all rights reserved.

Further Information

C++0x in General:

- "C++0X with Scott Meyers," *Software Engineering Radio* (Audio), 5 April 2010.
- "An Overview of the Coming C++ (C++0x) Standard," Matt Austern and Lawrence Crowl, *Google Tech Talk* (Video), 31 October 2008.
- "Overview: C++ Gets an Overhaul," Danny Kalev, *DevX.com*, 18 August 2008.
- "C++0x language feature checklist," VC10 Slow Chat, *CodeGuru.com*, December 2008, http://www.codeguru.com/forum/showthread.php?t=4668939.
  - ➡ Discusses C++0x support in VC10.
- "Everything you ever wanted to know about nullptr," Stephan T. Lavavej, *Channel 9* (Video), 19 October 2009.
  - ➡ Also discusses NULL, make_shared, perfect forwarding, auto.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

C++0x in General:

- "Standard Library Changes in C++0x," *Van's House*, 12 January 2009, http://blogs.msdn.com/xiangfan/archive/2009/01/12/standard-library-changes-in-c-0x.aspx.
- "The State of the Language: An Interview with Bjarne Stroustrup," Danny Kalev, *DevX.com*, 15 August 2008.
- *Summary of C++0x Feature Availability in gcc and MSVC*, Scott Meyers, http://www.aristeia.com/C++0x/C++0xFeatureAvailability.htm.
  - ➡ Includes links to summaries for other compilers.
- "C++0x: Ausblick auf den neuen C++-Standard," Bernhard Merkle, *heise Developer*, 11 November 2008.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

auto:

- "Lambdas, auto, and static_assert: C++0x Features in VC10, Part 1," Stephan T. Lavavej, *Visual C++ Team Blog*, 28 October 2008.
- *C++ Templates*, David Vandevoorde and Nicolai M. Josuttis, Addison-Wesley, 2003, chapter 11, "Template Argument Deduction."
  - Covers type deduction rules for templates and auto.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Unicode Support:

- *Pete Becker's Roundhouse Consulting Bits and Pieces*, http://www.versatilecoding.com/bitsandpieces.html.
- "Prepare Yourself for the Unicode Revolution," Danny Kalev, *DevX.com*, 12 April 2007.
- *The GNU C++ Library Documentation (section on* `codecvt`*)*, http://gcc.gnu.org/onlinedocs/libstdc++/manual/codecvt.html.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Further Information

Uniform Initialization Syntax:

- "Uniform and Convenient Initialization Syntax," Danny Kalev, *DevX.com*, 13 November 2008.
- "C++0x Initialization Lists," Bjarne Stroustrup, *Google Tech Talk* (Video), 21 February 2007.
- "Sequence Constructors Add C++09 Initialization Syntax to Your Homemade Classes," Danny Kalev, *DevX.com*, 12 March 2009.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Lambda Expressions:

- "Lambda Functions are Ready for Prime Time," Danny Kalev, *DevX.com*, 16 January 2009.
- "Lambdas, auto, and static_assert: C++0x Features in VC10, Part 1," Stephan T. Lavavej, *Visual C++ Team Blog*, 28 October 2008.
- "GCC C++0x Features Exploration," Dean Michael Berris, *C++ Soup!*, 15 March 2009.
- "Lambdas, Lambdas Everywhere," Herb Sutter, presentation at PDC10, 29 October 2010, http://channel9.msdn.com/Events/PDC/PDC10/FT13.
  - ➡ Shows many ways (some unconventional) to use lambdas.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Concurrency:

- *C++ Concurrency in Action*, Anthony Williams, Manning Publications, Summer 2011 (anticipated).
  - ➡ Draft edition available at http://www.manning.com/williams/.
- "Simpler Multithreading in C++0x," Anthony Williams, *DevX.com*, 13 December 2007.
- *Just Software Solutions Blog*, Anthony Williams, http://www.justsoftwaresolutions.co.uk/blog/.
  - ➡ Includes several "Multithreading in C++0x" tutorial posts.
- "Broken promises – C++0x futures," Bartosz Milewski, *Bartosz Milewski's Programming Café*, 3 March 2009.
- "Thanks for Not Sharing, Or, How to Define Thread-Local Data," Danny Kalev, *DevX.com*, 13 March 2008.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Concurrency:

- "Getting C++ Threads Right," Hans Boehm, *Google Tech Talk* (Video), 12 December 2007.
- "Designing Multithreaded Programs in C++," Anthony Williams, *ACCU 2009*, 28 April 2009.
  - Slides from a conference presentation.
- "Concurrency in the Real World," Anthony Williams, *ACCU 2010*, 14 April 2010.
  - Slides from a conference presentation.
- "Implementing Dekker's algorithm with Fences," Anthony Williams, *Just Software Solutions* (blog), 27 July 2010.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Overviews of TR1:

- *Scott Meyers' TR1 Information Page*, http://www.aristeia.com/EC3E/TR1_info.html.
  - Includes links to proposal documents.
- *The C++ Standard Library Extensions*, Pete Becker, Addison-Wesley, 2007, ISBN 0-321-41299-0.
  - A comprehensive reference for TR1.
- "The Technical Report on C++ Library Extensions," Matthew H. Austern, *Dr. Dobb's Journal*, June 2005.
- "The New C++ Not-So-Standard Library," Pete Becker, *C/C++ Users Journal*, June 2005.

Boost:

- *Boost C++ Libraries*, http://www.boost.org/.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

shared_ptr and weak_ptr (TR1 Versions):

- "Bad Pointers," Pete Becker, *C/C++ Users Journal*, Sept. 2005.
- "More Bad Pointers," Pete Becker, *C/C++ Users Journal*, Oct. 2005.
- "Weak Pointers," Pete Becker, *C/C++ Users Journal*, Nov. 2005.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  - Describes tr1::shared_ptr and uses it throughout the book.
  - TOC is attached.
- *Smart Pointer Timings*, http://www.boost.org/libs/smart_ptr/smarttests.htm.
  - Compares performance of 5 possible implementations.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

unique_ptr:

- "Who's the Smartest of 'Em All? Get to Know std::unique_ptr," Danny Kalev, *DevX.com*, 11 September 2008.

Pointer containers:

- Not in C++0x. An alternative to containers of smart pointers.
- *Boost.PointerContainer Documentation*, Thorsten Ottosen, http://www.boost.org/libs/ptr_container/.
- "Pointer Containers," Thorsten Ottosen, *Dr. Dobbs Journal*, October 2005.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Hash Tables (unordered_ containers) (TR1 Versions):

- "STL and TR1: Part III," Pete Becker, *C/C++ Users Journal*, February 2006.
- "Hash Tables for the Standard Library," Matt Austern, *C/C++ Users Journal Experts Forum*, April 2002.

Regular Expressions (regex) (TR1 Version):

- "Regular Expressions," Pete Becker, *Dr. Dobb's Journal*, May 2006.
- "A TR1 Tutorial: Regular Expressions," Marius Bancila, *CodeGuru.com*, 2 July 2008.
- *Boost.RegEx Documentation*, John Maddock, http://www.boost.org/libs/regex/doc/index.html.
  - ➡ Describes Boost's RE library, on which TR1's is based.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Further Information

Tuples (TR1 Versions):

- "The Header <tuple>," Pete Becker, *C/C++ Users Journal*, July 2005.
- "GCC C++0x Features Exploration," Dean Michael Berris, *C++ Soup!*, 15 March 2009.
  - Shows use of tuple.

Fixed-Size Arrays (array):

- "std::array: The Secure, Convenient Option for Fixed-Sized Sequences," Danny Kalev, *DevX.com*, 11 June 2009.
- "STL and TR1: Part II," Pete Becker, *C/C++ Users Journal*, January 2006.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Further Information

Generalized Function Pointers (function) (TR1 Versions):

- "Generalized Function Pointers," Herb Sutter, *C/C++ Users Journal,* August 2003.
- "Generalizing Observer," Herb Sutter, *C/C++ Users Journal Experts Forum*, September 2003.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  - Item 35 explains and demonstrates use of tr1::function.
  - The TOC is attached.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

From TR1 to C++0x:

- "Improvements to TR1's Facility for Random Number Generation," Walter E. Brown *et al.*, 23 February 2006, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1933.pdf.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Rvalue References and Move Semantics:

- "A Brief Introduction to Rvalue References," Howard E. Hinnant *et al.*, *The C++ Source*, 10 March 2008.
  - ➨ Details somewhat outdated.
- *C++ Rvalue References Explained*, Thomas Becker, June 2009, http://thbecker.net/articles/rvalue_references/section_01.html.
  - ➨ Good explanations of std::move/std::forward implementations.
- "Rvalue References: C++0x Features in VC10, Part 2," Stephan T. Lavavej, *Visual C++ Team Blog*, 3 February 2009.
- "GCC C++0x Features Exploration," Dean Michael Berris, *C++ Soup!*, 15 March 2009.
- "Howard's STL / Move Semantics Benchmark," Howard Hinnant, *C++Next*, 13 October 2010.
  - ➨ Move-based speedup for std::vector<std::set<int>> w/gcc 4.0.1.
  - ➨ Reader comments give data for newer gccs, other compilers.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Rvalue References and Move Semantics:

- "Making Your Next Move," Dave Abrahams, *C++Next*, 17 September 2009.
- "Your Next Assignment…," Dave Abrahams, *C++Next*, 28 September 2009.
  - ➡ Correctness and performance issues for move `operator=`s.
- "Exceptionally Moving!," Dave Abrahams, *C++Next*, 5 Oct. 2009.
  - ➡ Exception safety issues for move operations.
- "To move or not to move," Bjarne Stroustrup, *Document N3174 to the C++ Standardization Committee*, 17 October 2010.
  - ➡ Describes rules governing implicit move operations.
- "Class invariants and implicit move constructors (C++0x)," comp.lang.c++ discussion initiated 14 August 2010.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Further Information

Perfect Forwarding:

- "Onward, Forward!," Dave Abrahams, *C++Next*, 7 Dec. 2009.
- "Perfect Forwarding Failure Cases," comp.std.c++ discussion initiated 16 January 2010.
  - ➡ Discusses argument types that can't be perfectly forwarded.

© 2011 Scott Meyers, all rights reserved.

© 2011 Scott Meyers, all rights reserved.

Further Information

Explicit conversion functions:

- "Use Explicit Conversion Functions to Avert Reckless Implicit Conversions," Danny Kalev, *DevX.com*, 9 October 2008.

Variadic Templates:

- "Variadic Templates for C++0x," Douglas Gregor and Jaakko Järvi, *Journal of Object Technology*, February 2008.
- "An Introduction to Variadic Templates in C++0x," Anthony Williams, *DevX.com*, 24 April 2009.
- "Variadic Templates," *BoostCon Wiki*, http://tinyurl.com/mtpa66.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

decltype:

- "decltype: C++0x Features in VC10, Part 3," Stephan T. Lavavej, *Visual C++ Team Blog*, 22 April 2009.
- "Clean up Function Syntax Mess with decltype," Danny Kalev, *DevX.com*, 8 May 2008.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Further Information

Other Features:

- "C++0x Forward Enum Declarations Cut Down Compilation Time and Dependencies," Danny Kalev, *DevX.com*, 13 August 2009.
- "C++09 Attributes: Specify Your Constructs' Unusual Properties," Danny Kalev, *DevX.com*, 14 May 2009.
- "Overriding Virtual Functions? Use C++0x Attributes to Avoid Bugs," Danny Kalev, *DevX.com*, 12 November 2009.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Acknowledgements

Many, *many*, thanks to my pre-release materials' reviewers:

- Stephan T. Lavavej
- Bernhard Merkle
- Stanley Friesen
- Leor Zolman
- Hendrik Schober

Anthony Williams contributed C++0x concurrency expertise and use of his just::thread C++0x threading library.

- Library web site: http://www.stdthread.co.uk/
  - ➡ Use http://www.stdthread.co.uk/sm20/ for a **20% discount**.

Participants in comp.std.c++ provided invaluable information and illuminating discussions.

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant

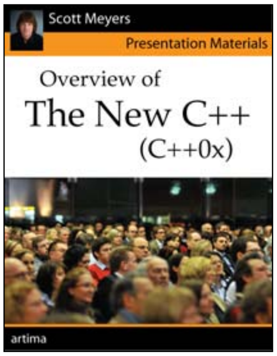
© 2011 Scott Meyers, all rights reserved.

Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:



© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

About Scott Meyers

Scott is a trainer and consultant on the design and implementation of C++ software systems. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

© 2011 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
© 2011 Scott Meyers, all rights reserved.

Contents

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

 © 2011 Scott Meyers, all rights reserved.

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

 © 2011 Scott Meyers, all rights reserved.