Scott Meyers

Presentation Materials

# Overview of The New C++ (C++11/14)

artima

Overview of the New C++ (C++11/14)

Artima Press is an imprint of Artima, Inc.
2070 N Broadway #305, Walnut Creek, California 94597



First version published April 5, 2010
Second version published May 6, 2010
Third version published April 24, 2011
Fourth version published October 25, 2011
Fifth version published January 6, 2012
Sixth version published December 12, 2012
This version published July 2, 2013
Produced in the United States of America

Cover photo by Stephan Jockel. Used with permission.

# Overview of the New C++ (C++11/14)

**Scott Meyers, Ph.D.**
Software Development Consultant
http://aristeia.com
smeyers@aristeia.com


**Last Revised: 7/2/13**

These are the official notes for Scott Meyers' training course, "Overview of the New C++ (C++11/14)". The course description is at http://www.aristeia.com/C++11.html. Licensing information is at http://aristeia.com/Licensing/licensing.html.

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

References to specific parts of the C++11 and C++14 standards give section numbers and, following a slash, paragraph numbers. Hence 3.9.1/5 refers to paragraph 5 of section 3.9.1.

In these notes, references to numbered documents preceded by N (e.g., N2973) are references to C++ standardization documents. Such documents are available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/.

[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 concurrency API. Day 2 usually goes to the end of the library material.]

# Overview

- **Introduction**
  - History, vocabulary, quick C++98/C++11 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, alignment control, etc.
- **Yet More Features**
- **Removed and Deprecated Features**
- **Further Information**





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.

# 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.

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

- 14 likely new components for the standard library.

2009: Selected "C++0x" features became commonly availabile.

2011: C++0x ratified ⇒ "C++11".

- 1353 pages.

2013: Full C++14 draft adopted.

- 1374 pages.

2014: Revised C++ Standard (minor).

2017?: Revised C++ Standard (major).





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

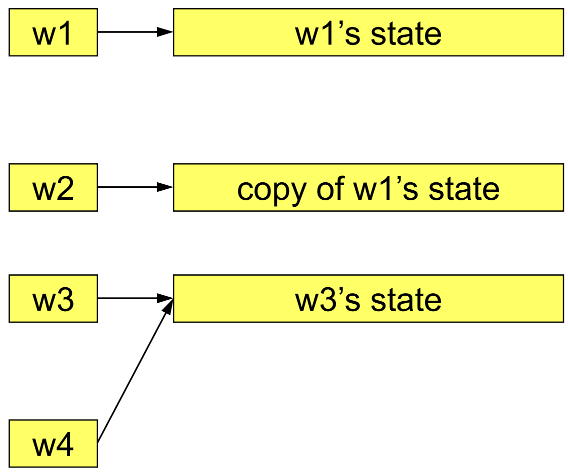
# Copying vs. Moving

C++ has always supported copying object state:

- *Copy* constructors, *copy* assignment operators

C++11 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.



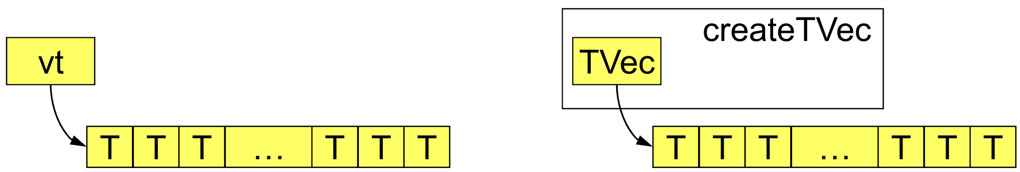


The diagrams on this slide make up a PowerPoint animation.

# 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
```







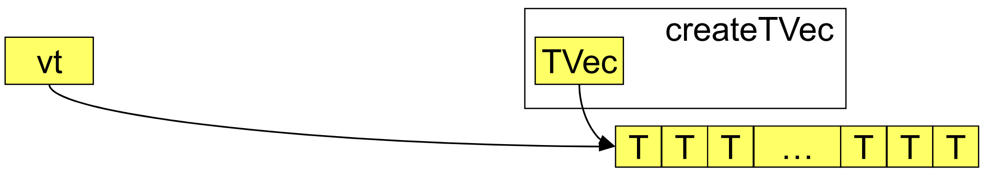

The diagrams on this slide make up a PowerPoint animation.

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.

# Copying vs. Moving

C++11 generally turns such copy operations into move requests:

```
TVec vt;
…
vt = createTVec();                    // implicit move request in C++11
```








The diagrams on this slide make up a PowerPoint animation.

C++11 "generally" turns copy operations on rvalues into move operations, but not always. Some operations (e.g., std::vector::push_back) offer the strong exception-safety guarantee, so moving can replace copying only if the move operations are known not to throw (e.g., by declaring them noexcept). Moving a container (such as in the example on this slide) requires that the container's allocator be movable, which need not be the case. If the allocator is not movable, the elements of the container must be individually copied, unless the element type's move constructor is known not to throw, in which case they may be moved.

# Copying vs. Moving

*Move semantics* examined in detail later, but:

- **Moving a key new C++11 idea.**
  - Usually an optimization of copying.
- Most standard types in C++11 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.





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

# Sample C++98 vs. C++11 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
```





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

## 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;
}
```





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++11), so to keep this example program simple and focused on C++11 features, we assume that every passed file name is legitimate, i.e., we embrace the "nothing could possibly go wrong" assumption.

# 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);
  }
}
```

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

Slide 10

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 :-)

# 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 wordCountInfoForFile =   // copy map returned by
      wordsInFile(argv[argNum]);                    // wordsInFile (modulo
                                                    // compiler optimization)

    for ( WordCountMapType::const_iterator i = wordCountInfoForFile.begin();
          i != wordCountInfoForFile.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));
}
```

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





wordCountInfoForFile 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.

# Counting Words Across Files: C++11

```
#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;
}
```





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

## Counting Words Across Files: C++11

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

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);
   }
}
```

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



Slide 13

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.

## Counting Words Across Files: C++11

```
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 wordCountInfoForFile = f.get();    // move map returned by wordsInFile

    for (const auto& wordInfo : wordCountInfoForFile) {
      wordCounts[wordInfo.first] += wordInfo.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));
}
```





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.

wordCountInfoForFile can be eliminated by writing the subsequent for loop as follows:

```
for (const auto& wordinfo: f.get()) {
...                                              // as above
```

This is more efficient (the move into wordCountInfoForFile is eliminated), and it requires less source code. To be fair, however, the corresponding C++98 code would declare wordCountInfoForFile to be a reference, which I'd expect would yield object code just as efficient as the use of f.get() in the range-based for above. The code I currently show has the advantage that it facilitates discussion of how a copy can silently become a move, and it requires no knowledge of how binding a by-value function return value to a reference prolongs the lifetime of the returned object.

# 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;
}
```

# 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);
    }
}
```

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

# Comparison

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

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





      const WordCountMapType wordCountInfoForFile =
         wordsInFile(argv[argNum]);
      for ( WordCountMapType::const_iterator i =
               wordCountInfoForFile.begin();
            i != wordCountInfoForFile.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));
}
```

```cpp
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 wordCountInfoForFile =
         f.get();
      for (const auto& wordInfo : wordCountInfoForFile) {



         wordCounts[wordInfo.first] += wordInfo.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));
}
```

# 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/

## ">>" as Nested Template Closer

">>" now closes a nested template when possible:

```
std::vector<std::list<int>> vi1;      // fine in C++11, error in C++98
```

The C++98 "extra space" approach remains valid:

```
std::vector<std::list<int> > vi2;     // fine in C++11 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::list<std::array<int, n>>2 >> L1;   // error in C++98: 2 shifts;
                                        // error in C++11: 1st ">>"
                                        // closes both templates

std::list<std::array<int, (n>>2) >> L2; // fine in C++11,
                                        // error in C++98 (2 shifts)
```

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



Slide 19

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

## 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.





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

# 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
```

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



**Slide 21**

Rules governing auto are specified in 7.1.6.4 of C++11.

As noted in the treatment of std::initializer_lists, the only way that auto type deduction differs from template parameter type deduction is when deducing a type from a braced initializer list. auto deduces "{ x, y, z }" to be a std::initializer_list<T> (where T is the type of x, y, and z), but template parameter deduction fails. (It's a "non-deduced context.")

As noted in the discussion on rvalue references, the fact that auto uses the type deduction rules for 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&&
```

# 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]
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# auto for Type Declarations

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
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





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
```

# 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"
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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,
  obj.begin() and obj.end() or begin(obj) and end(obj) are valid.

Includes:

- All C++11 library containers.
- Arrays and valarrays.
- Initializer lists.
- Any UDT T where T.begin() and T.end() or begin(T) and end(T) yield suitable iterators.





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

"UDT" = "User Defined Type".

Per 6.5.4/1, if a type supports both member begin/end and non-member begin/end, ranges use the member versions. If a type has either begin or end as a member, no non-member will be searched for, so a pathological class offering, e.g., member begin but no member end will not be usable in a range-based for.

## 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; }
```





[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).

# Range-Based for Loops

Range form valid only for for-loops.

- Not do-loops, not while-loops.





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

# 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
```





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.

# 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.





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

## 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
```

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



**Slide 31**

Normally, logAndCall would employ perfect forwarding, but because neither rvalue references nor std::forward have yet been introduced, I'm using pass-by-value here for both func and param.

nullptr thus meshes with C++11's support for perfect forwarding, which is mentioned later in the course.

# Enhanced enums

Specification of underlying type now permitted:

```
enum Color: unsigned int { red, green, blue };
enum Weather: std::uint8_t { sunny, rainy, cloudy, foggy };
```

Values must fit in specified type:

```
enum Status: std::uint8_t { pending,
                            ready,
                            unknown = 9999 };     // error!
```

Type specification is optional:

```
enum Color { red, green, blue };                  // fine, same
                                                  // meaning as in
                                                  // C++98
```





The underlying type for an enum is always available via std::underlying_type<*enumtype*>::type. The underlying type for either of the Color definitions on this page, for example, is std::underlying_type<Color>::type.

std::underlying_type may be applied only to enum types.

# Scoped enums

"Strongly typed enums:"

- No implicit conversion to int.
  - No comparing scoped enum values with ints.
  - No comparing scoped enum objects of different types.
  - Explicit cast to int (or types convertible from int) okay.
- Values scoped to enum type.
- Underlying type defaults to int.

```
enum class Elevation: char { low, high };   // underlying type = char
enum class Voltage { low, high };           // underlying type = int
Elevation e = low;                          // error! no "low" in scope
Elevation e = Elevation::low;               // fine
int x = Voltage::high;                      // error!
if (e) ...                                  // error!
if (e == Voltage::high) ...                 // error!
```

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



**Slide 33**

enum struct may be used in place of enum class. There is no semantic difference.

"Normal" enums may use scope-qualified access, but enumerant names are still visible in the declaring scope:

```
enum Color { red, green, blue };            // "normal" enum
int x = Color::red;                         // fine, scope-qualified access
int y = red;                                // also fine (as in C++98)
```

# Forward-Declaring enums

enums of known size may be forward-declared:

```
enum Color;                                  // as in C++98,
                                             // error!: size unknown
enum Weather: std::uint8_t;                  // fine
enum class Elevation;                        // fine, underlying type
                                             // implicitly int
double atmosphericPressure(Elevation e);     // fine
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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/



**Slide 35**

From 3.9.1/5 in C++11: "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 C++11 2.14.3/2, "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."

# Unicode Support

There are corresponding string literals:

```
u"UTF-16 string literal"              // ⇒ char16_ts in UTF-16
U"UTF-32 string literal"              // ⇒ char32_ts in UTF-32/UCS-4
"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"               // ☠
```

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



Slide 36

A code point is a specific member of the Unicode character space. Not all Unicode characters correspond to a single code point. Per http://cppwhispers.blogspot.com/2012/11/unicode-and-your-application-1-of-n.html, "the standard defines code-point sequences that can result in a single character. For example, a code-point followed by an accent code-point will eventually result in an accented character."

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. UTF-8, UTF-16, and UCS-4/UTF-32 are defined by both ISO 10646 and 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."

u-qualified character literals may not yield UTF-16 surrogate pairs, but characters in u-qualified string literals may apparently be surrogate pairs. Per 2.14.5/9, "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
```

[The characters corresponding to the code points in the examples on the bottom of the page are present in the comments, but, because they don't display property on all machines (presumably due to variations in the fonts installed), I've superimposed an image showing the same characters on top of the comments. To see if the characters display properly on your machine, move or delete the image.]

# 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>
```





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

# 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++11 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>.





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++11.

P.J. Plauger, who proposed codecvt_utf8_utf16 for C++11, explains the two seemingly redundant UTF-16 ⇄ UTF-8 conversion instantiations: "The etymologies of the two are different. There should be no behavioral difference."

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

# 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++11:

- 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.





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., std::ios_base.

Among the esoteric details are that the existence of a protected destructor in template std::codecvt implies that none of its instantiations – i.e., none of the standard facets -- work with std::wbuffer_convert and std::wstring_convert. Instead, it's expected that types derived from std::codecvt (e.g., from a standard facet) will be used. Standard library types satisfying this expectation are std::codecvt_utf8, std::codecvt_utf16, and std::codecvt_utf8_utf16.

More information regarding use of standard facets with std::wbuffer_convert and std::wstring_convert is in the comp.std.c++ thread at http://tinyurl.com/ykup5qe.

# 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 (w/o a skull & crossbones))"
```





"R" must be upper case and must come after "u8", "u", "U", etc. It can't be placed in front of those specifiers.

## 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)"
```





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).

## 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
```





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

# 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
```





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

# 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 };
```





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}) {}
```

Per Daniel Krügler's email of 12/21/11, this was unintentional, and this "feature" may be removed in a later bug-fix version of the standard.

## 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
```





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

# 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.




The technical definition of an aggregate is slightly more flexible than what's above. From 8.5.1/1: "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 8.5.4/4, elements in an initialization list are evaluated left to right.

## 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
```





"UDT" = "User Defined Type".

C++11 does not support C99's designated initializers:

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

# 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
```





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

## Uniform Initialization Syntax

Brace-initialized variables may use "=":

```
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 };
```





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

# Uniform Initialization Syntax

Other uses of brace initialization can't:

```
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!
```





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

# 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 "=".**





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

# 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.





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

# Brace Initialization and Implicit Narrowing

Sole exception: implicit narrowing.

- C++98 allows it via brace initialization, C++11 doesn't:

```
struct Point { int x, y; };
Point p1 = { 1, 2.5 };                          // fine in C++98:
                                                // implicit double ⇒ int
                                                // conversion;
                                                // error in C++11
Point p2 = { 1, static_cast<int>(2.5) };        // fine in both C++98
                                                // and C++11
```





The initializations of p1 and p2 require the "=" in C++98, because it's grammatically required. In C++11, the "=" is optional, but the narrowing conversion in p1's initialization would not be legal.

Narrowing conversions are defined in 8.5.4/7. 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
```

# 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.
```





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

# 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.





"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.

# 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





Per 8.5.4/6, the lifetime of the array is the same as that of the initializer_list object.

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 4.5-4.7, MSVC10, and MSVC11 November 2012 CTP, sizeof(std::initializer_list<T>) is 8. Per Stephan T. Lavavej's post at http://tinyurl.com/3gcsww8, VC10 lacks language support for std::initializer_list, and it was an error that <initializer_list> shipped with VC10. That header was removed from VC11 due to the lack of such support, but it was added back in for VC11's November 2012 CTP.

## 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.
```




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.

# 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.





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

# 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
```





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

# 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
```

Applies only to **brace-delimited arguments**:

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





The relevant parts of C++11 regarding initializer lists and overload resolution (the topic of the next few slides) are 13.3.1.7, 8.5.4/2-3, and 14.8.2.1/1.

# Initializer Lists and Overload Resolution

Not of only theoretical interest:

```
template <class T,                                    // from the C++11
          class Allocator = allocator<T> >            // standard
class vector {
public:
  ...
  vector(size_type n, const T& value, const Allocator& = Allocator());
  vector(initializer_list<T>, const Allocator& = Allocator());
  ...
};


std::vector<int> v1(10, 5);       // v1.size() == 10, all values == 5
std::vector<int> v2{10, 5};       // v2.size() == 2, values == {10, 5}
```

Choose carefully between {} and () when initializing objects!





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

## Braced Initializers and auto

auto deduces std::initializer_list for braced initializers:

```
auto i = { 2, 4, 6, 8 };          // i is std::initializer_list<int>
```

In general, templates deduce no type for braced initializers:

```
template<typename T>
void f(T param) { ... }

f({ 2, 4, 6, 8});                 // error! no type deduced for
                                  // "{ 2, 4, 6, 8 }"
```

- Only way that auto type deduction ≢ template type deduction.





Per 14.8.2.1/1, template parameter type deduction does not apply to braced initializers (it's a "non-deduced context"), unless the template parameter is itself a std::initializer_list. Hence

```
template<typename T>
void f1(T param);

f1({2, 4, 6, 8});                 // error! can't deduce T
```

but

```
template<typename T>
void f1(std::initializer_list<T> param);

f1({2, 4, 6, 8});                 // fine, T deduced as int
```

# Braced Initializers and auto

Especially for single-element braced initializers, this can confuse:

```
auto i1 = 10;                    // i1 is int
auto i2(10);                     // i2 is int
auto i3 {10};                    // i3 is std::initializer_list<int>
```

Particularly when such variables interact with overload resolution:

```
std::vector<int> v1(i1);         // v1.size() == 10, values == 0
std::vector<int> v2(i2);         // v1.size() == 10, values == 0
std::vector<int> v3(i3);         // v1.size() == 1, value == 10
```

Use care when initializing auto variables with braced initializers!





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

# 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 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
```





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

# 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);
};

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





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

# 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).
- Choose carefully between {} and () when initializing objects.





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

## 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());
```





The generated *MagicType* above is not technically accurate, because closure types are neither default-constructible nor assignable, but these details aren'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.

## 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.





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 5.1.2/2, closures are rvalues (prvalues, to be precise).

## Variable References in Lambdas

Closures may outlive their creating function:

```
std::function<bool(int)> returnClosure(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 = returnClosure(10);                     // f is essentially a
                                                // copy of λ's closure
```

In this call,

```
if (f(22)) ...                                  // invoke the closure
```

what are the values of a, b, c?

- returnClosure no longer active!





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

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

In the line with the comment "invoke the closure," we're really invoking the copy of the closure that's stored inside the std::function object.

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

# Variable References in Lambdas

This version has no such problem:

```
int a;                                      // now at global or
                                            // namespace scope
std::function<bool(int)> returnClosure()
{
  static int b, c;                          // now static
  ...
  return  [](int x)                         // now compiles
          { return a*x*x + b*x + c == 0; };
}
auto f = returnClosure();                   // as before
...
if (f(22)) ...                              // as before
```

a, b, c outlive returnClosure's invocation.





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

# Variable References in Lambdas

Rules for variables lambdas may refer to:

- **Non-static locals referenceable only if "captured."**

```
std::function<bool(int)> returnClosure(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
```

- **Variables of static storage duration always referenceable.**

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

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



Slide 71

Objects/variables of *static storage duration* are those defined at global, namespace, or file scope or declared static inside structs/classes. Not only is there no need to capture such objects, it's also not permitted.

## 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; });
}
```

Essentially 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/



Slide 72

Neither C++11 nor C++14 specify the access level of the data members in the closure type, but they are inaccessible, because they are “unnamed.” Hence my use of private as their access specifier in the code above (and elsewhere where I show closure types).

In C++11, 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.

As shown later, C++14 essentially supports move capture.

# 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; });
}
```

Essentially 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
```





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

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Capturing Local Variables

Different (non-static) 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; });
}
```

Essentially 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/

# 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.





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

# 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; });
```

Essentially 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/
© 2013 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(),                    // error! attempt
                         [minVal](int i) { return i > minVal; }    // to capture
    );                                                              // "this->minVal"
  …
}

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





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.

## 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, copies
      );                                                         // "this" into closure
   …
}

void Widget::doSomething() {
   auto it = std::find_if(li.cbegin(), li.cend(),
                          [&](int i) { return i > minVal; }      // also fine, holds
      );                                                         // ref to "this" in
   …                                                             // closure
}
```

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





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

# 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));
                });
```





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

# 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
};
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.




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

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

# 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.





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.

## 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/



**Slide 83**

Every lambda expression yields a unique closure type. VC10 names these types anonymous-namespace::<lambda0>, anonymous-namespace::<lambda1>, etc. VC11 names them class <lambda_db589cda92d21474f828ce3af1960352>, class <lambda_7e63fc6a92f6cf750cac78656e89c248>, etc. gcc 4.5-4.7 uses 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)); };
```

Returning an object like factorial from a function would be a bug, because the std::function object would contain a reference to a local object (factorial) that had been destroyed.

## 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
```





VC10, despite support for trailing return type syntax in general, does not compile the second declaration of multipleOf5 on this page. VC11 accepts it.

## 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 in C++11

std::function<bool(long)> makeFunc();         // fine

template<typename Func>
Func makeFunc();                              // fine, but generates
                                              // multiple functions,
                                              // and callers must
                                              // specify Func
```

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



Slide 85

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."

# Storing Closures

- Not allowed for class data members:

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

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





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

## 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.





The mechanics of the assignment to f are interesting, because, per 5.1.2/19, closures themselves are not assignable. Conceptually (and, modulo optimizations, in reality), std::function objects hold pointers to copies of the callable objects they hold, so in the assignment to f, a temporary std::function object is created holding a copy of the closure on the right hand side of the assignment. (The copy of the closure is created by move-construction.) Because operator= for std::function with an lvalue argument is defined in terms of swap, f then swaps its pointer with that of the temporary.

The fact that std::function::swap is noexcept is the basis for my belief that std::function objects are expected to contain pointers to the callable objects they hold.

# Stored Closures and Dangling References

In member functions, even "[=]" capture can lead to dangles.

- Class data members aren't copied, this is:

```
class Widget {
public:
  ...
  std::function<int (int)> numberCruncher()
  { return [=](int x) { return x * myVal; }; }     // copies "this", not myVal!
private:
  int myVal;
};
std::unique_ptr<Widget> pw(new Widget);  // create Widget
...
auto f = pw->numberCruncher();           // save closure
...
pw = nullptr;                            // delete Widget
...
auto value = f(22);                      // undefined! f refers to
                                         // deleted data
```





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

# Stored Closures and Dangling References

Copy-capturing local copies of data members avoids this problem:

```
class Widget {
public:
  std::function<int (int)> numberCruncher()
  {
    auto localMyVal(myVal);                             // copy myVal to local
    return [=](int x) { return x * localMyVal; };   // copy local into closure
  }
  ...
};
std::unique_ptr<Widget> pw(new Widget);      // as before
...
auto f = pw->numberCruncher();                  // as before
...
pw = nullptr;                                           // as before
...
auto value = f(22);                                   // now well-defined
```





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

# Lambdas in C++14

"Polymorphic" parameters permitted:

```
std::vector<std::shared_ptr<Widget>> vspw;
…
std::sort(vspw.begin(), vspw.end(),
          []( const std::shared_ptr<Widget>& p1,          // C++11
              const std::shared_ptr<Widget>& p2)
          { return *p1 < *p2; });

std::sort(vspw.begin(), vspw.end(),
          []( const auto& p1, const auto& p2)             // C++14
          { return *p1 < *p2; });
```

Generated classes have templatized operator()s:

```
class MagicType {
public:
  template<typename T1, typename T2>
  bool operator()(const T1& p1, const T2& p2) const
  { return *p1 < *p2; }
};
```





This example demonstrates that auto parameters can be useful even if only a single type will be passed. In such cases, it's the brevity of auto that's attractive, not its support for varying parameter types. (An example where the polymorphism is helpful is shown near the end of the course, where a variadic and polymorphic lambda performs perfect forwarding.)

# Lambdas in C++14

Return type may be deduced, even with multiple statements:

```
std::vector<double> v;
…
std::transform( v.begin(), v.end(), v.begin(),          // C++11
                [](double d)->double                    // (as before)
                {
                  makeLogEntry("std::transform", d);
                  return std::sqrt(std::abs(d));
                });

std::transform( v.begin(), v.end(), v.begin(),          // C++14
                [](double d)                            // (note no
                {                                       // return type)
                  makeLogEntry("std::transform", d);
                  return std::sqrt(std::abs(d));
                });
```





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

# Deduced Return Types in C++14

Return type deduction valid for *all* functions.

- Leading auto still required for non-lambdas.

```
auto f(int x);                  // C++14; return type to be deduced

class Widget {
public:
  auto mf() const;              // C++14; return type to be deduced
  …
private:
  int x, y;
};
```

Deduction based on function *definition*:

```
auto f(int x) { counter += x; }      // no return stmt ⇒
                                     // return type is void

auto Widget::mf() const
{ return x + y; }                    // return type is int
```





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

# Deduced Return Types in C++14

Multiple returns permitted if same type deduced for each:

```
auto magicValue1(int seed)           // fine; both returns are int
{
  if (seed >= 0) return seed;
  else return 42;
}

auto magicValue2(int seed)           // error! different return types
{
  if (seed >= 0) return seed;
  else return 4.2;
}

std::deque<long> d;
…
auto it = std::partition( d.begin(), d.end(),                // fine
                          [] (auto val)
                          { if (val % 10 == 0) return true;
                            if (val % 5 == 0) return true;
                            return false; });
```





Per C++14 7.1.6.4/7-9, the rules for deducing the type of a function with multiple returns are the same as the rules for deducing the type of multiple variables declared with a single auto declaration. The sole exception is that auto will deduce a std::initializer_list type for variables with braced initializers, while using braced initializers in return statements for functions with deduced return types is an error.

# Move Capture in C++14

No way to move objects into closures in C++11:

```
class Widget {
public:
  double operator()();
};

std::function<double()> getCalcFcn()
{
  std::unique_ptr<Widget> pw =              // std::make_unique is
    std::make_unique<Widget>();             // C++14 only

  return [pw] { return (*pw)(); };          // error! pw not copyable
}

std::function<double()> getCalcFcn()
{
  std::unique_ptr<Widget> pw = …;           // as above

  return [&pw] { return (*pw)(); };         // compiles, but
}                                           // getCalcFcn returns
                                            // dangling reference
```





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

## Move Capture in C++14

No problem in C++14:

```
std::function<double()> getCalcFcn()
{
  std::unique_ptr<Widget> pw =              // as before
    std::make_unique<Widget>();

  return [pw = std::move(pw)]               // move pw into closure;
         { return (*pw)(); };               // C++14 only
}
```

Generated class:

```
class MagicType {
public:
  MagicType(std::unique_ptr<Widget>&& ptr)
  : pw(std::move(ptr)) {}

  double operator()() const { return (*pw)(); }

private:
  std::unique_ptr<Widget> pw;
};
```





[std::move has been mentioned earlier in the course, but it has not yet been explained.]

## Generalized Captures in C++14

Feature actually much more general:

- Arbitrary declaration/initialization of closure data members.

```
int computeMinVal(int seed);
int computeMaxVal(int seed);

std::function<bool(int)>
compFunc(int minSeed, int maxSeed)
{
  return[minVal = computeMinVal(minSeed),
         maxVal = computeMaxVal(maxSeed)] (int x)
        { return minVal <= x && x <= maxVal; };
}
```

- "Move capture" doesn't really exist.
  - ➡ Above, neither minVal nor maxVal are local variables.

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



Slide 96

The feature actually permits the arbitrary declaration and initialization of *nonstatic* closure data members.

## Generalized Captures in C++14

For

```
return [ minVal = computeMinVal(minSeed),                  // from
         maxVal = computeMaxVal(maxSeed)] (int x)           // previous
         { return minVal <= x && x <= maxVal; };            // slide
```

generated code is:

```
class MagicType {
public:
  MagicType(int min, int max)
  : minVal(min), maxVal(max) {}

  bool operator()(int x) const { return minVal <= x && x <= maxVal; }

private:
  int minVal, maxVal;
};

…

return MagicType(computeMinVal(minSeed),
                 computeMaxVal(maxSeed));
```





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

## Closures as Function Pointers

Capture-less closures implicitly convert to function pointers:

```
int (*fp)(int) = [](int x) { return x * x; };
```

Such closures can be treated like functions.

- No need for std::function to refer to them.
- No captures ⇒ no stored pointers or references ⇒ no dangling.
- Often useful for callbacks with C-like APIs:

```
int atexit(void (*f)()) noexcept;          // from <cstdlib>
std::atexit([]{ logMsg("Shutting down..."); });
```

  - ➡ In C++11, function pointer linkage not specified ⇒ possible linkage problems.
    - ◆ C++14 specifies C++ linkage.
  - ➡ noexcept akin to throw(), but enables more optimizations.
    - ◆ Violated noexcept ⇒ terminate.
    - ◆ Exception specifications now deprecated.

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



Slide 98

The implicit conversion from closure to function pointer is specified in 5.1.2/6.

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). "

<cstdlib> actually declares two overloads for atexit, one each for C and C++ linkage. From 18.5/4:

```
extern "C" int atexit(void (*f)(void)) noexcept;
extern "C++" int atexit(void (*f)(void)) noexcept;
```

# Lambdas as const Initializers

Excellent as simple multi-statement initializers for consts:

```
const auto sortedInts = []()->std::vector<int> {       // init const vector
  std::vector<int> v(NUM_VALUES);                       // w/NUM_VALUES
  std::iota(v.begin(), v.end(), 0-NUM_VALUES/2); // sequential ints
  return v;                                              // centered at 0
}();

const auto priority = [=]()->Priority {
  makeLogEntry("Initializing priority");
  auto cInfo = getCustomerInfo(customerID);
  Priority p = (cInfo.salesInLast12Months() < bonusThreshhold)
               ? normalPriority
               :  highPriority;
  return p;
}();
```





The first lambda has no capture, because I'm assuming that NUM_VALUES has static storage duration (e.g., is a global). The second lambda uses a default capture, which I'm assuming is correct for reference to customerID and bonusThreshhold.

In C++14, both return type specifications on this page could be omitted.

# 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
```

A factory function can simplify user code:

```
template<typename T, typename CF>                 // factory for set
std::set<T, CF> make_set(CF cmpFunc)              // w/custom
{                                                 // comparison
  return std::set<T, CF>(cmpFunc);                // function
}

auto s = make_set<int*>([](int *pa, int *pb) { return *pa < *pb; });
```

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



Slide 100

[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
```

# 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.
- C++14 adds support for auto parameters, generalized captures, and less restrictive return type deduction.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Alias Templates

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>>
```





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

# Alias Templates

Alias templates 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>;
```





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

# 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.
```

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





Modifying the declarations for CallBackPtr to omit the explicit pointer symbol as follows,

```
typedef void CallBackPtr(int);        // note lack of "*"
using CallBackPtr = void (int);       // ditto
```

i.e., to simply specify the function types, compiles, but the resulting types seem to be function types, not function pointer types, and as such, variables of those types can't be initialized or assigned. For example, given

```
void mycallback(int);

using CallBackPtr = void (int);

CallBackPtr p = mycallback;
```

gcc 4.7 complains:

```
error: function 'void p(int)' is initialized like a variable
```

Behavior is the same with the conventional typedef syntax.

# 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.

Primary headers:

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





This course is about C++11, 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."

# 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.
```





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 this can be approximated by e.g., having the thread function (typically a closure) start by waiting on a std::future<void> corresponding to a std::promise set by the thread spawning the "suspended" thread.

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

# Data Lifetime Considerations

Functions called in ST systems 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);
```





"ST" = "Single-Threaded".

# 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.





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

# 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.





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

## 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/



Slide 110

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.

# Avoiding Lifetime Problems

Two basic strategies:

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





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

# 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.





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.

# 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.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





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

# Copying Arguments

Summary:

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

My preference: lambdas.

- Most natural syntax.
- Copying is explicit.





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

# 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
```




Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





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

# Mixing By-Value and By-Reference Arguments

- **Variadic thread constructor or bind**: Use C++11'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.





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

## 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.





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

## 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();
```




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.

# 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.





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.

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. Once the runtime has indicated that it will run the function deferred (e.g., via a return value of future::status::deferred from wait_for), the decision is irrevocable, because user code might takes actions based on that information.

Motivation for async calls using std::launch::deferred executing only when get/wait is called is in N2973 under "Eager and Lazy Evaluation."

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.

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

# 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.





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.

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)."

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

# 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).





Invoking get more than once on a std::future yields undefined behavior. Per 30.6.6/3, implementations are encouraged (but not required) to throw a std::future_error exception in this case.

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.

# 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
```





For unshared futures, wait_for is useful only 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 simply return std::future_status::deferred.

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++11's concurrency support.

## 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
```





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.

# Mutexes

C++11 has 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.





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."

# Mutexes

C++14 adds:

- std::shared_mutex:
  - Non-recursive reader/writer lock w/timeout support.
  - Adds lock_shared/try_lock_shared to std::timed_mutex API.

Based on Boost's shared_mutex, but with fewer capabilities:

- Can't upgrade read lock to exclusive lock.





Like C++11's mutex types, std::shared_mutex is neither copyable nor movable.

## 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.
  - ➡ Locks std::shared_mutexes in exclusive (write) mode.




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, though the term “guard” is also used.

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.”

# 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 unshared timed mutexes.
    - Locks std::shared_mutexes in exclusive (write) mode.

```
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
```

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



Slide 129

The name unique_lock is by analogy to unique_ptr. Originally, a "shared_lock" type was proposed for C++11 (to be a reader/writer lock), but it was not adopted until C++14.

## 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
```

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





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

# std::shared_lock (C++14)

C++14 adds:

- **std::shared_lock**: like std::unique_lock, but locks std::shared_mutexes for shared (read) access.

```
using SM = std::shared_mutex;            // typedef
SM m;                                    // mutex object
{
  std::shared_lock<SM> L(m);             // lock m in read mode
  …                                      // critical section
                                         // for reading
  L.unlock();                            // unlock m
  …
}                                        // nothing happens
```





The std::shared_lock API appears to be identical to the API for std::unique_lock, except that std::shared_lock works only with std::shared_mutex objects, and when it locks those objects, it does it in shared mode instead of in exclusive mode.

[The example on this page is essentially the same as the one used to introduce std::unique_lock. That's to emphasize the close relationship between std::unique_lock and std::shared_lock.]

# std::shared_lock (C++14)

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

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



**Slide 132**

[The example on this page is essentially the same as the one used to demonstrate std::unique_lock's capabilities. That's to emphasize the close relationship between std::unique_lock and std::shared_lock.]

# 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

}
```

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



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

# 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

}
```





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.)

# Multiple Mutex Acquisition

Works with std::shared_lock as well as std::unique_lock.

- E.g., for write access to assignment target, read access to source:

```
class Widget {
public:
  Widget& operator=(const Widget& rhs)
  {
    if (this != &rhs) {
      std::unique_lock<std::shared_mutex> dest(m, std::defer_lock);
      std::shared_lock<std::shared_mutex> src(rhs.m, std::defer_lock);

      std::lock(dest, src);    // lock dest in write mode, src in read mode

      ...                      // assign data
    }                          // unlock mutexes

    return *this;
  }

private:
  mutable std::shared_mutex m;
  ...
};
```

```
Widget w1, w2;
…
// Thread 1        // Thread 2
w1 = w2;           w2 = w1;
```

No deadlock!





[Slide is animated.]

Example is taken from *Shared locking in C++, Revision 1*, Howard Hinnant, C++ Standardization Committee Document N3568, 11 March 2013.

## 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 lock type.
  - Possibly less efficient, more flexible.
    - E.g., works with std::shared_lock<std::shared_mutex>>.





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 inherently tied to mutexes suggests that shared data is always involved. Pure notification of an event that occurs at most once can be achieved via use of a std::future<void>. Unlike condition variables, where if no thread is waiting, notification has no effect, any thread waiting on a std::future<void> will "see" that the future has been set, even if it was set before the wait occurred.

There are no examples of condition_variable_any in this course.

When a thread waiting on a std::condition_variable_any<std::shared_lock<std::shared_mutex>> is notify-ed, the mutex is locked in whatever mode (shared or exclusive) it had when the wait was entered.

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

# 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.




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.

## 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
}
```





[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).

Nicolai Josuttis remarks that "there is an important difference between waiting for a duration (such as seconds(1)) and waiting for a timepoint (now()+seconds(1)). The latter is not guaranteed to wait one second if time adjustments occur (unless the steady_clock is used). It might wait shorter or longer." This is true, but calling wait_for(*lock*, *duration*) is defined to be the same as wait_until(*lock*, chrono::steady_clock::now() + *duration*), i.e., use of steady_clock is the default.

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.

# 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.





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

# 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;
  …
};
```

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



**Slide 140**

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.

Thread-locals may be dynamically initialized and may be declared extern.

# Other Concurrency Features

- Thread-safe initialization of objects of static storage duration.
- Library thread safety guarantees (e.g., for std::cin/std::cout, STL containers, std::shared_ptr, etc.)
- 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.
- Operations on current thread, e.g., yield and sleep.
- Atomic types (e.g., std::atomic<int>) with memory ordering options.
- Query number of hardware-supported threads.
- Many other features for threads, locks, condition variables, etc.,
  - ➡ This was an *overview*.

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

**Slide 141**

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 specifc 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++11's library thread safety guarantees is to search chapters 17ff in the standard for "data race". Relevant sections 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 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.

# 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.





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

## 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.
- Enhanced enums increase type-safety, reduce namespace pollution, and permit control over the underlying type.
- Unicode string encodings support UTF-8, UTF-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.
- Alias templates allow “template typedefs” to be created.
- Concurrency support includes mutexes, locks, condition variables, thread-local data, asynchronous calls, and more.





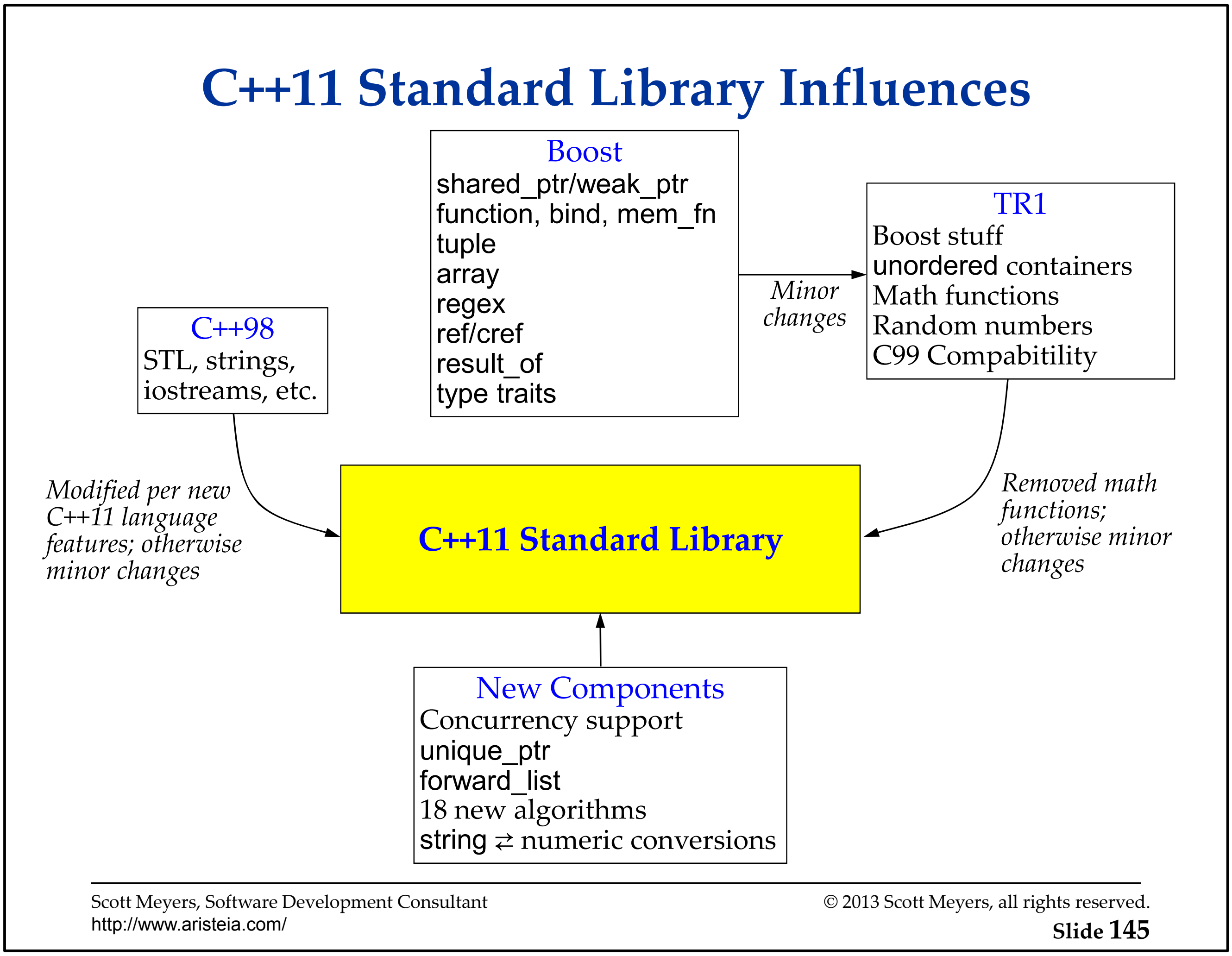
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

## Overview

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




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.




Although the C++98 box is smallest, it had the strongest influence on the C++11 standard library.

# 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/



Slide 146

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

There is no emplace version of insert taking a repeat count, because there would be no way for the function to distinguish a repeat count from an argument to be perfect-forwarded to the constructor of the to-be-created object.

# 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.





Regarding vector::shrink_to_fit, C++11 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 .

# TR1

- Standard C++ Committee Library "Technical Report 1."
- Basis for most new library functionality in C++11.
- Largely derived from Boost libraries.
- TR1 functionality in namespace std::tr1.
- C++11 TR1-derived functionality in std.
  - Not identical to that in TR1.
    - Uses new C++11 features.
    - Tweaks some APIs based on experience.
  - APIs mostly backwards-compatible





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

# From TR1 to C++11

Common C++11 enhancements:

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





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

# 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/



Slide 150

Libraries in blue are also in C++11. 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++11 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++11 specifies default versions for the engine and distributions, but it also allows for customized versions of both.

# From TR1 to C++11

| TR1 Functionality | C++11 Functionality Changes |
|---|---|
| **Reference Wrapper** | Minor enhancements. |
| **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++11. (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>). |


Slide 151

Of the 23 proposed mathematical special functions in TR1, 21 are preserved in the separate standard, “Extensions to the C++ Library to Support Mathematical Special Functions” (ISO/IEC 29124:2010). The FDIS is N3060 and is freely downloadable. 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++11 specification. From Stephan T. Lavavej: “In C++11, 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.

The “minor enhancements” to reference wrappers are, per Daniel Krügler’s 12/20/11 email, that functions may be bound and that const temporaries may not be. (TR1 already prohibited binding non-const temporaries.)

## 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.
  - Strange copy semantics.
    - No containers of auto_ptr in C++98/03.
    - auto_ptrs in C++11 containers OK, but copying prohibited.
- 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.





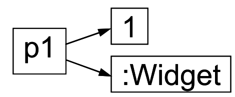
The "From TR1" in the title indicates that this is a C++11 feature based on TR1 functionality.

Although copying auto_ptrs in C++11 containers is prohibited, moving them is permitted.

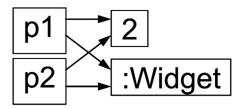
## shared_ptr

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

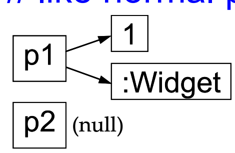
```
{
  std::shared_ptr<Widget> p1(new Widget);
```




```
  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
```

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



**Slide 153**

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

"RC" = "Reference Count".

## 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);
```

  - 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
```





[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.

# 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/



Slide 155

"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.

# 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
```





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

# 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;
```





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++11) 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++11 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.

# 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.





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 C++11 is 17.6.4.8/2.

# 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.





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

# 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.





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

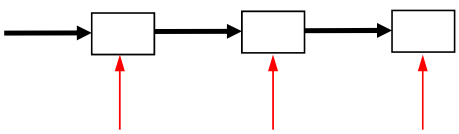
# weak_ptr

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

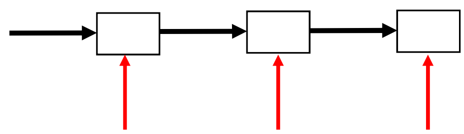
- When a resource's RC → 0, its weak_ptrs *expire.*
  - ➡ The shared_ptr releasing a resource expires all weak_ptrs:

```
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



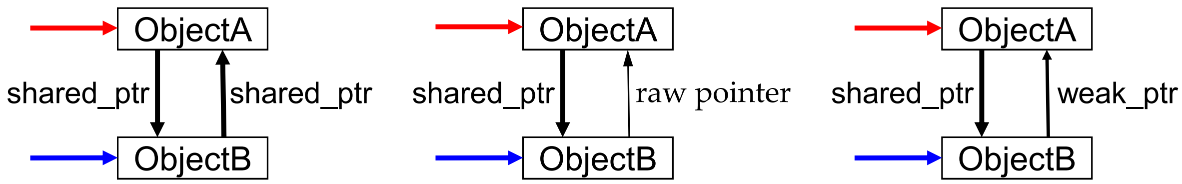


Calling expired may be faster than calling use_count, because use_count may not be constant-time. Calling unique is not an alternative, because unique does not exist for weak_ptrs.

"RC" = "Reference Count".

# 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





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".

## 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
```



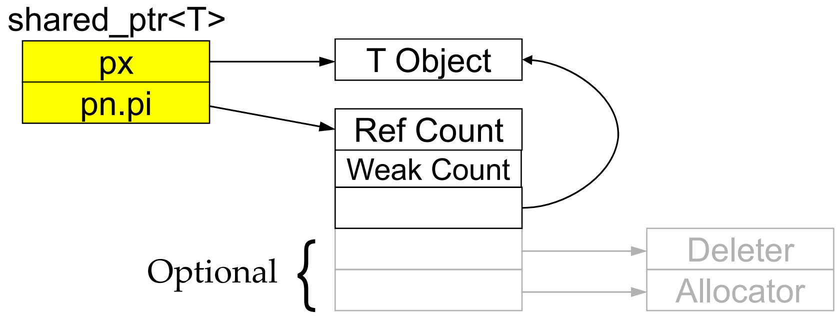


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

# Cost of shared_ptrs

Sample implementation (Boost 1.52):

- 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.





“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.

As of Boost 1.52, it appears that Boost does not implement the "We Know Where You Live" optimization.

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.

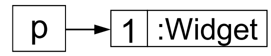
Memory allocation for the RC is avoided if std::make_shared (discussed on next page) is used.

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.”

# From TR1 to C++11

- 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);
```

  - Both also enable "We Know Where You Live" optimization.
  - But no way to specify custom deleters.
- 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.




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".

## 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.





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

# unique_ptr

Successor to auto_ptr (which C++11 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.
  - VC10-11 and gcc 4.7: both typically hold 1.





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 VC10-11 and gcc 4.7 "typically" holding only 1 pointer is based on the assumption that "typical" use involves no custom deleter.

## 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/





[This slide mentions lvalues for the first time.]

## 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/



Slide 169

[This slide mentions rvalues for the first time.]

## 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())
```





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

# 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.





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

# 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
```





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

# 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.
  - ➡ One is expected to be added.





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.

## 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.



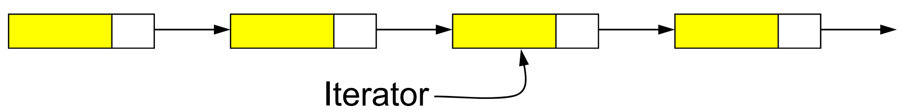


unique_ptr and shared_ptr do different things, so their APIs can't be the same, but in some cases they are different for no apparent reason.

# 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.

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



**Slide 175**

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.

# 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
```

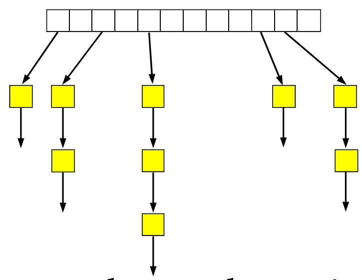




Scott Meyers, Software Development Consultant
http://www.aristeia.com/

# 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++11 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.

# 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 offers 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.





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.

# 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 { … };
```





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

## 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





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 19.5.2.1/1, C++11 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).

## Hashing Functions

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

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

std::unordered_set<Widget> sw;


struct IntHasher {
  std::size_t operator()(int i) const { … };
};

std::unordered_map<int, std::string, IntHasher> mis;
```





"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.

## 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 s1 to use new B
float z_max = s1.max_load_factor();  // get current zmax (defaults to 1)
```

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





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.

## 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.





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.

# 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';
}
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





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

# From TR1: Tuples

Motivation:

- pair should be generalized.
- Tuple utility demonstrated by other languages.





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

# 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/



Slide 187

"Containers" is in quotes, because tuple doesn't, in general, adhere to the container interface.

# 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.





TMP can be used to generate code to iterate over the contents of a tuple.

# get

Using named indices makes for more readable code:

```
enum EmpInfo { Name, Addr, HireDate };

Name empName(std::get<Name>(info));
Address empAddr(std::get<Addr>(info));
Date empHDate(std::get<HireDate>(info));
```




Using an enum class here is less convenient, because each enumerant would have to be cast to its underlying integral type before passing it to get. A discussion of this issue is at http://stackoverflow.com/questions/10742275/stdget-using-enum-class-as-template-argument.

# 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.)
```

Function returns tuple + caller uses tie ⇒ function has >1 return values.

- C++11 offers functions with multiple return values!





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);
```

Because std::tie can be used with std::tuple-like entities like std::pair and std::array, I don't use code font for "tuple" in the statement at the bottom of the slide.

# 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());
}
```





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).

# 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
  …
}
```





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

# 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.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





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

# 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.





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

# 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
```





For std::array objects with a size of 0, results of invoking data are “unspecified.”

# Fixed-Size Arrays

Because arrays are fixed-size,

- No insert, push_back, erase, clear, etc.
- No dynamic memory allocation.
  - Hence no allocator.

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





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

# 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
```





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

# 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/



Slide 199

Value initialization is defined in 8.5/7.

"UDT" = "User Defined Type".

# 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:
  Widget(int = -1);
  ...
};
std::array<Widget, 5> arr4;           // default-construct all Widgets
                                      // (i.e., using -1 as ctor param)
std::array<Widget, 5> arr5 {0, 1};    // same as "{0, 1, -1, -1, -1}"
```

All behavior above is same as for built-in arrays.




The aggregate initialization rules for std::arrays are the same as for built-in arrays.

The initialization for arr5 would not be valid if the Widget constructor were explicit, because when an aggregate is initialized with a braced initializer list, the values are used to copy initialize the members of the aggregate (per 8.5.1/2) , and explicit constructors can be used only for direct initialization (per 12.3.1/2).

# arrays are Aggregates

Because array is an aggregate:

- All (non-static) data 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
```





Aggregates may have non-public data members that are static, but I find it hard to imagine why std::array would have any. Daniel Krügler, however, remarks (per his 9/6/10 email): "I could think of at least one very reasonable example: A zero-sized array could return a non-null-pointer to some static storage for member function data."

# 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';
```





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

# 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.





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

# 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, little reason to use C-style arrays.**





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

# 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.





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

## 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.





Conceptually, C++11 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.

## 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
```





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."

# 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

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



Slide 208

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).

## 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);
}
```





Information on matches found can be retrieved through an optional match_results parameter. The next slide gives an example.

# 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*.





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;
```

# 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.





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

# 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
}
```





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.

## 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;
}
```





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*" or "\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.

# 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;
}
```





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

## 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.





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.

# From TR1: Generalized Functors

Motivation:

- Function pointers and member function pointers are rigid:
  - Exact parameter/return types and ex. specs. must be specified.
  - Can point to only one of static and 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.





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.)

## 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
```





The term "callable entity" is mine and slightly more restricted than the C++11 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.)

# 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())()
```





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

# 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
```

# std::function Basics

- Declared in <functional>.
- std::functions are type-safe wrappers for callable entities:

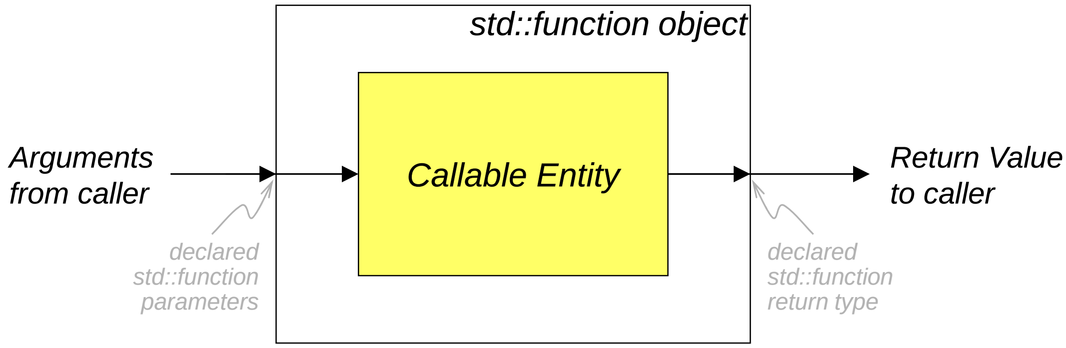
```
std::function<int(std::string&)> f;      // f refers to callable entity
                                         // compatible with given sig.

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
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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
```



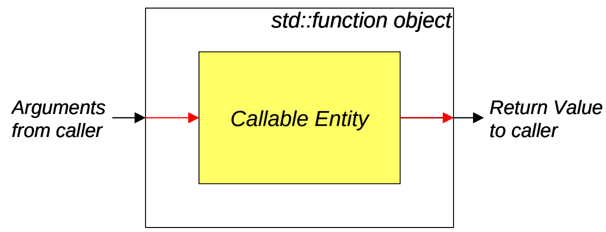


The outer box represents the std::function object, the inner box the callable entity it wraps (i.e., forwards calls to).

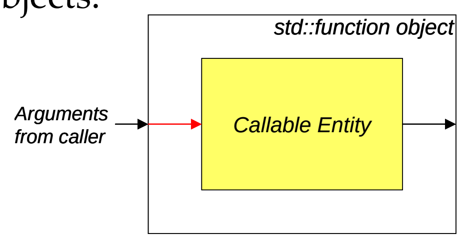
# Compatible Signatures

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 to the function object's.




➡ All entity return types are compatible with void-returning function objects.








Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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;
};
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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





For static member functions, the use of "&" before the name is optional when taking their address (i.e., same as non-member functions).

# 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
```





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

# 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
```




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.

# 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!
```





Equality testing is disallowed via the use of an explicit conversion to bool (i.e., std::function contains the member function, “explicit operator bool() const”). As explained later in this course, explicit conversion functions may not be called in a test such as “if (f1 == f2)”.

# 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.





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

# 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++11.





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

# 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.

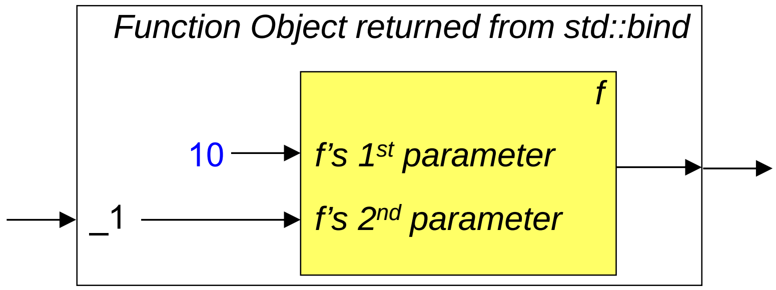




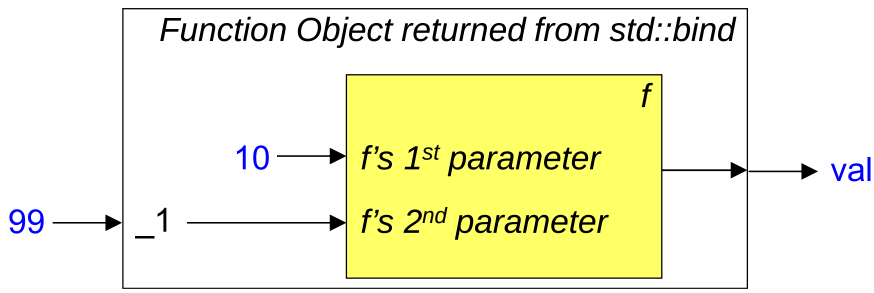
The information on this page is likely to make sense only after examples have been presented.

# 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
```








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.

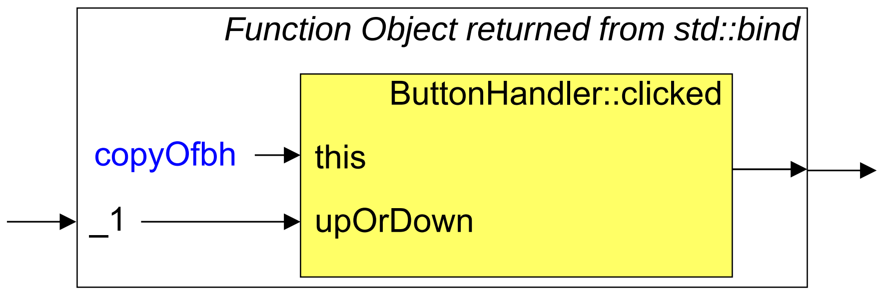
bf is not the same object as the one returned from bind, but in all likelihood, it's been move-constructed from the rvalue returned by bind.

# 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));
```








_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.

# 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));
```





std::unique_ptr lvalues must be wrapped by std::ref when bound, because std::unique_ptr isn't copyable. (It's only movable.) If the lvalue's value need not be preserved, std::move may be used instead of std::ref.

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.)

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

## 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.



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.

# 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));
```





_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.

# 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/

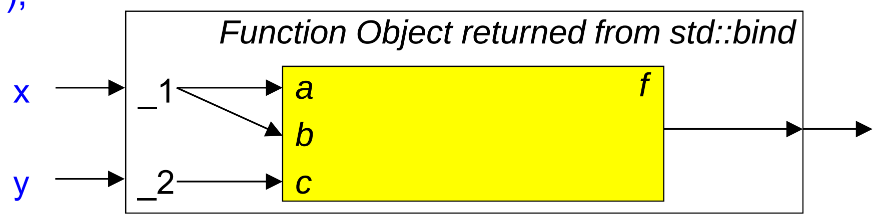
**Slide 236**

_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.

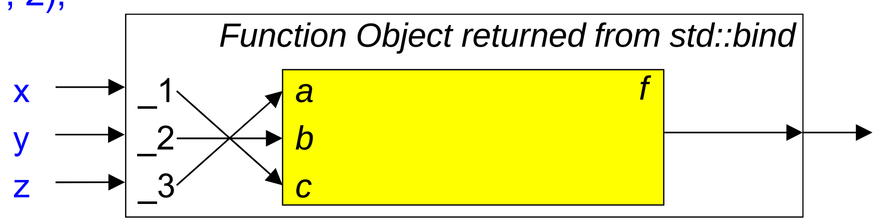
# Fun With bind

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);
```








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.

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

## 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.





All the examples on this page are taken from the foregoing bind discussion.

## 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)
                          )
              );
```





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.

# Lambdas vs. bind

Lambdas typically generate better code.

- Calls through bind involve function pointers ⇒ no inlining.
- Calls through closures allow full inlining.





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

# 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.





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

# New Algorithms for C++11

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.

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



**Slide 242**

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.

# New Algorithms for C++11

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/

**Slide 243**

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."

# Extended C++98 Algorithms in C++11

| | |
|---|---|
| swap | New overload taking arrays |
| min | New overloads taking initializer lists |
| max | New overloads taking initializer lists |





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

# Numeric→string Conversion Functions

<string> offers to_string for built-in numeric types:

```
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

- Results consistent with sprintf.
  - But string construction ⇒ exceptions possible.
- Types promotable to int (e.g., char, short, etc.) use int overload.
  - Presumably no advantage to specialized versions.

Equivalent overloads exist for to_wstring.





These function declarations are copied out of 21.5/7.

# string→Numeric Conversion Functions

Inverse conversions also available:

```
int                  stoi(const string& str, size_t *idx = 0, int base = 10);
long                 stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long        stoul(const string& str, size_t *idx = 0, int base = 10);
long long            stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long   stoull(const string& str, size_t *idx = 0, int base = 10);
float                stof(const string& str, size_t *idx = 0);
double               stod(const string& str, size_t *idx = 0);
long double          stold(const string& str, size_t *idx = 0);
```

- Thin wrappers around C's strto*type* functions (e.g., strtol, strtod, etc.).
  - ➧ As in C, no functions for short, unsigned short, unsigned int.
- Conversion impossible or range error ⇒ exception.

Equivalent overloads exist for wstring.





From a post at StackOverflow (http://tinyurl.com/8z3cybk):

> Each of these take a string as input and will try to convert it to a number. If no valid number could be constructed, for example because there is no numeric data or the number is out-of-range for the type, an error is thrown.
>
> If conversion succeeded and *idx is not 0, idx will contain the index of the first character that was not used for decoding. This could be an index behind the last character.
>
> Finally, the integral types allow to specify a base, for digits larger than 9, the alphabet is assumed (a=10 until z=35).

Regarding the lack of support for shorts, Pete Becker posted at StackOverflow (http://tinyurl.com/9d7qyju):

> These functions all do arithmetic on the value that's passed to them. Types that are smaller than int are promoted to int (or unsigned int) for arithmetic, so there's no computational benefit from having versions that take types smaller than int. (These came in through my proposal, so I know the history intimately.)

# 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.
- Conversion functions exist between strings and numbers.





There are some more subtle library changes in moving from C++03 to C++11, e.g., function objects used with STL algorithms in C++03 are generally prohibited from having side effects, while in C++11, 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 26.7.2/2 of C++11, the corresponding wording is "In the range [first,last], binary_op shall neither modify elements nor invalidate iterators or subranges."

# Overview

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



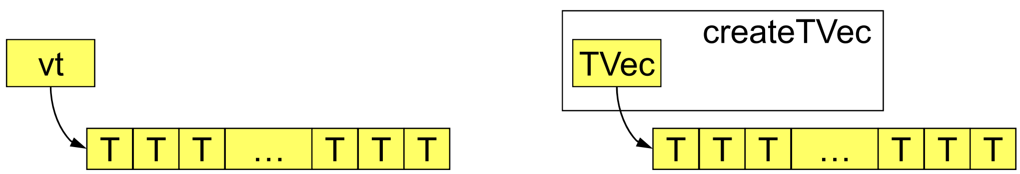


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

## 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
```







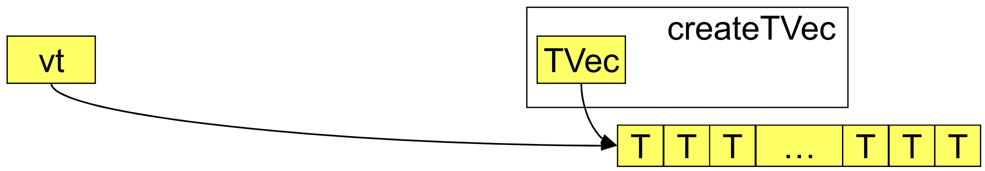

The diagrams on this slide make up a PowerPoint animation.

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.

# Move Support

*Moving* values would be cheaper:

```
TVec vt;
…
vt = createTVec();                    // move data in return value object
                                      // to vt, then destroy return value
                                      // object
```





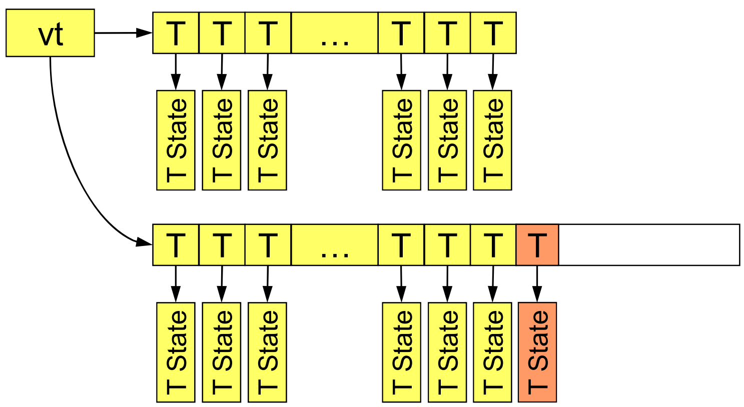



The diagrams on this slide make up a PowerPoint animation.

# 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
```







The diagrams on this slide make up a PowerPoint animation.

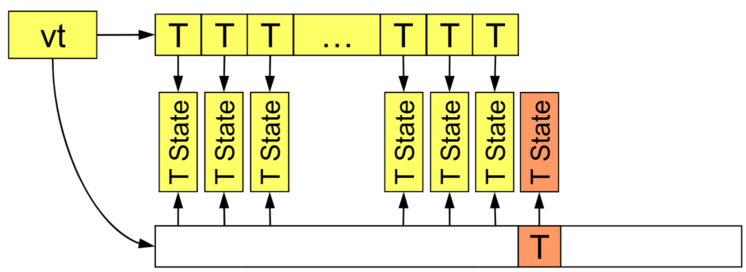
The new element has to be added to the new storage for the vector before the old elements are destroyed, because it's possible that the new element is a copy of an existing element, e.g. vt.push_back(vt[0]).

This slide assumes C++98 behavior, but it's worth noting that in C++11, the above analysis about having to insert the new element before relocating the existing elements applies only to push_back taking an lvalue. Per 17.6.4.9/1 bullet 3, push_back taking an rvalue may assume that the bound object has no aliases, and that means that e.g., vt.push_back(std::move(vt[0])) has undefined behavior.

## 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.

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



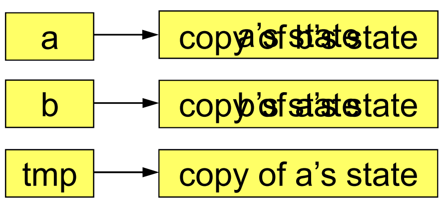
Slide 252

The diagrams on this slide make up a PowerPoint animation.

# 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/

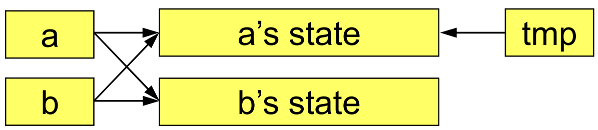


Slide 253

The diagrams on this slide make up a PowerPoint animation. That's why there appears to be overlapping text.

# 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
```








The diagrams on this slide make up a PowerPoint animation.

std::move is defined in <utility>.

# Move Support

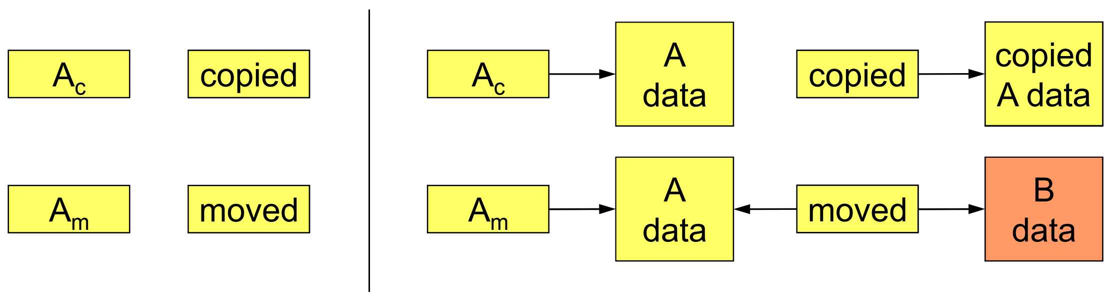
Moving most important when:

- 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.**





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.

## Simple Performance Test

Given

```
const std::string stringValue("This string has 29 characters");

class Widget {
private:
  std::string s;

public:
  Widget(): s(stringValue) {}
  ...
};

typedef std::vector<Widget> TVec;
```

consider these use cases again:

```
vt = createTVec();                // return/assignment of TVec

vt.push_back(T object);           // push_back onto full TVec
```

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



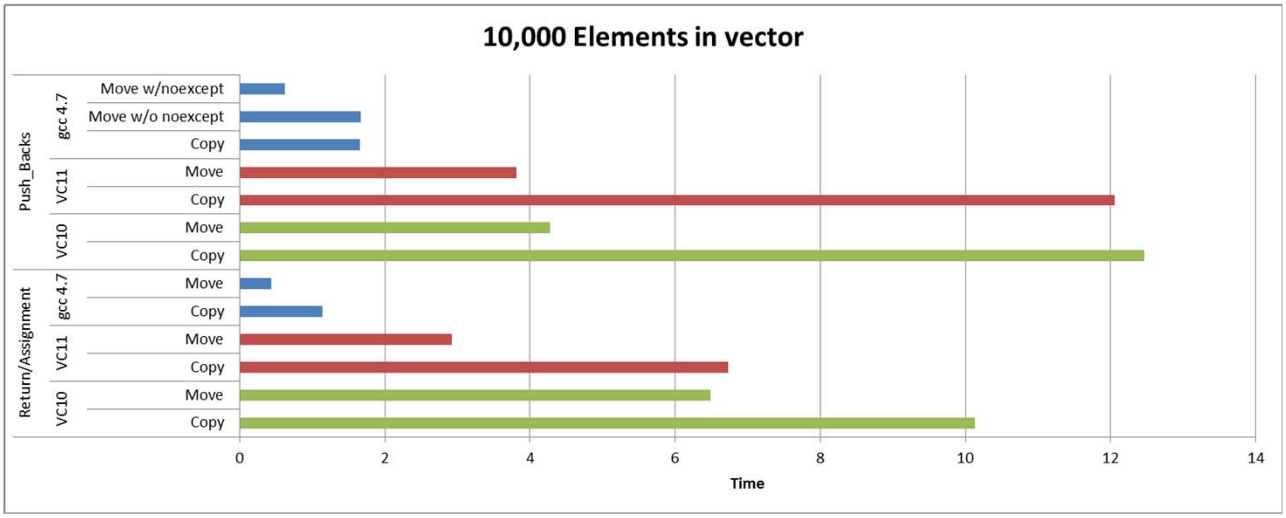
**Slide 256**

For the push_back test, the actual code used was:

```
std::vector<Widget> vw;
Widget w;

for (std::size_t i = 0; i < n; ++i) {      // append n copies
    vw.push_back(w);                        // of w to vw
}
```

As such, what's being push_backed is an lvalue, not an rvalue.

# Performance Data




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



Slide 257

All data gathered May 2012 on a Lenovo W510 laptop. VC11 data is for the beta release.

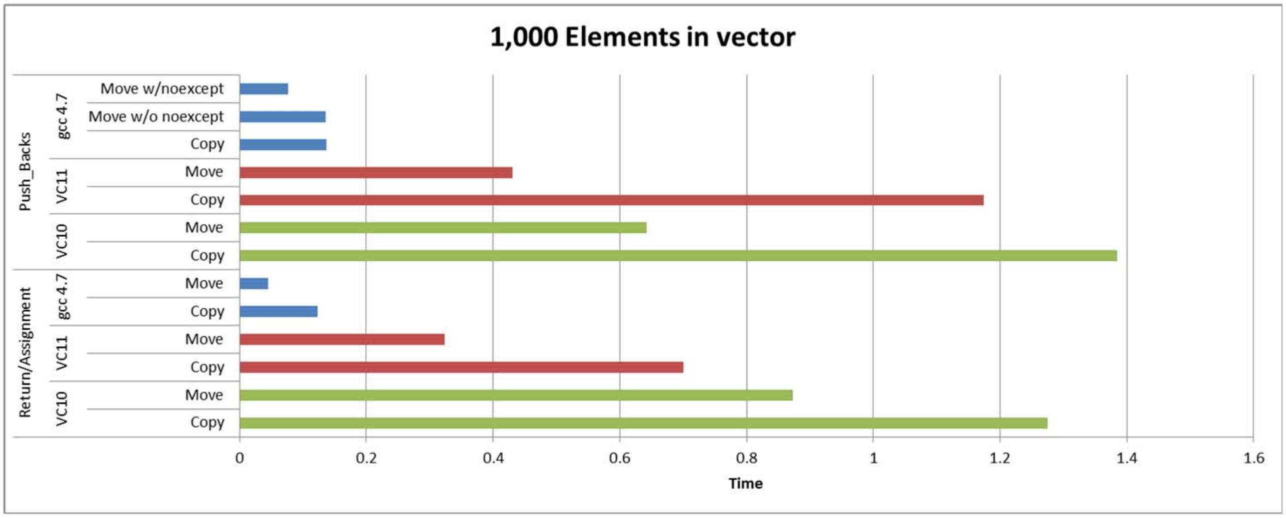
For the push_back test, the difference between copy and move pertains to the std::vector's element type (in my tests, Widget), because it's the values in the vector that are copied or moved from the vector's old memory to its new memory. For the return/assignment test, the copy/move distinction pertains to the std::vector itself, because that's what's being copied or moved. (When the vector itself is copied, its values are unconditionally copied.)

std::vector::push_back provides the strong guarantee, and because elements must be "shifted" from existing memory to new memory when there is no excess capacity, copy operations must be employed unless the move operations are known not to throw. Both VC10 and VC11 ignore this requirement; they perform moves unconditionally. gcc 4.7 moves only if the move operations are declared noexcept. With element types offering move operations but not declaring them noexcept, the push_back tests exhibit essentially the same performance as element types offering only copy operations (and in fact should perform identically), while tests on element types offering noexcept move operations are notably faster.

For the return/assignment test, the situation is different. I used std::vectors with equal allocators, which permits std::vector's move operator= to simply copy the internal pointers from the source vector to the target vector. The elements in the vector are neither copied nor moved, so it's immaterial whether their operations are declared noexcept.

Microsoft apparently made significant improvements in the performance of its library/compiler from VC10 to VC11 beta, especially for the smaller vector sizes on the following pages, but in each case, moving is notably faster than copying.

# Performance Data

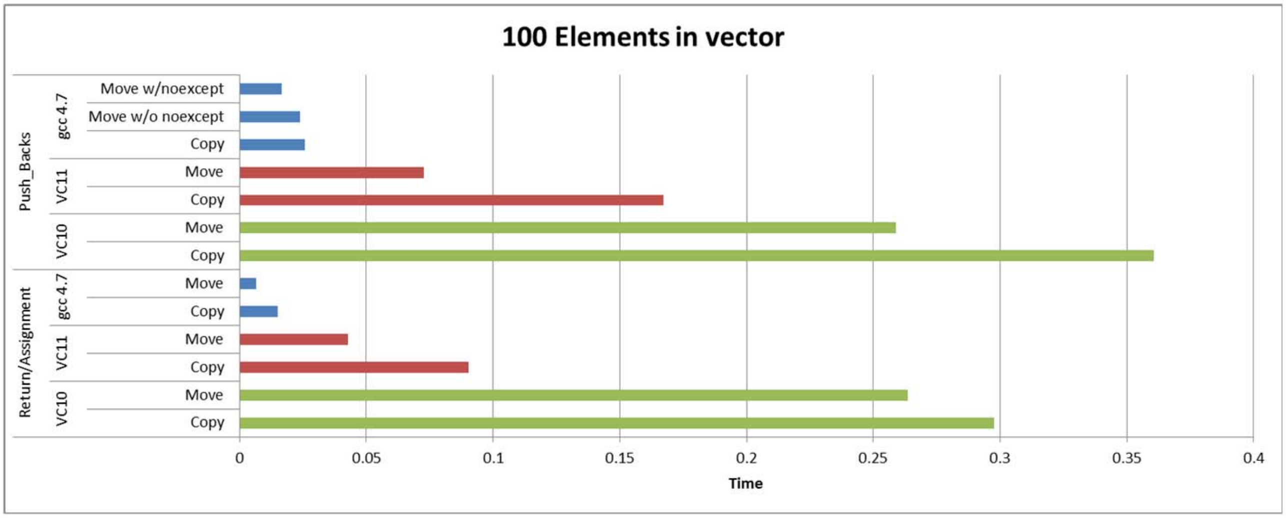







Scott Meyers, Software Development Consultant
http://www.aristeia.com/

# Performance Data




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





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

# Move Support

Lets C++ recognize move opportunities and take advantage of them.

- How recognize them?
- How take advantage of them?





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

# 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);  // f and str are lvalues,
                                 // 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&.




The this pointer is a named object, but it's defined to be an rvalue expression.

Per 5.1.1/1 in C++11, 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.

## 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
```





In some cases, it's known that an lvalue being copied will never be referenced again, and in those cases, C++11 treats the lvalue as an rvalue and permits copies to be turned into moves. In fact, C++11 permits the copy/move operations to be completely elided, but if compilers chose not to elide them, the lvalue must be treated as an rvalue. Situations where this apply include return statements and throw expressions. Details are in 12.8/31 and 12.8/32.

# 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 createTVec();              // as before
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++11");
f(s);                           // lvalue source: copy needed
```





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.

# Rvalue References

C++11 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.**





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.





General rules governing reference binding are in 8.5.3, and rules governing the interaction of reference binding and overloading resolution are in 13.3.3.1.4 and 13.3.3.2, especially 13.3.3.2/3 bullet 1 sub-bullet 4, 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. One way for an overload taking a const T&& parameter to arise is to pass a const rvalue to a function template taking a universal reference.

There was a time in draft C++11 when lvalues were permitted to bind to rvalue references, and some compilers (e.g., gcc 4.3 and 4.4 (but not 4.5 or later versions), 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 should be aware that some older compilers supporting rvalue references may implement the "version 1" rules instead of the version 2 rules.

# 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
```





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

# 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++11 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 :-)

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



The crux of why rvalue-references-to-const are rarely useful is the special handling accorded T&& parameters in 14.8.2.1/3:

> 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 C++11 20.8/2:

```
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;
```

# 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&);
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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&&) noexcept;               // move constuctor

  Widget& operator=(const Widget&);        // copy assignment op
  Widget& operator=(Widget&&) noexcept;    // 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
```





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

# Move Operations and noexcept

Move operations need not be noexcept, but it's preferable.

- Moves should be fast, and noexcept ⇒ more optimizable.
- Some contexts require noexcept moves (e.g., std::vector::push_back).
- Move operations often have natural noexcept implementations.

In this course, I declare move operations noexcept by default.



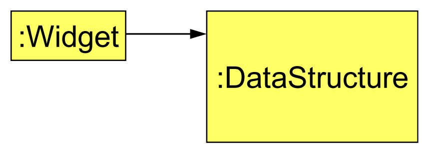


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

# Implementing Move Semantics

Move operations take source's value, but leave source in valid state:

```
class Widget {
public:
  Widget(Widget&& rhs) noexcept
  : pds(rhs.pds)                       // take source's value
   { rhs.pds = nullptr; }              // leave source in valid state
  Widget& operator=(Widget&& rhs) noexcept
  {
    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…





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) { swap(rhs, *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".

# 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++11 condones this.

- In contrast to copy operator=.

A fix is simple, if you are inclined to implement it:

```
Widget& Widget::operator=(Widget&& rhs) noexcept
{
  if (this == &rhs) return *this;  // or assert(this != &rhs);
    …
}
```





The condoning of "self-move-assignment yields undefined behavior" is in the standard at 17.6.4.9, and the motivation for it is 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/ .

# Implementing Move Semantics

Part of C++11's string type:

```
string::string(const string&);           // copy constructor
string::string(string&&) noexcept;       // move constructor
```

An incorrect move constructor:

```
class Widget {
private:
  std::string s;

public:
  Widget(Widget&& rhs) noexcept        // 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.





The std::string copy constructor is not declared noexcept, because that constructor may throw a std::bad_alloc or a std::length_error exception.

# Implementing Move Semantics

Another example:

```
class WidgetBase {
public:
  WidgetBase(const WidgetBase&);             // copy ctor
  WidgetBase(WidgetBase&&) noexcept;         // move ctor
  …
};

class Widget: public WidgetBase {
public:
  Widget(Widget&& rhs) noexcept              // move ctor
  : WidgetBase(rhs)                          // copies!
  { … }
  …
};
```

- rhs is an **lvalue**, because it has a name.
  - ➡ Its declaration as Widget&& not relevant!





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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) noexcept                    // move constructor
  : WidgetBase(std::move(rhs)),                    // request move
    s(std::move(rhs.s))                            // request move

  { … }
  Widget& operator=(Widget&& rhs) noexcept  // 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.





The move assignment operator on this page fails to worry about move-to-self.

## 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?





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

# Implementing std::move

std::move is simple – in concept:

```
template<typename T>                     // return as an rvalue
T&&                                      // whatever is passed
move(MagicReferenceType obj) noexcept    // in; must work with
{                                        // both lvalue/rvalues
  return obj;
}
```

Between concept and implementation lie arcane language rules.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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
```





This page describes behavior that I believe was and continues to be de facto standard for C++98/03 compilers. Daniel Krügler clarifies the official status, however:

> The page implies that C++98 did allow reference collapsing. This isn't quite right actually, especially the first example was not allowed as described in [temp.deduct] p2:
>
> "[..] Attempting to create a reference to a reference type [..]“
>
> Strictly speaking, even C++03 didn't allow that, but the core language was a bit fuzzy, therefore some compilers did allow it. The actual resolution was due to
>
> http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#106
> http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#540

# Reference Collapsing in Templates

C++11's rules take rvalue references into account:

- T& & ⇒ T& // from C++98
- T&& & ⇒ T& // new for C++11
- T& && ⇒ T& // new for C++11
- T&& && ⇒ T&& // new for C++11

Summary:

- Reference collapsing involving a & is always T&.
- Reference collapsing involving only && is T&&.

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



Slide 279

These rules are defined by 8.3.2/6 in C++11. They apply not only to template parameters, but also to types created via decltype and typedefs.

# 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) noexcept
{
  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&.





std::remove_reference is part of the type traits functionalityin C++11. It turns both T& and T&& types into T.

## 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) noexcept;     // possible move
                                          // instantiation

std::move(createTVec());                  // error!
```

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



**Slide 281**

Note that this page shows move as a function, not a template.

# std::move's Parameter Type

An rvalue reference doesn't, either.

- Lvalues can't bind to them.

```
TVec&& std::move(TVec&& obj) noexcept;       // 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&&.





Note that this page shows move as a function, not a template.

# Why Not Just Overload?

Overloading could solve the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T& lvalue) noexcept
{
  return static_cast<T&&>(lvalue);
}

template<typename T>
typename std::remove_reference<T>::type&&
move(T&& rvalue) noexcept
{
  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.

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



Slide 283

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.

# 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&&)
```

# Implementing std::move

std::move's parameter is thus T&&:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept
{
  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.





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

# Implementing std::move

A cast eliminates the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept
{
  using ReturnType =
    typename std::remove_reference<T>::type&&;

  return static_cast<ReturnType>(obj);
}
```

This is a correct implementation.





It's possible to avoid repeating std::move's return type in the body of the function by applying decltype to move itself:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept
{
  using ReturnType = decltype(std::move(obj));

  return static_cast<ReturnType>(obj);
}
```

Per Daniel Krügler's email of 11/1/2011, the fully qualified call to std::move in the definition of ReturnType is needed to avoid the possibility of inadvertantly referring to a user-defined move made visible via ADL.

In C++14, the return type in an implemenation could be replaced by auto, but it would have to be constexpr auto, because in C++14, both std::move and std::forward are constexpr functions.

# T&& Parameters in Templates

Compare conceptual and actual std::move declarations:

```
template<typename T>                          // conceptual
T&&
move(MagicReferenceType obj) noexcept;

template<typename T>                          // actual
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept;
```

T&& really is a magic reference type!

- For lvalue arguments, T&& becomes T& ⇒ lvalues can bind.
- For rvalue arguments, T&& remains T&& ⇒ rvalues can bind
- For const/volatile arguments, const/volatile becomes part of T.
- T&& parameters can bind *anything*.





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

# Universal References

Two conceptual meanings for T&& syntax:

- **Rvalue reference**. Binds rvalues only.

```
void f(Widget&& param);          // takes only non-const rvalue
```

- ***Universal*** **reference**. Binds lvalues and rvalues.

```
template<typename T>
void f(T&& param);               // takes lvalue or rvalue,
                                 // const or non-const
```

  - My term (which I hope will become common).
  - Really an rvalue reference in a reference-collapsing context.





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

# Universal References

Function templates with a T&& parameter need not generate functions taking a T&& parameter!

```
template<typename T>
void f(T&& param);              // universal reference

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&&)
```





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

## auto&& ≡ T&&

auto type deduction ≡ template type deduction, so auto&& variables are also universal references:

```
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&
```




auto type deduction isn't identical to template parameter type deduction, because auto can deduce std::initializer_list types, but templates can't.

# Rvalue References vs. Universal References

Read code carefully to distinguish them.

- Both use "`&&`" syntax.
- Behavior is different:
  - Rvalue references bind only rvalues.
  - Universal references bind lvalues *and* rvalues.
    - I.e., may become either `T&` or `T&&`, depending on initializer.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

## Rvalue References vs. Universal References

Signs of a universal reference:

- Syntactic form: T&&.
- Type deduction for T.

Consider std::vector:

```
template<class T, class Allocator=allocator<T>>    // from C++11
class vector {                                      // Standard
public:
  ...
  void push_back(const T& x);                 // lvalue reference
  void push_back(T&& x);                      // rvalue reference!

  template<class... Args>
  void emplace_back(Args&&... args);          // universal reference
  ...
};
```





[The syntax for variadic templates has not yet been introduced.]

In push_back, T&& is an rvalue reference, because there's no type deduction involved. T is determined by the type of the vector (i.e., by T in vector<T>), not by the argument passed to push_back.

## 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.

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



Gadget's move constructor is not declared noexcept, because it invokes Widget's copy constructor, which is itself not declared noexcept (which is unsurprising for a copy constructor).

Move requests on types that are not copyable but also lack move support will fail to compile.

# Move is an Optimization of Copy

If Widget adds move support:

```
class Widget {
public:
  Widget(const Widget&);              // copy ctor
  Widget(Widget&&) noexcept;          // move ctor
};

class Gadget {                        // as before
public:
  Gadget(Gadget&& rhs) noexcept
  : 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.

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



**Slide 294**

Gadget's move constructor has now been declared noexcept, because it's invoking Widget's move constructor, which is noexcept. This demonstrates that if a class changes its interface to complement a copy operation that may throw with a move operation that is noexcept (as in the case of Widget), callers may be able to add noexcept to functions invoking what used to be copy operations but are now move operations.

## 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 when moving faster than copying.**
- **Use std::move for lvalues that may safely be moved from.**





Both move and copy operations may throw, and the issues associated 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.

Generic (i.e., template-based) code invoking a move operation on an unknown type T where the exception-safety guarantee offered by T's move operation affects the guarantee the generic code can offer should offer a *conditional* guarantee, e.g., the generic code is nothrow only if T's move operation is nothrow. There is nothing move-specific about this idea; it's suitable for all generic code that invokes unknown operations. Numerous examples can be found in the standard library by searching for "Throws: Nothing unless" or "Throws: Nothing if".

N2983 explains how std::move_if_noexcept can be used in the 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.

## 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.





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."

## 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.





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.

# 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=
```

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



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

# 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) noexcept;       // user-declared move ctor
                                         // ⇒ no implicit copy ops;
  Widget3&                               // user-declared move op=
    operator=(Widget3&& rhs) noexcept;   // ⇒ no implicit copy ops
};
```

# 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!
```





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

# 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++11.
- Declaring a destructor ⇒ no implicit copy ops.
  - Too big a change for C++11.

However:

- Implicit copy ops **deprecated** in classes with user-declared copy, move, or dtor operations.
  - Compilers may issue warnings.





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

# 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
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.
- std::moves on shared lvalues require synchronization.
  - E.g., use of std::lock_guard or std::unique_lock.





MT = "Multi-threaded".

Because move operations modify the moved object, they are writes to that object, hence race with any read or write on the same object.

# 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 synchronization.





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

# 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.





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

## 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) noexcept
  { 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;
};
```

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



**Slide 306**

setName is declared noexcept, because std::string's move assignment operator is noexcept. setCoords is not declared noexcept, because std::vector's move assignment operator is not declared noexcept.

# Beyond Move Construction/Assignment

A setter for both ⇒ two parameters ⇒ four overloads:

```
class Widget {
public:
  void setNameAndCoords(const std::string& newName,            // lvalue
                        const std::vector<int>& newCoords);    // lvalue

  void setNameAndCoords(const std::string& newName,            // lvalue
                        std::vector<int>&& newCoords);         // rvalue

  void setNameAndCoords(std::string&& newName,                 // rvalue
                        const std::vector<int>& newCoords);    // lvalue

  void setNameAndCoords(std::string&& newName,                 // rvalue
                        std::vector<int>&& newCoords);         // rvalue
  ...
};
```

$n$ parameters requires $2^n$ overloads.

- Impractical for large $n$.
- Boring/repetitive/error-prone for smaller $n$.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Perfect Forwarding

Goal: one function that "does the right thing:"

- Copies lvalue args, moves rvalue args.

Solution is a **perfect forwarding** function:

- Templatized function forwarding T&& params to members:

```
class Widget {
public:
  template<typename T1, typename T2>
  void setNameAndCoords(T1&& n, T2&& c)
  {
    name = std::forward<T1>(n);           // forward n to std::string::op=
    coordinates = std::forward<T2>(c);    // forward c to std::vector::op=
  }
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```





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

# Perfect Forwarding

Once again:

- A *templatized function forwarding* T&& *params* to members:

```
class Widget {
public:
  template<typename T1, typename T2>
  void setNameAndCoords(T1&& n, T2&& c)
  {
    name = std::forward<T1>(n);           // forward n to std::string::op=
    coordinates = std::forward<T2>(c);  // forward c to std::vector::op=
  }
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```

Effect:

- Lvalue arg passed to n ⇒ std::string::operator= receives lvalue.
- Rvalue arg passed to n ⇒ std::string::operator= receives rvalue.
- Similarly for c and std::vector::operator=.





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

## Perfect Forwarding

Despite T&& parameters, code fully type-safe:

- Type compatibility verified upon instantiation.
  - ➡ Only std::string-compatible types valid for n.
  - ➡ Only std::vector<int>-compatible types valid for c.

More flexible than typed parameters.

- Accepts/forwards all compatible parameter types.
  - ➡ n can be assigned std::string, char*, or const char*.





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

# Perfect Forwarding

Flexibility can be removed via static_assert (described soon).

- E.g. perfect-forwarding setName could look like this:

```
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/





[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.

# 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.

Conceptual implementation is easy:

```
template<typename T>                          // For lvalues (T is T&),
T&& std::forward(T&& param) noexcept          // 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.





The C++11 specification for std::forward (20.2.3/1-2) consists of two templates:

```
template <class T>
T&& forward(typename remove_reference<T>::type& t) noexcept
{ return static_cast<T&&>(t); }

template <class T>
T&& forward(typename remove_reference<T>::type&& t) noexcept
{ return static_cast<T&&>(t); }
```

The first template is designed for lvalues, the second for rvalues.

The use cases that std::forward is designed to accept and reject are detailed in N2951 and N3143 (see Further Information), but one goal is to disable template argument deduction, so that std::forward(*expr*) won't compile; std::forward<T>(*expr*) must be used instead. This goal is achieved by declaring the function parameter type to be remove_reference<T>::type, because there's no type deduction performed on dependent types. In earlier implementations (corresponding to slightly different specifications for std::forward) and in earlier versions of these training notes, this goal was achieved via the use of std::identity instead of std::remove_reference.)

## Perfect Forwarding

- Applicable only to function templates.
- Preserves arguments' lvalueness/rvalueness/constness when forwarding them to other functions.
- Implemented via std::forward.





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.

# 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 and destructor not user-declared.





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

# default Member Functions

Generated versions are:

- Public
- Inline
- Non-explicit

defaulted member functions have:

- **User-specified declarations with the usual compiler-generated implementations.**





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

# 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&&) noexcept = default;       // declare move ctor, use
                                             // default impl.
  ...
};
```





As far as I know, the only difference between

```
Widget() = default;
```

and

```
Widget() {}
```

is that the former yields a *user-declared* constructor, while the latter a *user-provided* constructor. Classes with user-declared constructors may be trivial, but classes with user-provided constructors may not be. So =default is preferable to {} when the class needs to be trivial. A discussion of this issue can be found at http://stackoverflow.com/questions/13576055/how-is-default-different-from-for-default-constructor-and-destructor.

# 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&&) noexcept = default;  // declare as private
  ...
};
```

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



**Slide 317**

There does not appear to be any practical behavioral difference between

```
virtual ~Widget() = default;
```

and

```
virtual ~Widget() {}.
```

One tiny semantic difference is that the former is defined only if used, and this can change whether the code compiles (in a very edgy corner case). Details are at http://stackoverflow.com/questions/13576055/how-is-default-different-from-for-default-constructor-and-destructor.

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.

# delete Functions

deleted functions are defined, 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
};
```




Per 8.4.3/1, "=delete" is a function *definition*, so all the functions on this page are not just declared, they are also defined.

"=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.

I'm not declaring Gadget's move operation noexcept, because (1) it wouldn't make any difference, and (2) I don't want people to get into the habit of blindly adding noexcept to all move operations. Such operations should be declared noexcept only if the author has verified that they cannot yield exceptions.

Function templates 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.

# 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*)
```





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

# 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
```





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++11. (Literal types are defined in 3.9/10 in C++11).)

## 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.





As noted earlier, I am unaware of any practical difference between declaring the Widget constructor as shown or simply defining it with an empty function body.

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
```

# 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;
};
```





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.

# 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(w), size(w.size), flex(w.flex)
{
  registerObject(this);
}
```





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.

Per normal convention, the Widget copy constructor constructs its base class part using its parameter instead of by calling calcBaseVal (as is done by the other constructors).

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

## 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)                              // #4 (same as
  : Base(w), size(w.size), flex(w.flex)                //       on last
  { registerObject(this); }                            //       slide)

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;
};
```





A constructor that delegates to another constructor may not do anything else on its member initialization list. That's why the copy constructor (#4) can't forward to #5: it needs its member initialization list to specify how to initialize its base class.

A constructor that delegates to itself (directly or indirectly) yields an "ill-formed" program.

# 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(): Widget(defaultFlex)
  {
    makeLogEntry("Widget default constructor");
  }
  ...
};
```





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

# 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++11

  using Base::Base;       // okay in C++11 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++11 due to ctor inheritance

Derived d2(5, 10);        // normal use of Derived::Derived(int, int)
```





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.

# 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
```





The error is diagnosed at the point of use of the inheriting constructor (i.e., the declaration of d).

## 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
```

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



**Slide 328**

It's not quite true that Derived::x and Derived::y are uninitialized. 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.

# 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
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.
- Nonstatic data members may have default initialization values.
- Delegating constructors call other constructors.
- Inherited constructors come from base classes.





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

# 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/

# 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.





Diagnostic messages may use any kind of string literal (e.g., u"...", u8"...", and R"...".)

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)
```

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

# 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)");
  …
};
```





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

# explicit Conversion Functions

explicit now applicable to conversion functions:

```
class Widget {
public:
  explicit Widget(int i);                    // C++98 and C++11
  …
  explicit operator std::string() const;     // C++11 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
```





This slide shows uses of static_cast, but other cast syntaxes (i.e, C-style and functions-style) would behave the same way.

## 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!
```




The "explicitness" of an operator bool function is ignored in cases where the standard calls for something being "contextually converted" to bool.

# Variadic Templates

Templates may now take arbitrary numbers and types of parameters:

```
template <class... Types>                    // std::tuple is in C++11
class tuple;

template<class T, class... Args>             // std::make_shared is
shared_ptr<T>                                // in C++11
  make_shared(Args&&... params);
```

Non-type parameters also okay:

```
template<typename T, std::size_t... Dims>    // this template is
class MultiDimensionalArray;                 // not in C++11
```

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);
```

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



**Slide 336**

The declarations for tuple and make_shared are copied from C++11, which is why they use "class" instead of my preferred "typename" for template type parameters. In C++11, the function parameter pack is named "args", but I've renamed it to "params" to make it easier to distinguish verbally from the template parameter "Args" (which is in C++11).

## 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/



Slide 337

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 C++11),

```
template<class... TTypes, class... UTypes>
bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

# Parameter Packs

Manipulation often 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 = 1 + Count<Rest...>::value;
};

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
```





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

# Parameter Packs

Count purely an exercise; C++11's sizeof... does the same thing:

```
template<typename… Types>
struct VerifyCount {
  static_assert(Count<Types…>::value == sizeof...(Types),
                "Count template is broken!");
};
```

Unpack (...) and sizeof... only two operations for parameter packs.





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

# 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.





This example passes everything by const T&, but perfect forwarding would probably be a better approach.

## 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$_n$);





The ellipsis is always at the end of the pattern.

# 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(std::forward<PTypes>(params))...);  // call normalize
}                                                    // on each
                                                     // unpacked elem
```

Call to func expands to:

$$\texttt{func(normalize(std::forward<PTypes}_1\texttt{>(params}_1\texttt{)),}$$
$$\texttt{normalize(std::forward<PTypes}_2\texttt{>(params}_2\texttt{)),}$$
$$\texttt{...,}$$
$$\texttt{normalize(std::forward<PTypes}_n\texttt{>(params}_n\texttt{)));}$$





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

# 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<$T_0$, $T_1$, …, $T_{n-1}$> needs *n* fields, each of correct type.
    - std::tuple<$T_0$, $T_1$, …, $T_{n-1}$> inherits from std::tuple<$T_1$, $T_2$,…, $T_{n-1}$>





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. std::common_type is defined in terms of the behavior of the ?: operator, i.e., std::common_type returns the same type as cascading ?: operators invoked on the same set of types.

std::tuple need not use inheritance to generate its fields. Composition is also a possibility. However, the recursive structure generation exemplified in the following slides terminates with an empty class, and using inheritance, the empty base optimization makes it possible to avoid setting aside any space for this class. With composition, this optimization is not possible.

## 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.
};
```

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



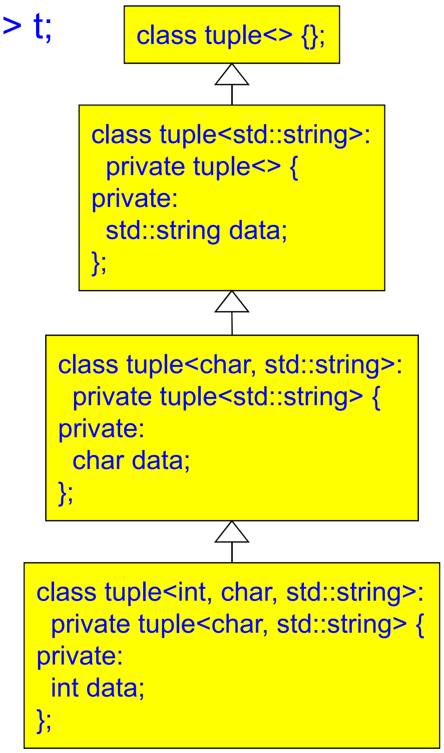
Slide 344

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.

# Generated Hierarchy

```
std::tuple<int, char, std::string> t;
```








Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Sketch of std::tuple_element

Yields the $k^{th}$ type in a std::tuple<$T_0$, $T_1$, …, $T_{n-1}$>.

```
template <std::size_t k,
          typename Tuple>                        // declare
struct tuple_element;

template <typename T,
          typename... TRest>
struct tuple_element<0, tuple<T, TRest...>> {    // base case
  typedef T type;
};

template <std::size_t k,
          typename T, typename... TRest>
struct tuple_element<k, tuple<T, TRest...>> {    // general case
  typedef typename tuple_element<k-1, tuple<TRest...>>::type
          type;
};
```

Real implementations more complex (e.g., handle cv-qualified tuples).





[People rarely ask how this is implemented, but they often ask how std::get is implemented. I show that on the next page, but std::tuple_element is std::get's return type, so we cover that first.]

# Sketch of std::get

```
template <std::size_t k, typename Tuple>
// base case
typename std::enable_if<k == 0,
                                   typename tuple_element<k, Tuple>::type&
                                 >::type
get(Tuple& t)
{
   return t.data;                              // get must be a friend of tuple
}

template <std::size_t k, typename T, typename... TRest>
// general case
typename std::enable_if<k != 0,
                                   typename tuple_element<k, tuple<T, TRest...>>::type&
                                 >::type
get(tuple<T, TRest...>& t)
{
  tuple<TRest...>& base = t;                   // get must be a friend of tuple
  return get<k - 1>(base);
}
```

This code based on an example by Andrei Alexandrescu.

- Again, real implementations more complex.





In the first version of get, friendship with tuple is required in order to access tuple's data data member. In the second version, it's necessary to perform the conversion from derived class to private base class.

[The next page shows how to declare the get functions as friends of tuple.]

# Declaring gets friends

Two keys:

- friend immediately precedes return type (i.e., std::enable_if).
- Avoid clashing template parameter names.

```
template<typename T, typename… TRest>                    // as
class tuple<T, TRest…>: private tuple<TRest…> {          // before
...
template <std::size_t k, typename Tuple>
friend
typename std::enable_if<k == 0,
                        typename tuple_element<k, Tuple>::type&
                       >::type
get(Tuple& t);
template <std::size_t k, typename T2, typename... TRest2>
friend
typename std::enable_if<k != 0,
                        typename tuple_element<k, tuple<T2, TRest2...>>::type&
                       >::type
get(tuple<T2, TRest2...>& t);
};
```

# 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).





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# decltype

Common 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
}
```





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.

# 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.





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

# 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
```





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

## 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)...);
}
```

With return type deduction (C++14):

```
template<typename F, typename... Ts>          // C++14 only
auto logAndInvoke(std::ostream& os,
                  F&& func, Ts&&... args)
{
  os << std::chrono::system_clock::now();
  return func(std::forward<Ts>(args)...);
}
```

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





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).

# 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);
}
```

With return type deduction (C++14):

```
template<typename T1, typename T2>
auto mult(T1&& a, T2&& b)
{
  return std::forward<T1>(a) * std::forward<T2>(b);
}
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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-return-type problem in C++11.

auto is for everybody. decltype is primarily for template authors.

# Perfect Forwarding Lambdas (C++14)

decltype essential for perfect forwarding lambdas:

- Use auto&& for parameters.
  - Variadic, if needed.
- Use decltype for type to pass to std::forward.

```
auto loggingWidgetMaker =
  [](auto&&... params)
  {
    makeLogEntry("Creating std::unique_ptr<Widget>...");
    return
      std::make_unique<Widget>(std::forward<decltype(params)>(params)...);
  };


std::unique_ptr<Widget> up = loggingWidgetMaker(arg1, arg2, arg3, ... argn);
```

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



Slide 356

The usual convention is to pass an lvalue reference type to std::forward for lvalues and a non-reference type for rvalues. Here, we follow the convention for lvalues, but for rvalues, we pass an rvalue reference type. This works, because the reference-collapsing rules yield the same result any time a non-reference is replaced with an rvalue reference. I discuss this more fully in my 5/6/13 blog entry, "C++14 Lambdas and Perfect Forwarding."

## Alignment Control

Useful for e.g., aligning objects on cache line or page boundaries.

Based on alignof and alignas:

```
class Widget { ... };
std::size_t alignment = alignof(Widget);  // query alignment
alignas(128) Widget w;                    // specify object alignment
alignas(64) class Gadget { ... };         // specify type alignment
Gadget g;                                 // 64-byte aligned
```

alignas may also take a type:

```
alignas(Widget) int x;          // ≡ "alignas(alignof(Widget)) int x;"
```

Alignment may be made only stricter, not weaker:

```
alignas(1) double d;            // error if double normally requires
                                // alignment stricter than 1
```





alignof and alignas are new keywords in C++11.

alignas may be applied to enum types as well as to class types (i.e., classes, structs, unions).

C++11 offers TR1's alignment_of, but it's semantically redundant, because std::alignment_of<T>::value ≡ alignof(T).

A C++11 alignment feature not covered in these materials (except in this note) is function std::align, which, per 20.6.5, adjusts a pointer into a buffer so that it points at a specified alignment boundary.

# From TR1: std::aligned_storage

Defines types for aligned-but-uninitialized buffers:

```
typedef std::aligned_storage<sizeof(Widget), alignof(Widget)>::type
          BufType1;

BufType1 buf1;          // buf1 can hold a Widget (size & alignment are OK)

new (&buf1) Widget;     // construct Widget in buf1


typedef std::aligned_storage<sizeof(Gadget), alignof(Gadget)>::type
          BufType2;

BufType2 buf2[10];                     // buf2 can hold 10 Gadgets

for (int  pos = 0; pos < 10; ++pos )   // construct Gadgets
  new (&buf2 + pos) Gadget;            // in buf2
```

Note: buf1 and buf2 not on heap.

- Facilitates use of stack-based buffers.
  - Like alloca, except:
    - Standardized
    - Buffer size must be known during compilation





The buffers aren't necessarily truly uninitialized. Objects of types defined by std::aligned_storage that have static storage duration will be zero-initialized, as usual.

# std::aligned_storage

Expected implementation builds on alignas:

```
template <std::size_t Len,
          std::size_t Alignment = default-alignment>   // see below
struct aligned_storage {
  typedef struct {
    alignas(Alignment) unsigned char __data[Len];
  } type;
};
```

- std::aligned_storage<L, A>::type ≡ unsigned char array of size L aligned via alignas<A>.
- When Alignment omitted, type has strictest alignment of all types T where sizeof(T) ≤ Len.
  - ➡ Presumably requires compiler support.





The remark that this is the expected implementation is based on 20.9.7.6/1, as is the shown code.

*default-alignment* is not present in TR1's aligned_storage. In TR1, two parameters must always be specified.

# std::aligned_union

Like std::aligned_storage, but takes types instead of alignment.

- Resulting type at least as large as specified.
  - Size of 0 ⇒ big enough to hold all specified types.

```
typedef std::aligned_union<0, int, double, Widget, Gadget>::type
        BufType;

BufType buf;            // buf can hold int, double, Widget or Gadget
                        // (size & alignment are OK)

new (&buf) Widget;      // construct Widget in buf
```

Useful for things like Boost.Variant:

```
template<typename Ts...>             // conceptual implementation!
class Variant {
private:
  typename std::aligned_union<0, Ts...>::type buffer;  // buffer can
  ...                                                  // hold anything
};                                                     // in Ts

Variant<int, double, Widget, Gadget> v;
```





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.
- Alignment control options include alignof, alignas, std::aligned_storage and std::aligned_union.





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

# 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/

# More C++11 Features

- 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>.
- Generalized constant expressions (constexpr).
- User-defined literals (e.g., 10_km, 30_sec).





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.).

# Still More C++11 Features

- 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 preventing derivation and for constraining virtual function overrides.
- Attributes express special optimization opportunities and provide a standard syntax for platform-specific extensions.




The identifiers override and final are contextual keywords. Both classes and virtual functions may be declared final. final classes may not be used as bases (per 9/3), and final virtual functions may not be overridden in derived classes. A virtual function declared override must override a function in a base class. The following is an amalgam of examples from 10.3/4-5:

```
struct B {
   virtual void f() const final;
   virtual void f(int);
};

struct D : B {
   void f() const;              // error: D::f attempts to override final B::f
   void f(long) override;       // error: wrong signature overriding B::f
   void f(int) override;        // OK
};
```

# Coming in C++14 (Probably)

- Binary literals, e.g., 0b1011111100001010.
- Dynamic arrays: STL containers with size fixed at construction.
- Runtime-sized arrays: arrays w/o size set during compilation.
- std::optional represents values that may not exist.
- Variable templates enable parameterized constants,. e.g., pi<float>, pi<long double>.
- Standard user-defined literal suffixes, e.g., "Hello"s, 100ns, etc.
- std::quoted manipulators facilitate IO of strings containing whitespace.





The "probably" is because C++14 is currently in only draft form.

Binary literals begin with either 0B or 0b.

The difference between runtime-sized arrays and dynamic arrays is that runtime-sized arrays are part of the *language*, are always stack-allocated, and have no STL interface (e.g., no iterators, no size function), while dynamic arrays are part of the *library* (std::dynarray), may use heap allocation, and have an STL interface. Neither can be resized after construction, hence neither support insertion nor erasing functions.

Standard C++14 suffixes for standard library types are s (for string literals defining a std::string object) and, per 20.13.5.8/1, "h, min, s, ms, us, ns denote duration values of the corresponding types hours, minutes, seconds, milliseconds, microseconds, and nanoseconds respectively if they are applied to integral literals. If any of these suffixes are applied to a floating point literal the result is a chrono::duration literal with an unspecified floating point representation."21.7/5 notes that "The same suffix s is used for chrono::duration literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals."

## 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 is deprecated.
- Implicit generation of copying functions for classes with other copying functions or a destructor is deprecated.
- Exception specifications are deprecated.
  - ➡ noexcept replaces "throw()".
- auto_ptr is deprecated. (Use unique_ptr instead.)
- bind1st/bind2nd are deprecated. (Use bind or lambdas instead.)
- ptr_fun, mem_fun, mem_fun_ref, unary_function, and binary_function are deprecated.
  - ➡ Neither bind nor function require them.




The full set of deprecated features is summarized in C++11 Annex D, but not all are newly deprecated. Some were already deprecated in C++98/03.

In C++11, old-style exception specifications ("throw (...)") are known as *dynamic-exception-specification*s, and new-style specifications ("noexcept(...)") as simply *exception-specification*s.

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. "

# 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/

# Further Information

The C++11 standard:

- *Programming Languages — C++*, Document 14882:2011, ISO/IEC, 2011-09-01.
  - ISO product page: http://tinyurl.com/5soe87 (CHF238 ≡ ~$245).
  - ANSI product page: http://tinyurl.com/8m6vb5m ($30).
  - First post-Standard draft: http://tinyurl.com/6wkboer (free).
    - Essentially identical to the standard.





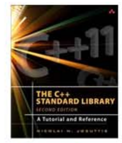
Many sources listed in this section have no URLs, because they are easy to find via search engine. The fewer URLs I publish, the fewer will be broken when target sites reorganize.

Prices shown for the C++11 standard are as of August 20, 2012.

# Further Information

C++11 in General:

- "C++11," *Wikipedia*.
- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.




- *C++11 - the recently approved new ISO C++ standard*, Bjarne Stroustrup, http://www.stroustrup.com/C++11FAQ.html.
- cppreference.com.
  - ➡ Reference source for standard C++ and standard C.
- "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.
- "C++0X with Scott Meyers," *Software Engineering Radio* (Audio), 5 April 2010.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Further Information

Counting words using C++11:

- "Jumping into C++," Eric Battalio, *Visual C++ Team Blog*, 18 January 2013.
  - Shows a similar program; blog comments very worthwhile.
- "The Great C++ Wordament: Meet Windows," Kenny Kerr, *kennykerr.ca*, 21 January 2013.
  - Reaction to above; optimizes code in various ways.





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

# 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.





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

# Further Information

Enhanced enums:

- "Strongly typed enumerations," *Wikipedia*, http://tinyurl.com/7szjdey.
  - Subsection of entry for "C++11".
- "enum class -- scoped and strongly typed enums," Bjarne Stroustrup, http://tinyurl.com/2cvylcc.
  - Subsection of "C++11 – the recently approved new ISO C++ standard."
- "Closer to Perfection: Get to Know C++11 Scoped and Based Enum Types," Danny Kalev, *smartbear.com*, 6 February 2013.
- "Better types in C++11 - nullptr, enum classes (strongly typed enumerations) and cstdint," Alex Allain, CPrograming.com.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.
- "Comments on 'C++0x feature support in GCC 4.5'," *reddit*, 13 March 2011, http://tinyurl.com/4xobt2r.
  - Contains useful discussion of pros/cons of UTF-16.
- "Unicode and your application," QbProg, *Do you really want me to write code like that?!* (blog), multi-part series, 11 November 2012 – 6 January 2013.





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

# 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.





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

# 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.
- "Efficient Use of Lambda Expressions and std::function," Cassio Neri, *Dr. Dobb's*, 1 January 2012.
- "Stateless Lambdas and the Old Guard," Kenny Kerr, *Kenny Kerr* (blog), 22 August 2012.





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

# Further Information

Lambdas in C++14:

- "Generic (Polymorphic) Lambda Expressions (Revision 3)," Faisal Vali *et al.*, C++ Standardization Committee Document N3649, 19 April 2013.
- "Generic lambda-capture initializers, supporting capture-by-move," Ville Voutilainen, C++ Standardization Committee Document N3610, 15 March 2013.
  - ➡ High-level feature overview.
- "Wording Changes for Generalized Lambda-capture," Daveed Vandevoorde and Ville Voutilainen, C++ Standardization Committee Document N3648, 17 April 2013.
  - ➡ Official wording for the feature.
- "C++14 Lambdas and Perfect Forwarding," Scott Meyers, *The View from Aristeia* (blog), 6 May 2013.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Further Information

Deduced Function Return Types (C++14):

- "Return type deduction for normal functions," Jason Merrill, C++ Standardization Committee Document N3638, 17 April 2013.

noexcept:

- "Exception Specifications in C++ 2011," Dietmar Kühl, *Overload*, June 2011.
- "Using noexcept," Andrzej Krzemieński, *Andrzej's C++ blog*, 10 June 2011.



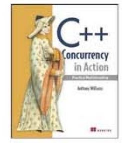


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

# Further Information

Concurrency:

- *C++ Concurrency in Action*, Anthony Williams, Manning Publications, 2012.




- "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.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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.
- "Async Tasks in C++11: Not Quite There Yet," Bartosz Milewski, *Corensic Multicore Concurrency Blog*, 10 October 2011.





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

# Further Information

C++14's std::shared_mutex/std::shared_lock:

- "Shared locking in C++, Revision 2," Howard Hinnant et al, C++ Standardization Committee Document N3659, 19 April 2013.
  - N3568 (linked to by above) provides essential background.
- "Real-World Concurrency," Bryan Cantrill and Jeff Bonwick, *ACM Queue*, September 2008.
  - Includes usage guidance for reader/writer locks.

# 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.
  - 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/.





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

# 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.
- "STL11: Magic && Secrets," Stephan T. Lavavej, *Going Native 2012*, 2 February 2012.
  - Describes "We Know Where You Live" optimization.
- "Does the WKWYL optimazion in make_shared<>() introduce a penalty for some multithreading applications?," *StackOverflow*, 15 January 2013.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Further Information

unique_ptr:

- "C++11: unique_ptr<T>," Mark Nelson, *http://marknelson.us/*, 24 June 2012.
- "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++11. 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.

forward_list:

- "Singly Linked Lists in C++," Thomas Guest, *Word Aligned* (blog), 7 February 2013.





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

# Further Information

Hash Tables (unordered_ containers):

- "STL and TR1: Part III," Pete Becker, *C/C++ Users Journal*, February 2006.
- "Hash Functions for C++ Unordered Containers," Mark Nelson, *Blog at http://marknelson.us/*, 3 September 2011.

Regular Expressions (regex):

- "Regular Expressions," Pete Becker, *Dr. Dobb's Journal*, May 2006.
- "A TR1 Tutorial: Regular Expressions," Marius Bancila, *CodeGuru.com*, 2 July 2008.
- C++11 ECMAScript regex, cpprocks.com.
  - ➡ One-page summary of regex syntax and usage API.





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

# 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.





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

# 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.

std::bind:

- "Arbitrary binding pt3 – type safe version," Crazy Eddie, *Crazy Eddie's Crazy C++* (blog), 31 December 2012.
  - Summarized how binding can be implemented.

Numeric ⇄ std::string Conversions:

- "String to int / double Conversions," John Ahlgren, *John Ahlgren* (Blog), 13 May 2012.

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

Slide 386

The article for std::bind is not easy reading, but I haven't seen anything else that discusses how std::bind could be implemented.

# Further Information

From TR1 to C++11:

- "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.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 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, September 2011, 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.
- "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.





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

# Further Information

Rvalue References and Move Semantics:

- "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.





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

# Further Information

Universal references:

- "Universal References in C++11," Scott Meyers, *Overload*, October 2012.
  - Written presentation (monochrome PDF).
- "Universal References in C++11," Scott Meyers, *isocpp.org*, 1 November 2012.
  - Web rendition of above (w/links and syntax-colored code).
- "C++ and Beyond 2012: Scott Meyers - Universal References in C++11," Scott Meyers, *Channel 9*, 9 October 2012.
  - Video of a live presentation.





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

# Further Information

Perfect Forwarding:

- "Onward, Forward!," Dave Abrahams, *C++Next*, 7 Dec. 2009.
- "forward," Howard E. Hinnant, *Document N2951 to the C++ Standardization Committee*, 27 September 2009.
  - Pros/cons of several potential std::forward implementations.
  - "Proposed wording for US 90," Howard Hinnant and Daniel Krügler, *Document N3143 to the C++ Standardization Committee*, 15 October 2010.
    - Complements N2951 with a new use case and offers final specification for std::forward.
- "Perfect Forwarding Failure Cases," comp.std.c++ discussion initiated 16 January 2010.
  - Discusses argument types that can't be perfectly forwarded.





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

# Further Information

Explicit conversion functions:

- "Use Explicit Conversion Functions to Avert Reckless Implicit Conversions," Danny Kalev, *DevX.com*, 9 October 2008.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Further Information

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 Are Fun—and Now Legal," Andrei Alexandrescu, *C++ and Beyond 2011*, 8 August 2011.
  - Basis for my code for std::tuple_element and std::get.
- "Size matters," Martinho Fernandes, *Flaming Dangerzone* (blog).
  - Describes TMP-computed minimal-size layout for std::tuple.
    - Part 1, 6 July 2012.
    - Part 2, 2 December 2012.
    - Part 3, 9 December 2012.





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

# Further Information

decltype:

- "A Note About decltype," Andrew Koenig, *Dr. Dobbs*, 27 July 2011.
- "Appearing and Disappearing consts in C++," Scott Meyers, *C++Next*, 1 April 2011.
  - ➡ Includes some discussion of decltype subtilties.
- "decltype: C++0x Features in VC10, Part 3," Stephan T. Lavavej, *Visual C++ Team Blog*, 22 April 2009.
- "C++ auto and decltype Explained," Thomas Becker, http://thbecker.net/articles/auto_and_decltype/section_01.html, May 2013.
- "Clean up Function Syntax Mess with decltype," Danny Kalev, *DevX.com*, 8 May 2008.





Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2013 Scott Meyers, all rights reserved.

# Further Information

Alignment Control:

- "Probe and Modify Your Types' Alignment Programmatically," Danny Kalev, *DevX.com*, 10 July 2008.
- "alignas / alignof," George Flanagin, 17 June 2011, http://www.georgeflanagin.com/c++11/alignas.php.
- "Boost.Variant," Eric Friedman and Itay Maman, *Boost.org*.

# Further Information

Other C++11 Features:

- "Compile-time computations," Andrzej Krzemieński, *Andrzej's C++ blog*, 6 May 2011.
  - Introduction to constexpr.
- "C++ constexpr," and "C++ constexpr foldr," Ulrik Sverdrup (?), http://kaizer.se/wiki/log/archive/, 21 October 2011.
  - Example uses of constexpr functions.
- "User-defined literals," Andrzej Krzemieński, *Andrzej's C++ blog*.
  - Part 1, 12 August 2012.
  - Part 2, 23 October 2012.
  - Part 3, 29 October 2012.
- "System error support in C++0x," Chris Kohlhoff, *Thinking Asynchronously in C++* (blog), five-part series, 7 April – 14 April 2010.





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

# Further Information

Other C++14 Features:

- "Binary Literals in the C++ Core Language," James Dennett, C++ Standardization Committee Document N3472, 19 October 2012.
- "C++ Dynamic Arrays," Lawrence Crowl and Matt Austern, C++ Standardization Committee Document N3662, 19 April 2013.
- "Runtime-sized arrays with automatic storage duration (revision 5)," Jens Maurer, C++ Standardization Committee Document N3639, 16 April 2013.
- "A proposal to add a utility class to represent optional objects (Revision 3)," Fernando Cacciola and Andrzej Krzemieński, C++ Standardization Committee Document N3527, 10 March 2013.
  - ➡ Succeeded by N3672, but that document lacks motivational information.





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

# Further Information

Other C++14 Features:

- "Variable Templates (Revision 1)," Gabriel Dos Reis, C++ Standardization Committee Document N3651, 19 April 2013.
- "User-defined Literals for Standard Library Types (part 1 - version 4)," Peter Sommerlad, C++ Standardization Committee Document N3642, 18 April 2013.
- "Quoted Strings Library Proposal (Revision 2)," Beman Dawes, C++ Standardization Committee Document N3654, 19 April 2013.





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

# 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++11 concurrency expertise and use of his just::thread C++11 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.





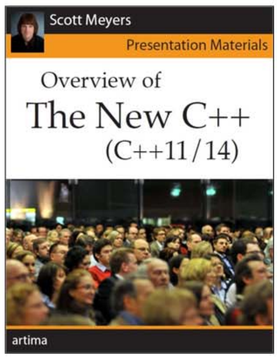

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

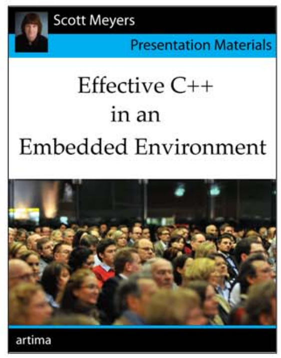
# Licensing Information

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:








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

# 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





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

# Contents



Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.