



Scott Meyers

Presentation Materials

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

A large, out-of-focus crowd of people, mostly men, are seated in rows, looking towards the left side of the frame. The background is dark with some blurred lights, suggesting an indoor event space.

Sample

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

Thank you for downloading this sample from the presentation materials for Scott Meyers' *Overview of the New C++ (C++11/14)* training course. If you'd like to purchase the complete copy of these notes, please visit:

http://www.artima.com/shop/overview_of_the_new_cpp

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

Copyright © 2010-2013 Scott Meyers. All rights reserved.

Cover photo by Stephan Jockel. Used with permission.

All information and materials in this document are provided “as is” and without warranty of any kind.

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

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

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

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

© 2013 Scott Meyers, all rights reserved.
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**

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

© 2013 Scott Meyers, all rights reserved.

Slide 2

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

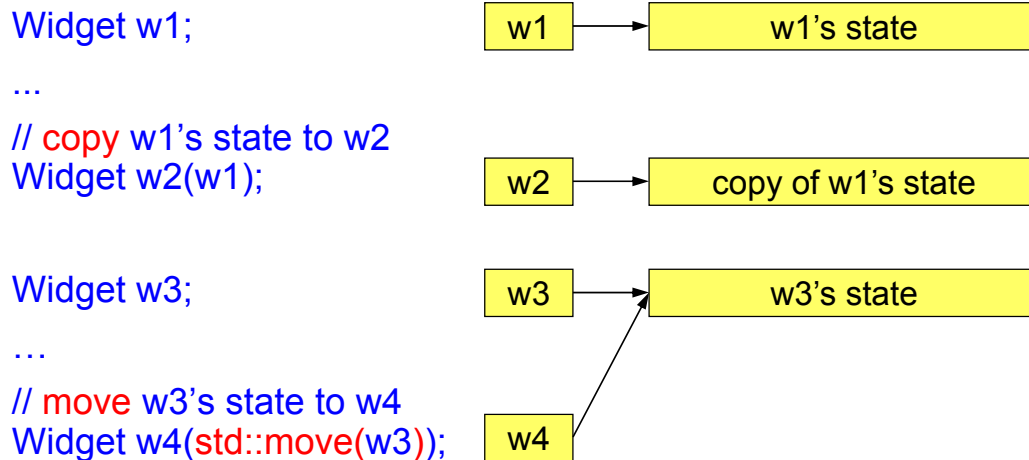
- 1998: ISO C++ Standard officially adopted (“C++98”).
 - 776 pages.
- 2003: TC1 (“Technical Corrigendum 1”) published (“C++03”).
 - Bug fixes for C++98.
- 2005: TR1 (Library “Technical Report 1”) published.
 - 14 likely new components for the standard library.
- 2009: Selected “C++0x” features became commonly available.
- 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).

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:



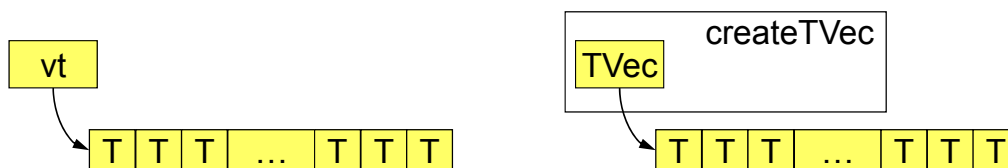
Note: `w3` continues to exist in a valid state after creation of `w4`.

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.

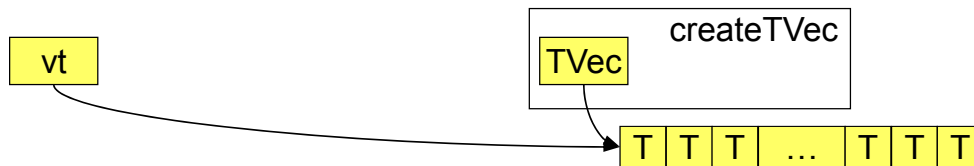
Copying vs. Moving

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.

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

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

© 2013 Scott Meyers, all rights reserved.

Slide 8

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

template<typename MapIt>
void showCommonWords(MapIt begin, MapIt end, const std::size_t n)
{
    // print n most
    // 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/>

© 2013 Scott Meyers, all rights reserved.

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));
}
```

`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;
}
```

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>
void showCommonWords(MapIt begin, MapIt end, const std::size_t n)
{
    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);
    }
}

```

// print n most
// common words
// in [begin, end)

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

© 2013 Scott Meyers, all rights reserved.

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));
}
```

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

© 2013 Scott Meyers, all rights reserved.

Slide 14

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);
    }
}

```

Comparison

<pre> 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)); } </pre>	<pre> 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)); } </pre>
--	---

Overview

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

“>>” 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>>> 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)
```

[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.

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
```

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 `const`/`volatile` 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]
```


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
```

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"
```

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_ptr`s 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.

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.

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
```

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!
```

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
```

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
```

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 `\unnnn` and `\Uxxxxxxxx`:

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

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 properly 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>
```

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.”

Conversions Among Encodings

C++98 supports only IO-based conversions.

- Designed for multibyte external strings \rightleftharpoons 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` \Rightarrow `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"xyzzzy("[A-Za-z_]w*")xyzzzy"); // "(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., `xyzzzy` 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).