Moving to Modern C++:
New Language Features Used Everywhere

Steve Dewhurst
Dan Saks

1

Modern C++

The C++ programming language is defined by a formal international standard specification. That standard was updated in 2011 and again in 2014. Modern C++ is the language as specified by these recent standards.

Compared to the earlier standards, Modern C++ introduces a significant number of new language and library features. This course focuses primarily on the language features of Modern C++ and programming techniques that use those features.

2

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-937-324-3601
dan@dansaks.com
www.dansaks.com

3

4

Legal Stuff

- These notes are Copyright © 2017 by Stephen C. Dewhurst and Daniel Saks.
- If you have attended this course, then:
  - You may make copies of these notes for your personal use, as well as backup copies as needed to protect against loss.
  - You must preserve the copyright notices in each copy of the notes that you make.
  - You must treat all copies of the notes — electronic and printed — as a single book. That is,
    - You may lend a copy to another person, as long as only one person at a time (including you) uses any of your copies.
    - You may transfer ownership of your copies to another person, as long as you destroy all copies you do not transfer.

5

More Legal Stuff

- If you have not attended this course, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the course, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

6

About Steve Dewhurst

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of the critically-acclaimed books *C++ Common Knowledge* and *C++ Gotchas*, and the co-author of *Programming in C++*. He has written numerous technical articles on C++ programming techniques and compiler design.

As a Member of Technical Staff at AT&T Bell Laboratories, Steve worked with C++ designer Bjarne Stroustrup on the first public release of the C++ language and cfront compiler. He was lead designer and implementer of AT&T's first non-cfront C++ compiler. As a compiler architect at Glockenspiel, Ltd., he designed and implemented a second C++ compiler. He has also written C, COBOL, and Pascal compilers.

Steve served on both the ANSI/ISO C++ standardization committee and the ANSI/IEEE Pascal standardization committee.

7

About Steve Dewhurst

Steve has consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading. He has been a frequent and highly-rated speaker at industry conferences such as *Software Development* and *Embedded Systems*. He was a Visiting Scientist at CERT and a Visiting Professor of Computer Science at Jackson State University.

Steve was a contributing editor for *The C/C++ User's Journal*, an editorial board member for *The C++ Report*, and a cofounder and editorial board member of *The C++ Journal*.

Steve received an A.B. in Mathematics and an Sc.B. in Computer Science from Brown University in 1980 and an M.S. in Engineering/ Computer Science from Princeton University in 1982.

8

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan is a contributing editor for *embedded.com* online. He has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught C and C++ to thousands of programmers around the world. He has presented at conferences such as *Software Development*, *Embedded Systems*, and *C++ World*. He has served on the advisory boards of the *Embedded Systems* and *Software Development* conferences.

9

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ standards committees and as a member of the ANSI C standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. Previously, he was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/Information Science from Case Western Reserve University.

10

Past C++ Standards

- ***1998***: “C++98”
  - the first international C++ standard (ISO [1998])
- ***2003***: “C++03”
  - a revised international C++ standard (ISO [2003])
  - bug fixes
  - nothing else new
- ***2005***: “TR1”
  - Library “Technical Report 1” (ISO [2005])
  - proposals for library extensions
  - not a new standard

11

Modern C++ Standards

- ***2011***: “C++11”
  - a new international C++ standard (ISO [2011a])
  - significant new language features
  - most of TR1, plus more library components
- ***2014***: “C++14”
  - the latest international C++ standard (ISO [2014])
  - mostly improvements to C++11 features
  - a few new features, too

12

New Language Features Used Everywhere

13

NULL

- Various C++ standard headers define NULL as a macro, typically as either:

```
#define NULL 0
#define NULL 0L
```

- Using NULL can lead to unintuitive behavior:

```
void f(int);
void f(char *);
~~~
f(NULL);        // calls f(int);
```

- Similar problems arise with template argument deduction…

14

Deduction-Induced Failure

```
template <typename F, typename T>
void invoke(F func, T arg) {
    func(arg);
}
~~~
void finalize(Widget *);
```

- Template argument deduction works just fine here:

```
Widget *wp = new Widget;
~~~
invoke(finalize, wp);       // OK
```


Deduction-Induced Failure

```
template <typename F, typename T>
void invoke(F func, T arg) {
    func(arg);
}
~~~
void finalize(Widget *);
```

- Template argument deduction doesn't work "as expected" here:

```
invoke(finalize, NULL);     // compile error!
```

- The compiler deduces `NULL`'s type to be `int`, not a pointer type.
- Inside the template instantiation, the call `finalize(NULL)` fails because there's no standard conversion from `int` to `widget *`.

nullptr

- C++11 provides a new keyword, `nullptr`.
- `nullptr` is a null pointer constant of type `std::nullptr_t`.
  - It can be converted implicitly or compared to any pointer or pointer-to-member type.
  - It can't be implicitly converted or compared to any integral type, except for `bool`.
- `0`, `0L`, and `NULL` remain valid null pointer constants.
  - They're no longer fashionable.

17

nullptr

- Using `nullptr` provides more intuitive behavior for overload resolution:

```
void f(int);
void f(char *);
~~~
f(NULL);        // still surprising: calls f(int);
f(nullptr);     // unsurprising: calls f(char *);
```

- Ditto for template argument deduction…

18

nullptr

```
template <typename F, typename T>
void invoke(F func, T arg) {
    func(arg);
}
~~~
void finalize(Widget *);
```

- Argument deduction with nullptr is less surprising:

```
invoke(finalize, NULL);     // compile error!
invoke(finalize, nullptr);  // OK...
```

- The compiler deduces nullptr's type to be std::nullptr_t, which converts to Widget *.

19

>> in Template Argument Lists

- A template type argument can be any type name, not just an identifier.
- For example,

```
list<string> names;
list<node *> trees;
```

- A template type argument can even be the name of a nested template specialization, as in:

```
list<rational<int> > ratios;
```

- Notice the space between the two > operators...

20

>> in Template Argument Lists

- C++03 always interprets >> as a single operator.
  - You can blame a guy named Max Munch.
- C++11 interprets >> occurring in a template type argument list as two separate > operators.

```
list<rational<int>> ratios; // OK in C++11; not in C++03
```

- A template can have non-type parameters, as in:

```
template <size_t N>
class bitset;
```

- The argument to this template must be a constant expression…

21

>> in Template Argument Lists

- For example,

```
bitset<sizeof(int)> b1;
```

- The argument expression can have operators.
- An expression with an > or >> operator must be enclosed in () so the compiler won't mistake the operator for a closing delimiter:

```
bitset<sizeof(int)>>1> b2;   // OK in C++03; not in C++11
bitset<(sizeof(int)>>1)> b3; // OK in C++03 and C++11
```

✓ *Parenthesize.*
- It's best to avoid complexity in template argument lists.

22

Using assert

- The `assert` macro is defined in the standard header `<cassert>`.
- Calling `assert(e)` expands to code that tests the value of expression `e` at run time:
  - If `e` is true (non-zero), nothing happens.
  - If `e` is false (zero), the program writes a diagnostic message to `stderr` and aborts execution by calling the standard `abort` function.

23

Using assert

- For example, you can use `assert` to verify the layout of a structure:

```
struct timer {
    uint8_t MODE;
    uint32_t DATA;
    uint32_t COUNT;
};
assert(offsetof(timer, DATA) == 4);
```

- `offsetof(t, m)` (defined in `<cstddef>`) returns the offset in bytes of member `m` from the start of structure type `t`.
- This `assert` call verifies that `DATA` has an offset of 4 within the `timer` structure.

24

`assert` is a Runtime Check

- This assertion does indeed catch the alignment problem:

```
assert(offsetof(timer, DATA) == 4);
```

- Unfortunately:
  - It defers until run time a check that should be done at compile time.
  - It can appear only within a function, so you have to wrap it inside a function and call that function to do the check.
- Not all assertions can be checked at compile time, but this one can…

25

`static_assert`

- C++11 supports compile-time assertions as a language (not a library) feature.
- A *static_assert-declaration* has the form:

```
static_assert(e, s);
```

- `e` must be a constant expression.
- `s` must be a string literal.
- If `e` converted to `bool` is true, the declaration has no effect.
- Otherwise, the compiler generates a diagnostic message containing `s`, and the program fails to compile.

26

static_assert

- A `static_assert` can appear anywhere a declaration can appear:

```
struct timer {
    uint8_t MODE;
    uint32_t DATA;
    uint32_t COUNT;
};
static_assert(
    offsetof(timer, DATA) == 4,
    "DATA must be at offset 4 within timer"
);
```

27

Replace Comments with Code

✓ *Don't express in comments what you can express in code.*
- The compiler will check code, but not comments.
- Comments like this are never necessary:

```
typedef unsigned special_register; // MUST BE 4 BYTES!!!
```

- Let the compiler check, and sleep better.

```
typedef unsigned special_register;
static_assert(sizeof(special_register) == 4,
    "special_register must be 4 bytes");
static_assert(alignof(special_register) == 4,
    "bad alignment for special_register");
```

28

Assist the Compiler With Error Messages

- Use of `static_assert` can make sense even in situations where the compiler will already produce an error.
- Unfortunately, compiler errors are often voluminous (particularly for errors involving templates) and hard to find in the source code.
- Recognizable output from a `static_assert` can be the needle in that haystack of error messages that will help you to find the actual error.

```
static_assert(0 <= d && d < base,
              "bad digit in user-defined literal");
```

29

Compiler Error Messages

- In amongst reams of compiler messages like…

```
'initializing': truncation of constant value

'd': enumerator value '-9' cannot be represented as
'unsigned int', value is '4294967287'
```

- …we'll find this gem…

```
bad digit in user-defined literal
```

- …and realize that we have to upgrade our user-defined literal implementation to C++14, and accept apostrophes in numeric literals.

30

Primary Template Declaration Issues

- It's common to declare a primary template, but implement only partial specializations and specializations.

```
template <typename Ptr> // primary declaration
class MungePtr;

template <typename T>   // specialization
class MungePtr<T *> {
    ~~~
};
```

- Unfortunately, compiler error messages resulting from specialization of the primary can be…idiosyncratic.

31

Primary Errors

- A static assertion can help to clarify:

```
template <typename Ptr>
class MungePtr {
    static_assert(false,
                  "improper non-ptr arg to MungePtr");
};
```

- Unfortunately, this code will never compile.
- Because there are no dependent names in the assertion, the `static_assert` is processed in the first stage of template translation, whether or not the primary template is actually specialized.

32

Primary Solution

- One solution is to introduce a dependent name in an expression that will always be false:

```
template <typename Ptr>
class MungePtr {
    static_assert(sizeof(Ptr) == 0,
                  "improper non-ptr arg to MungePtr");

};
```

- The dependent name Ptr prevents the assertion from being translated unless the primary template is selected in a specialization.

33

Type Traits

- The <type_traits> header is largely a collection of traits classes.
- Each traits class provides a single piece of information about an aspect of a type.
- The header contains a number of class templates that can extract information about types.
- Some are rather pedestrian:
  - is_arithmetic
  - is_member_function_pointer
  - is_pointer
  - is_unsigned

34

Non-Basic Type Traits

- Some you'd prefer the compiler to handle:
  - `is_convertible`
  - `is_polymorphic`
- Others you'd be hard pressed to write without help from the compiler:
  - `has_trivial_destructor`
  - `has_virtual_destructor`
  - `is_pod`
  - `is_nothrow_copy_constructible`

35

Poor Man's Concepts

- You can use type traits along with `static_assert` to check implicit requirements in template code.
- For example:

```
template <typename T>
void munge_shape(T *s) {
    static_assert(
        is_base_of<Shape, T>::value,
        "argument must be a pointer to Shape"
    );
    ~~~
}
```

36

Poor Man's Concepts

- As another example:

```
template <typename T>
T *copy_it(T const *src, size_t n) {
    static_assert(
        is_trivially_copyable<T>::value,
        "array type must be memcopy-able"
    );
    ~~~
}
```

- Strictly speaking, this use of `static_assert` is what Stroustrup [2013] calls "constraint checking" rather than concept checking.

37

Example: Implementing `is_void`

- Most of these traits could be implemented to provide a compile-time boolean constant as the query result:

```
template <typename T>
struct is_void {            // T isn't void...
    enum: bool { value = false };
};

template <>
struct is_void<void> {      // ...unless T is void.
    enum: bool { value = true };
};
```

38

The Standard Library Approach

- The C++ Standard Library uses a different approach.
- The standard type traits derive from one of two base classes that represent “true” and “false.”
- This is a common and convenient way to “import” several related pieces of information about a type or value…

39

Standard Implementation of is_void

- The implementation might look like this:

```
template <typename T>
struct is_void: public false_type { };

template <>
struct is_void<void>: public true_type { };

template <>
struct is_void<void const>: public true_type { };
```

- …and so on for `void volatile` and `void const volatile`.

40

A Helper Template

- C++11 actually defines `false_type` and `true_type` in terms of a general "helper" template for integral constants.
- `integral_constant<T, v>` encodes compile-time value `v` with its type `T` and the type of its wrapper:

```
template <typename T, T v>
struct integral_constant {
    typedef integral_constant<T, v> type;
    static constexpr T value = v;
    typedef T value_type;
    ~~~
};
```

41

Compile Time True and False

- `true_type` and `false_type` are typedef aliases for specializations of `integral_constant`:

```
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

- `is_void` inherits members `type`, `value`, and `value_type` from its public base class, which is either `true_type` or `false_type`.

42

Accessing Static Type Information

- You can use a type trait by directly accessing a member inherited from `true_type` or `false_type`:

```
is_signed<SomeType>::value  // is SomeType signed?
```

- Note that the nested name `value` is a compile-time constant.
- Alternatively…

43

Using An Implicit Conversion

- You can use overload resolution along with a derived-to-base conversion…

```
template <typename T>        // use when T has a virtual
void impl_func(T const &arg, true_type const &);
template <typename T>        // use when T doesn't
void impl_func(T const &arg, false_type const &);
~~~
template <typename T>
void interface_func(T const &an_arg) {
    impl_func(an_arg, is_polymorphic<T>());
}
```

derived-to-base conversion to either `true_type` or `false_type`

44

And The Point Is…

- The depth of information available from these compile-time predicates can be used to perform fine-grained static assertions, such as…
- "Confirm that type A is convertible to type B, and that B has a nothrow copy constructor."

```
static_assert(
    is_convertible<A, B>::value
        && is_nothrow_copy_constructible<B>::value,
    u8"my idiosyncratic needs are not met\u2049"
); // !?
```

45

Stroustrup's Syntactic Simplification

- Stroustrup [2013] suggests using constexpr "helper" functions:

```
template <typename T>
constexpr bool is_signed_v() noexcept {
    return is_signed<T>::value;
}
```

- The helper function lets you write slightly simpler expressions to access traits, as in:

```
is_signed<X>::value      // without the helper
is_signed_v<X>()         // with the helper
```

- This simplification is not part of the C++ Standard Library.

46

Using Variable Templates

- C++14 introduces variable templates, which permit a further simplification of Stroustrup's mechanism:

```
template <typename T>
constexpr auto is_signed_v = is_signed<T>::value;
~~~
is_signed_v<X> // as opposed to is_signed_v<X>()
               // or is_signed<X>::value
```

- These syntactic simplifications are available in the standard library starting with C++17.

47

Code Elimination and `#if`

✓ *Prefer `if` over `#if` to eliminate code.*

- Mixing compile-time and runtime control flow is particularly damaging to readability.
- If code is hard to understand, it's hard to maintain.
- You effectively wind up with several different programs generated from the same source.

48

Code Elimination and #if

- For example, here we have two different programs — a debug version and a release version:

```
void buggy()  {
    #ifndef NDEBUG
    // ~~~ debugging code ~~~
    #endif
    ~~~
}
```

- Real programs often have hundreds of `#if` versions.

49

Let the Compiler Compile

- It's typically better to let the compiler see *all* the source code:

```
void buggy() {
    if (debug) {
        // debugging code…
    }
    ~~~
}
```

- Now the compiler can parse and statically check the debug code.

50

Code Elimination

- If the condition is a constant expression, then most compilers will eliminate unreachable debugging code:

```
bool const debug = false;
~~~
if (debug) {                    // a constant expression
    ~~~
}
int const debugLevel = 3;
~~~
if (debug && debugLevel > 3) {  // ditto
    ~~~
}
```

51

More Flexible Elimination

- Note that the preprocessor doesn't have access to compiler-provided static information (such as `type_traits`).
- The (post-preprocessor) compiler can handle this condition:

```
if (debug && debugLevel > 3
    && !has_trivial_assign<X>::value) {
    ~~~
}
```

- The preprocessor can't.
- Thus, the compiler is a more precise and extensive tool for code elimination.

52

Unsafe Optimizations

- "I know that all Ts are simple structures, so I can use memcpy to copy them."

```
template <typename T>
inline T *copy_array(T const *s, size_t n)  {
    size_t const amt = sizeof(T) * n;
    return static_cast<T *>(
        memcpy(::operator new(amt), s, amt)
    );
}
```

- That statement might be true initially, but may become false in the future.
- The compiler won't be able to detect the error in this code.

53

Safe Assumptions

- It's safer make our assumption explicit:

```
template <typename T>
inline T *copy_array(T const *s, size_t n)  {
    static_assert(
        is_pod<T>::value,
        "argument must be an array of PODs"
    );
    size_t const amt = sizeof(T) * n;
    return static_cast<T *>(
        memcpy(::operator new(amt), s, amt)
    );
}
```

54

Safer, Self-Maintaining Optimization

- In general,

✓ *Check assumptions statically, and then take appropriate action.*

- If the condition being checked can be represented as a constant-expression, the compiler will typically remove the unused code.
- This use of constant folding and code elimination is one of many similar techniques for static optimization...

55

- "If T is a plain ol' struct equivalent, use memcpy."
- "Otherwise, initialize each element with a constructor call."

```
template <typename T>
inline T *copy_array(T const *s, size_t n) {
    size_t const amt = sizeof(T) * n;
    T *d = static_cast<T *>(::operator new(amt));
    if (is_pod<T>::value) { // a constant-expression
        d = static_cast<T *>(memcpy(d, s, amt));
    } else {                // not perfectly correct!
        for (size_t i = 0; i != n; ++i) {
            new (&d[i]) T (s[i]);
        }
    }
    return d;
}
```

56

Safer, Self-Maintaining Optimization

- In the previous example, we should also be concerned with a third case:
  - a copy constructor that might throw an exception.
- We could detect and handle that case with another use of type traits.

✓ *Note that the compiler will perform code elimination only after all the code in the function has been parsed, instantiated, and statically checked for correctness.*

- This is usually a good thing.

57

Two-Phase Translation

- The implementation of `copy_array` contained two separate algorithms.
- Because `copy_array` is a template, it undergoes two-phase translation.
- In the first phase the template is parsed and any non-dependent names are bound.
- In the second phase the template is specialized with a specific type `T`, and the compiler completes the translation.
- All of the code in the template must compile, even if the compiler later removes code that is unreachable.

58

Reversing a Sequence

- Here's another traditional problem: What's the best way to reverse a sequence described by two iterators?
- If the iterators are random access, we can do this:

```
template <typename Ran>
void reverse(Ran b, Ran e) {
    for (; b < e; ++b)
        iter_swap(b, --e);
}
```

59

Reversing a Bidirectional Sequence

- If all we have available is a bidirectional sequence, we can use a marginally slower algorithm:

```
template <typename Bi>
void reverse(Bi b, Bi e) {
    for (; b != e && b != --e; ++b)
        iter_swap(b, e);
}
```

- This is not the type of decision we would typically leave up to users.
- We'd prefer to have a self-maintaining automatic selection of the correct algorithm.

60

Algorithm Selection

- Here's a first attempt to package both algorithms together:

```
template <typename Bi>
void reverse(Bi b, Bi e) {
    using category
    = typename iterator_traits<Bi>::iterator_category;
    if (is_same<category,
        random_access_iterator_tag>::value)
        for (; b < e; ++b)
            iter_swap(b, --e);
    else // bidirectional or lesser...
        for (; b != e && b != --e; ++b)
            iter_swap(b, e);
}
```

61

Packaging Problems

- This approach works fine for random access iterators…

```
vector<int> v{ 1,2,3,4,5,6, };
reverse(v.begin(), v.end());
```

- ...but fails for purely bidirectional iterators.

```
list<int> lst(v.begin(), v.end());
reverse(lst.begin(), lst.end()); // error!
```

- In the second phase of translation, the list's bidirectional iterator is asked to do something it can't.

```
for (; b < e; ++b) // error! no < for bidirectional
```

62

Repackaging for Two-Phase Translation

- The traditional solution is to repackage the code that causes the conflict so that it is not subject to the second phase of translation.

```
template <typename Bi> void _reverse(Bi b, Bi e,
              bidirectional_iterator_tag) { ~~~ }

template <typename Ran> void _reverse(Ran b, Ran e,
              random_access_iterator_tag) { ~~~ }

template <typename Bi>
inline void reverse(Bi b, Bi e) {
    typedef typename
      iterator_traits<Bi>::iterator_category category;
    _reverse(b, e, category());
}
```

63

Constexpr If

- Repackaging incompatible code is a proven solution to the problem, but is complex and doesn't scale well.
- C++17 introduces *constexpr if* to simplify this common situation.

```
if constexpr(condition) {
    // part 1
}
else {
    // part 2
}
```

- The condition in a constexpr if must be a compile-time Boolean constant.

64

Discarding Statements

- If the condition in a constexpr if is false, its statement is "discarded." It is not subject to the second phase of template translation.

```
if constexpr(sizeof(char) == 0) {
    ~~~ // discarded
}
else {
    ~~~ // instantiated
}
```

- Only non-discarded code is instantiated.

65

```
template <typename Bi>
void reverse(Bi b, Bi e) {
  using category = typename
      iterator_traits<Bi>::iterator_category;
  if constexpr(std::is_same<category,
                 random_access_iterator_tag>::value)
      for (; b < e; ++b) iter_swap(b, --e);
  else if constexpr(std::is_same<category,
                 bidirectional_iterator_tag>::value)
      for (; b != e && b != --e; ++b) iter_swap(b, e);
  else           // lesser iterator category...
      static_assert(is_void<Bi>::value,
      "bad iterator category for reverse algorithm");
}
```

66

Typedef

- A typedef is a handy way to introduce a new name for a type without actually creating a new type.
- For example:

```
typedef std::map<key, mapped_type> Map;
typedef Map::iterator I;
Map aMap;
~~~
for (I i = aMap.begin(); i != aMap.end(); ++i) {
    ~~~
}
```

67

Typedef

- It's common practice to use typedefs to simplify complex declarations.
- For example, here's a declaration for a function named `set_callback` that accepts and returns a pointer to function:

```
void (*set_callback(void (*)()))(); // ???
```

- Using a typedef makes it easier to read:

```
typedef void (*FP)();
FP set_callback(FP);                // Oh...
```

68

Alias Declarations

- In C++11, you can use an alias-declaration as an alternate syntax for defining a typedef name.
- An ***alias-declaration*** has the form:

  `using` *identifier* = *type-id*;

- For example, you can declare `FP` as a typedef using either:

```
typedef void (*FP)();       // traditional typedef
using FP = void (*)();      // alias-declaration
```

- The effect is the same either way.

69

Alias Templates

- The real advantage is that an alias-declaration may be a template.
- A typedef cannot.
- You can use a parameterized alias-declaration:
  - to partially-specialize a template, or
  - to simplify complex usage of a template.
- For example,

```
template <typename T>
using Vector = vector<T, MyAlloc<T>>;
~~~
Vector<string> vs;  // vector<string, MyAlloc<string>>
Vector<int> vi;     // vector<int, MyAlloc<int>>
```

70

Inflexible Policies

- Consider a stack implementation that uses policy-based design (PBD) to allow the user to select the implementation policy:

```
template <typename T, template <typename> class C>
class Stack {
public:
    void pop() { cont_.pop_back(); }
    ~~~
private:
    C<T> cont_;
};
```

- `Stack<T, C>` expects template parameter `C` to be a container that can itself be specialized with a single type argument…

71

Unusable Standard Containers

- Unfortunately, all the viable standard containers (`deque`, `list`, and `vector`) have two type parameters:

```
Stack<string, list> names;      // compile error!
```

- But, you say, `list` *can* be specialized with one type argument:

```
list<string> roster;            // one type argument
```

- Indeed you can, because it has default argument(s):

```
template <class T, class A = allocator<T>> class list;
```

- However…

72

Alias Templates

- Template specialization doesn't consider default arguments when substituting arguments for template parameters:

```
Stack<string, list> names;  // can't use list<T, A> here
```

- Fortunately, you can employ an alias template to adapt the standard containers to our needs:

```
template <typename T>
using List = list<T, allocator<T>>;
~~~
Stack<string, List> names; // works...
```

73

Partially-Specialized Standard Containers

- All standard sequence container templates have default arguments for all template parameters after the first one.
- You can use an alias template for partial specialization:

```
template <typename T>
using Deque = deque<T>; // trailing arguments defaulted
~~~
template <
    typename T, template <typename> class C = Deque
> class Stack {
    ~~~
private:
    C<T> cont_;
};
```

74

Going the Other Way

- What if you have a non-template container you'd like to use?
- Now you have to *add* a template argument rather than get rid of one or more.
- No problem:

```
struct Cont { ~~~ };        // a container of strings
template <typename>
using Templatized = Cont;   // just throws away argument

Stack<string, Templatized> names; // done.
```

75

Syntactic Difficulties

- Older template metaprogramming features of the standard library can be syntactically challenging:

```
is_same<    // is this a random access STL iterator?
    typename iterator_traits<Iter>::iterator_category,
    random_access_iterator_tag
>::value
```

- The expression uses long identifiers.
- It also requires explicitly use of the keyword `typename` to identify the nested name `iterator_category` as a type.
- A "template typedef" alias can simplify the syntax…

76

Simplifying With “Template Typedef”

- For example, these alias templates can categorize iterators:

```
template <typename It>
using Category
    = typename iterator_traits<It>::iterator_category;

template <typename It>
using IsExactlyRand
    = is_same<Category<It>, random_access_iterator_tag>;

template <typename It>
using IsExactlyBi
    = is_same<Category<It>, bidirectional_iterator_tag>;
~~~
```

77

Simplifying With Alias Declarations

- This alias template can determine if an iterator is bidirectional:

```
template <typename It>
using IsBi = is_true<
    IsExactlyRand<It>::value || IsExactlyBi<It>::value
>;
```

- The `is_true` template is non-standard.
- You can define it as:

```
template <bool c>
struct is_true: std::integral_constant<bool, c>::type {
};
```

78

A Using Idiom

- Using newer parts of the C++ Standard Library, notably `<type_traits>`, can be syntactically challenging.
- For example, suppose you want to strip the reference modifier from a template type parameter `T`:

```
typename remove_reference<T>::type
```

- In the definition of a template with type parameter `T`, `remove_reference<T>::type` is a ***dependent name***.
- The compiler assumes that dependent names ***do not*** name types.
- Therefore, you must use the keyword `typename` to tell the compiler that the nested name `type` is a type name.
- Annoying.

79

Syntactic Improvement

- An alias template can improve the syntax somewhat:

```
template <typename T>
using remove_reference_t
      = typename remove_reference<T>::type;
~~~
remove_reference_t<T>
```

- The C++14 Standard Library provides alias declarations for type transformations like this.
- It doesn't provide alias templates for type queries such as `is_polymorphic`.
  - Accessing a type query's nested type name is rarely necessary.

80

Order of Using and Specializations

- Note that a templated using that precedes a template specialization may be used in the definition of a specialization.

```
template <typename T> class Templ;    // primary

template <typename T>                 // using
using Templ_t = typename Templ<T>::type;

template <typename T>                 // specialization
class Templ<const T *> {
    using type = Templ_t<T> &;
}
~~~
```

81

SFINAE

- "***S***ubstitution ***F***ailure ***I***s ***N***ot ***A***n ***E***rror" in template argument deduction.
- That is, if argument deduction finds at least one match, the failed matches aren't errors, as in:

```
template <typename T> void f(T);
template <typename T> void f(T *);
~~~
f(1024);        // no error, specializes first f
```

- The call `f(1024)` can match `f(T)`, but not `f(T *)`.
- The failure to match `f(T *)` is not an error.
- If `f(T)` were not present, it would be an error.
- The term SFINAE was introduced in Vandevoorde [2003].

82

Department of Redundancy Department?

- Consider two function templates with identical interfaces, but different requirements:

```
template <typename T>
void munge(T const &u) { ~~~ }  // for unions only!

template <typename T>
void munge(T const &c) { ~~~ }  // for classes only!
```

- There are two problems:
  - The second template is an invalid redefinition.
  - Even if the second definition were valid, calling `munge` would be ambiguous.
- Use of `enable_if` solves these problems…

83

enable_if

- A typical use of `enable_if` looks like:

```
typename enable_if<cond>::type
```

- Here, *cond* is a constant-expression that yields (something convertible to) a `bool`.
- If *cond* is true, then:
  - `enable_if<`*cond*`>` has a member named `type`, and
  - `enable_if<`*cond*`>::type` refers to that type.
- If *cond* is false, then:
  - `enable_if<`*cond*`>` doesn't have a member named `type`, and
  - `enable_if<`*cond*`>::type` is an invalid type name.
  - …and that's a substitution failure.

84

Fail, That Others May Succeed!

- Basically, enable_if allows a template to "fall on its sword" or (less violently) "step aside" from consideration.
- The munge for unions will have a valid return type only if T is a union:

```
template <typename T>
typename enable_if<is_union<T>::value>::type
munge(T const &u);
```

- Similarly for munge for classes:

```
template <typename T>
typename enable_if<is_class<T>::value>::type
munge(T const &c);
```

85

Fine-Grain Selection

- Now we can use SFINAE for fine-grain function selection.

```
union U { } u;
class C { } c;
munge(u);   // OK: class version substitution fails
munge(c);   // OK: union version substitution fails
munge(12);  // substitution failure error: both fail
```

- If the condition supplied to enable_if is satisfied, there's a return type available, and substitution succeeds.
- If the condition supplied to enable_if isn't satisfied, there's no return type available, and substitution fails.
- But SFINAE if there's a correct substitution.

86

enable_if

- Here's an implementation of enable_if:

```
template <bool cond, typename ReturnType = void>
struct enable_if {
    typedef ReturnType type;
};

template <typename ReturnType>
struct enable_if<false, ReturnType> {
    // no member named "type"
};
```

- enable_if<true> has a nested type named type.
- enable_if<false> doesn't.

87

enable_if For Selection

- This facility allows us to effectively overload function templates based on arbitrary properties of types:

```
template <typename T>
typename enable_if<is_signed<T>::value>::type
g(T const &a) { ~~~ }

template <typename T>
typename enable_if<is_unsigned<T>::value>::type
g(T const &a) { ~~~ }
```

- SFINAE assures you can call g as long as one specialization has a valid return type.

88

SFINAE "Preprocesses" Overloading

- In the previous example we were not overloading on the signedness of the argument.
- We instead used `enable_if` to first eliminate some candidate function templates prior to overload resolution.
- In the case of the overloaded function template g, there would be at most one candidate for overload resolution.
- Use of SFINAE essentially provides two-phase overload resolution.

89

SFINAE is not Overload Resolution

- Consider:

```
template <typename T> // #1
typename enable_if<is_signed<T>::value>::type
func(T t);
template <typename T> // #2
typename enable_if<is_integral<T>::value>::type
func(T t);
~~~
func(12);   // ambiguous overload, #1 and #2 considered
func(12.3); // OK, only #1 considered
```

- SFINAE is applied prior to overload resolution, but has the effect of allowing us to overload on arbitrary properties of a type.

90

“Overloading” on Argument Size

```
template <typename T>
typename enable_if<(sizeof(T)<4)>::type f(T const &a) {
    // do something if T is small
}

template <typename T>
typename enable_if<(sizeof(T)==4)>::type f(T const &a) {
    // do something different if T is mid-sized
}

template <typename T>
typename enable_if<(sizeof(T)>4)>::type f(T const &a) {
    // do something completely different if T is big
}
```

91

“Overloading” on PODness

```
template <typename T>
typename enable_if<is_pod<T>::value>::type
f(T const &a) {
    // do something if T is a POD
}

template <typename T>
typename enable_if<!is_pod<T>::value>::type
f(T const &a) {
    // do something different if T is not
}
```

92

enable_if For Adding Constraints

- enable_if may also be used to add additional constraints to a successfully-matched function template.
- Here's a function template that doesn't want to hear from anything that's not a Shape:

```
template <typename T>
typename enable_if<is_base_of<Shape, T>::value>::type
munge_shape(T const &a) {
    ~~~
}
```

- Admittedly, the syntax of the return type is…off-putting.

93

Default Function Template Arguments

- In C++11, function templates may have default template arguments.
- This allows a slight syntactic improvement:

```
template <
    typename T,
    typename = typename
        enable_if<is_base_of<Shape, T>::value>::type
>
void munge_shape(T const &a) { ~~~ }
```

- Now substitution will fail if it can't determine the type of the default template parameter.
- There are many ways to fail…

94

Using to the Rescue

- In syntactic situations like this, use of using is of use:

```
template <typename T>
using IsShape = typename
    enable_if<is_base_of<Shape, T>::value>::type;
```

- Our snobby function template is now fairly readable:

```
template <typename T, typename = IsShape<T>>
void munge_shape(T const &a);
```

- "... we never failed to fail; it was the easiest thing to do."

95

Constraints on Class Templates

- The same technique may be used to constrain class template specialization:

```
template <typename T>
using Empty
    = typename enable_if<is_empty<T>::value>::type;

template <typename T, typename = Empty<T>>
class ReturnForDeposit {    // empties only, please...
    ~~~
};
```

96

auto as a Type Specifier

- Consider the following map:

```
map<string, list<unsigned>> m;
```

- A loop that visits the pairs in the map typically looks like:

```
for (map<string, list<unsigned>>::iterator i = m.begin();
i != m.end(); ++i) {
    // do something with *i
}
```

- The for-statement initializer is *very* wordy.
- So much so that the for-statement doesn't fit on one line.
- This happens in real code, not just in PowerPoint slides.

97

auto as a Type Specifier

- In C++11, auto is a type specifier rather than a storage class specifier.
- For objects declared auto, the compiler deduces the object's type from its initializer's type:

```
for (auto i = m.begin(); i != m.end(); ++i)
    // do something with *i
```

- Here, i's initializer is m.begin().
- The compiler deduces m's type to be the initializer's type, namely:

```
map<string, list<unsigned>>::iterator
```

98

auto is a Placeholder

- auto is a placeholder for a type to be determined later, either explicitly or, more typically, by deduction.
- There are three typical reasons for wanting to delay:
  - Necessity, if there's not yet enough information to specify the type.
  - Convenience: if the type declaration is syntactically complex.
  - Safety and maintainability: if the type being declared changes frequently.
- Note:
  - auto doesn't involve runtime typing.
  - It delays determination of the type until later in compile time.

99

auto as a Type Specifier

- You can use auto in otherwise ordinary object declarations:

```
auto n = 10;        // same as int n = 10
auto x = 1.2;       // same as double x = 1.2
auto s = "hello";   // same as char const *s = "hello"
```

- In auto declarations, the compiler deduces the object's type using the template argument deduction rules:

```
template <typename T>
void f(T x);
~~~
f(v);               // deduce x's type from v
auto x = v;         // does essentially the same thing
```

100

auto as a Type Specifier

- You can combine auto with const and/or volatile:

```
auto const max = 10;    // max is "int const"
```

- auto also gets along fine with constexpr:

```
constexpr auto retsize = sizeof(f());
constexpr auto tentothe6 = pow(6);
```

101

auto as a Type Specifier

- An auto declaration may use declarator operators:

```
auto n = 10;
auto const *p = &n;     // p is "int const *"

auto volatile &port
    = *reinterpret_cast<serial_port *>(0x3FFD000);
                    // port is "serial_port volatile &"
```

102

auto as a Type Specifier

- In an auto declaration without an explicit reference type:
  - an initializer of an array or function type decays to a pointer
  - top-level const and volatile qualifiers disappear...
  - ...just as with template argument deduction.

```
int const m = 10;        // top-level const
auto n = m;              // n is int, but not const
++n;                     // OK: n is non-const
int const a[] = {        // const isn't top-level
    8, 6, 7, 5, 3, 0, 9
};
auto p = a;              // p is "int const *"
*p = 2;                  // error: *p is const
```

103

auto as a Type Specifier

- A declaration with the auto specifier may declare more than one object:

```
auto columns = 1920, rows = 1080;
```

- However, every object must have an initializer:

```
auto i = 0, j;           // error: j uninitialized
```

- The type deduced for each initializer must be the same:

```
auto x = 1, y = 1.0;     // error: i is int, y is double
```

104

auto as a Type Specifier

- Generally, we prefer to declare a single object per statement.
- For example, the declaration

```
auto columns = 1920, rows = 1080;
```

  is probably better rendered as two declaration statements:

```
auto columns = 1920;
auto rows = 1080;
```

- The single common exception to this rule is for "hoisting" a calculation out of a loop:

```
for (auto i = m.begin(), e = m.end(); i != e; ++i)
```

105

auto and Proxies

- The return type of `vector<bool>::front` is likely to be a *proxy* — a struct containing:
  - a pointer to a word containing the bit of interest and
  - an offset or mask for that bit.
- When an initializing expression yields a proxy, `auto` deduces a surprising, but correct, result.

```
vector<bool> vb;
~~~
auto top = vb.front();      // top is a proxy type
bool front = vb.front();    // front is a bool
```

- Object `top` isn't `bool`, but rather the deduced proxy type (with a conversion to `bool`).

106

The Explicitly-Typed Initializer Idiom

- Meyers [2015] recommends using a `static_cast` and `auto` in these circumstances:

```
auto top = static_cast<bool>(vb.front());
```

- Meyers calls this the "Explicitly-Typed Initializer" idiom.
- Sutter [2013] provisionally recommends the following equivalent, but declines to name the practice:

```
auto top = bool{ vb.front() };
```

- In either case, using `auto` forces an initializer to be present.

107

Let's Not Be Smug

- Do you know the smug feeling of superiority we get when we see a summer intern write:

```
vector<Widget> v;
~~~
Widget *wp = v.begin();   // well, it used to work!
```

- Well, let's not be so smug.
- How many of us write the following?

```
size_t count = v.size(); // well, it's always worked!
```

- This will probably work, but there's no guarantee.

108

Irrational Confidence

- The actual return type of `vector::size` is (probably) indirectly determined by the allocator used to specialize the `vector`.
- We should have written:

```
vector<Widget>::size_type count = v.size(); // size_t?
```

- By the same token, the summer intern should have written:

```
vector<Widget>::iterator wp = v.begin();    // Widget *?
```

- This is better…

109

Maintenance…

- …but it's not good enough.
- What happens under maintenance?

```
vector<Widget, myWeirdAllocator> v;        // oops...
~~~
vector<Widget>::size_type count = v.size();
```

- Is the `size_type` for a `vector<Widget>` appropriate for the return type of `vector<Widget, myWeirdAllocator>::size`?
- Possibly…

110

Prefer auto as a Type Specifier

- In general…
- ✓ *Prefer auto over other type specifiers.*
- It leads to code that's clear, correct, and self-maintaining.
- What's not to like?

```
vector<Widget, myWeirdAllocator> v;
~~~
auto wp = v.begin();    // self-maintaining...
auto count = v.size();  // ...and readable to boot
```

- Using auto also guarantees initialization:

```
size_t count;  // maybe a warning...
auto count2;   // error!
```

111

Tracking vs. Sticking

- Sutter [2014], as usual, has the *bon mot*:
- ✓ *"To make type **track**, deduce."*
- ✓ *"To make type **stick**, commit."*
- In our earlier examples, we wanted the size_type of our container to ***track***:

```
auto count = v.size();  // track
```

- We wanted the return type from a vector<bool> to ***stick***:

```
auto top = static_cast<bool>(vb.front());  // stick
bool top = vb.front(); // sticky, but less desirable
```

112

The decltype Specifier

- decltype is a compile-time "type of" operator for use as a type specifier.
- For example,

```
int i;
decltype(i) j;      // j has type "int"

int const c = i;
decltype(c) k;      // error: k is "int const" ...
                    // ... but with a missing initializer
```

- Caution: decltype(*(e)*) can be different from decltype(*e*).
- Here's a simplified explanation…

113

The decltype Specifier

- For decltype(*e*):
  - If e is the qualified or unqualified name of an object at block or namespace scope, or the name of a function parameter, then decltype(e) is the declared type of the named entity.
  - If e is a class member access such as x.m or p->m, then decltype(e) is the declared type of m.
    - This applies if e is just m, where m is equivalent to this->m.
  - Otherwise, decltype(e) is the static type of e.
- For decltype(*(e)*):
  - If e is an lvalue (refers to an object), then decltype((e)) is "T &", where T is the type of e.
  - Otherwise, decltype((e)) is decltype(e).
- For example…

114

The decltype Specifier

```
struct S {
    S(): n (42) { }
    int m;
    int const n;
    void f() const {        // this is "S const *"
        decltype(m) x = n;  // this->m is "int const"
        ~~~                 // but decltype(m) is int
    }
};
~~~
S x;
decltype(S::m) i = 0;       // i is int
S const *p = &x;
decltype(p->m) j = i;       // p->m is "int const"
++j;                        // OK: j is int
```

115

The decltype Specifier

```
struct S {
    S(): n (42) { }
    int m;
    int const n;
    ~~~
};

S x;
decltype(x.m) i = 0;     // i is int
decltype((x.m)) k = i;   // k in "int &"
++k;                     // increments i
int const m = i;
decltype(m) n;           // error: n is "int const" ...
                         // missing initializer
```

116

The decltype Specifier

- decltype is another "unevaluated context."
- That is, decltype(e) doesn't actually evaluate e.
- For example, decltype(abs(x)) doesn't generate code to call abs(x).
  - It yields the return type of abs(x) at compile time.
- This is similar to the behavior of sizeof:
  - sizeof(abs(x)) doesn't generate code to call abs(x).
  - It simply yields the size (in bytes) of the return type of abs(x).

117

decltype vs. auto

- You can use decltype instead of auto.
- That is, these are equivalent:

```
auto i = c.begin();
decltype(c.begin()) i = c.begin();
```

- The latter is wordier, to no advantage.
- Rather, decltype serves other purposes, such as…

118

Trailing Return Type

- Consider the following function template:

```
template <typename T, typename U>
R difference(T t, U u) {
    return t - u;
}
```

- Here, R is the function return type.
- R might vary depending on T and U.
- C++03 has no way to deal with this.
- C++11 solves this problem using decltype in combination with a trailing return type…

119

Trailing Return Type

- The keyword auto indicates that the function return type appears after the parameter list and an ->, as in:

```
template <typename T, typename U>
auto difference(T t, U u) -> decltype(t - u) {
    return t - u;
}
```

- In this case, the return type is the decltype of the result of t - u, for whatever types T and U are.
- auto acts as a placeholder for a return type later determined from types appearing in the parameter list.

120

Complex Leading Return Types

- You can use decltype in a conventional (leading) return type.
- However, using a leading return type specification is typically more complicated:

```
template <typename T, typename U>
decltype(declval<T>() - declval<U>())
difference(T t, U u) {
    return t - u;
}
```

- The standard function template declval<T>() returns an operand of type T.
- It's only for use in expressions within unevaluated contexts, such as decltype and sizeof.

121

Trailing Returns for Members

- A trailing return type for a class member is in the scope of the class:

```
class List {
    ~~~
    struct Node { ~~~ };
    Node *erase(Node *);
    Node *erase_after(Node *);
    auto erase_before(Node *) -> Node *;
};

auto List::erase(Node *p) -> Node * { ~~~ }
List::Node *List::erase_after(Node *p) { ~~~ }
```

122

Deduced Return Types in C++14

- C++14 lets you use auto and skip the trailing return type.
- In that case, the compiler deduces the return type (if possible) from the return expression type:

```
template <typename T, typename U>
auto difference(T t, U u) {         // C++14 only
    return t - u;
}
```

- Here, auto is a placeholder for a return type deduced from the return expression.
- Return type deduction is a ***major*** convenience when writing function templates.

123

Deduced Return Types in C++14

- However, be careful that the intended type is deduced!
- Here's a function template with an explicitly-stated return type:

```
template <typename T, typename S>
T &process(T &t, S const &s) {      // return is T &
    return t += s;
}
```

- Here's the same template with a (C++14) auto return type:

```
template <typename T, typename S>   // in C++14...
auto process(T &t, S const &s) {    // deduce return type
    return t += s;                  // from expression
}
```

124

Deduced Return Types in C++14

- Recall how compilers deduce template parameter types.
- For example,

```
template <typename T>
void f(T arg);
~~~
string a, b;
f(a += b);  // a += b is "string &"; deduce T as string
```

- The return type of `string::operator +=` is “`string &`”.
- Thus, `a += b` yields a “`string &`”.
- Still, the compiler deduces `f(a += b)` to be `f<`***`string`***`>(a += b)`, not `f<`***`string &`***`>(a += b)`.

125

Deduced Return Types in C++14

- Compilers deduce `auto` return types in (almost!) the same way that they deduce template argument types.
- For example,

```
template <typename T, typename S>
auto process(T &t, S const &s) {    // deduced as T, even
    return t += s;                  // if this is T &
}
```

- We probably want the `decltype` rules for deduction to apply in this case, not the `auto` rules…

126

C++14 decltype(auto)

- Recall:

```
vector<string> v;        // operator [] returns string &
~~~
auto a = v[0];           // a is string
auto &b = v[0];          // b is string &
decltype(v[0]) c = v[0]; // c is string &
```

- The fix is easy: tell the compiler you want decltype deduction:

```
template <typename T, typename S>
decltype(auto) process(T &t, S const &s) {  // reference
    return t += s;            // if this is a reference
}
```

127

A decltype(auto) Gotcha

- Recall that decltype treats a parenthesized lvalue expression differently from an unparenthesized one:

```
template <typename T, typename S>
decltype(auto) process(T &t, S const &s) {
    auto local = t += s;    // local is not a reference
    // three mutually-exclusive returns
    return local;   // (1) return a copy of value
    return t += s;  // (2) return a reference to t
    return (local); // (3) return a reference to local!
}
```

- Returning a reference to local (3) could lead to undefined behavior…

128

A decltype(auto) Gotcha

```
template <typename T, typename S>
decltype(auto) process(T &t, S const &s) {
    auto local = t += s;
    ~~~
    return (local); // (3) return a reference to local!
}
~~~
auto copy = process(a, b);          // OK: copy local
decltype(auto) ref = process(a, b); // bind to local!
```

- `ref`'s definition binds it to `local` after `local`'s lifetime has ended.
- Any attempt to access `ref` will yield undefined behavior.

✓ *Don't parenthesize entire return expressions in the presence of `decltype(auto)`… or in general.*

129

decltype(auto)

- `decltype(auto)` is intended primarily for return type deduction, but you can use it elsewhere:

```
vector<string> v;
~~~
auto a = v[0];              // a is string
auto &b = v[0];             // b is string &
decltype(v[0]) c = v[0];    // c is string &
decltype(auto) d = v[0];    // d is string &, too
```

130

Initialization

- Consider the following simple class:

```
class X {
public:
    X(int);
};
```

- The class has an implicitly-declared copy constructor, copy assignment, and destructor.
- All are public, inline, and trivial.

131

Direct Initialization

- C++03 offers different ways to do the “same” initialization:

```
X a (42);       // direct initialization
X b = 42;       // copy initialization
X c = X(42);    // copy initialization
```

- The first initialization is direct initialization.
- It invokes the constructor `X(int)`.
- The other two initializations are copy initializations…

132

Copy Initialization

- These are copy initializations.

```
X b = 42;      // copy with implicit conversion
X c = X(42);   // copy with explicit conversion
```

- Conceptually, they both do the following:
  - initialize a temporary `X` object using the constructor `X(int)`,
  - use the copy constructor to initialize the `X` being declared, and
  - call the `X` destructor to destroy the temporary.
- The first definition creates the temporary implicitly.
- The second creates it explicitly.

133

Temporaries and Copy Initialization

- Under certain common circumstances, compiler can “optimize away” the construction and destruction of the temporary object:

```
X a (42);
X b = 42;      // can be optimized to X b (42);
X c = X(42);   // can be optimized c (42);
```

- Actually, it’s more than an optimization.
- It’s a semantic transformation because it eliminates two functions and any side effects they may have.
- The various C++ Standards explicitly permit this transformation.

134

Temporaries and Copy Initialization

- A compiler need not do the optimization.
- However, most compilers will.
- Still, it's best to say precisely what you mean:

```
X a (42);    // preferred
```

- For predefined types, use whichever form you think is clearest…

```
int i = 12;
int j (12);
```

- …but be consistent!

135

Honoring Access Control

- The compiler still must check the access of calls to functions that are optimized away:

```
class Y {
public:
    Y(int);
    ~Y();
private:
    Y(Y const &);
};
~~~
Y e = Y(1066);  // error! can't access copy constructor
Y f (1066);     // OK: doesn't use copy constructor
```

136

explicit

- Declaring a single argument constructor as `explicit` further restricts the initialization syntax:

```
class Z {
public:
    explicit Z(int);
    Z(Z const &);
    ~~~
};

Z g = 1066;     // error! implicit conversion
Z h (1066);     // OK: direct init, no conversion
Z i = Z(1066);  // OK: explicit conv and copy init
```

137

Copy vs. Direct Initialization

- C++ uses copy initialization:
  - in = initializers
  - to pass arguments
  - to return values
  - to throw exceptions (to copy exception objects)
  - to catch exceptions
- It uses direct initialization everywhere else, including:
  - in member initialization lists
  - in new expressions
  - to create anonymous temporary objects

138

Zero, Default and Value Initialization

- To ***zero-initialize*** means:
  - For an object of scalar type, initialize to 0 (zero) converted to the object's type.
  - For an object of class type, zero initialize each non-static data member.
  - For an array, zero initialize each element.
- To ***default-initialize*** means:
  - For an object of class type, initialize by applying the default constructor.
  - For an array, default initialize each element.
  - Otherwise, do nothing.
- To ***value-initialize*** means to default-initialize if possible; otherwise zero-initialize.

139

Zero, Default and Value Initialization

- For objects of scalar type with automatic or dynamic storage, default initialization means do nothing:

```
int *p;             // default init; do nothing
p = new int;        // default init; do nothing
```

- Objects of scalar type with static storage are zero-initialized:

```
static int *q;      // zero init: it's static
```

- Objects of scalar type with dynamic storage are zero-initialized when the initializer is ():

```
q = new int ();     // zero init: explicit () initializer
```

140

Uniform Initialization Syntax

- C++11 extends the existing initialization syntax to include ***braced initializers***.
- Braced initializers force value initialization.
- For example:

```
string a = "Hello...";      // copy init
string b ("Hello!");        // direct init
string c = {"Hello..."};    // copy init
string d {"Hello!"};        // New! direct init

int f ();           // function
int g {};           // New! object with value init to 0
```

141

C++'s "Most Vexing Parse"

- Braced initialization can help with vexing parses:

```
string s ();    // function
string t {};    // New! object with default init

Date date1 (Month(m), Day(d), Year(y)); // function
Date date2 {Month(m), Day(d), Year(y)}; // New! object

vector<int> v1 (            // function
    istream_iterator<int>(cin), istream_iterator<int>()
);
vector<int> v2 {            // New! object
    istream_iterator<int>(cin), istream_iterator<int>()
};
```

142

Restrictions on Narrowing

- Braced initialization takes an intelligent approach to preventing narrowing of types:

```
int val1 = 12.3;          // OK, but truncates!
int val2 {12.3};          // error: no narrowing allowed
int value1 = 12;          // OK
char c1 {value1};         // error: no narrowing allowed
int const value2 = 12;
char c2 = value2;         // OK, if 12 fits in a char
```

143

Initializer Lists

- In C++03, it was difficult to initialize an STL container with a range of values:

```
int a[] = { 1, 4, 1, 4, 2 };
int const n = sizeof(a)/sizeof(a[0]);
vector<int> v (a, a+n);
```

- C++11 let's you initialize a container using braced initializer list:

```
vector<int> v { 1, 4, 1, 4, 2 };
```

- The standard `vector` class template has a constructor that accepts an `initializer_list` as an argument…

144

Initializer Lists and Construction

- This container class template has such a constructor:

```
template <typename T>
class Cont {
public:
    Cont();
    Cont(size_t n);
    Cont(size_t n, T const &val);
    Cont(initializer_list<T> init);
    ~~~
};
```

- `initializer_list` objects are small, like iterators.
- You almost always pass them by value, not by reference.

145

Normal Confusion

- Consider these initializations:

```
Cont<int> a;            // default ctor
Cont<int> b (12);       // one-argument ctor
Cont<int> c (12, -1);   // two-argument ctor
Cont<int> d {1, 2, 3};  // initializer_list ctor
```

- So far, so good, until...

```
Cont<int> e {12};       // initializer_list ctor
Cont<int> f {12, -1};   // initializer_list ctor
```

- Overload resolution strongly favors initializer lists.

146

Edge Cases

- ...and let's not forget:

```
Cont<int> g {};    // default ctor
Cont<int> h ({}); // initializer_list ctor
Cont<int> i {{}}; // initializer_list ctor
Cont<int> j = {}; // default ctor
```

- ...and remember the difference:

```
auto a = {12};    // a is initializer_list<int>
auto b {12};      // b is int
auto c = {1, 2}; // c is initializer_list<int>
auto d {1, 2};    // error!
initializer_list<int> e {1, 2}; // OK...
```

147

Overload Resolution Confusion

- The overload resolution rules are complex.
- In Modern C++, they're even more complex because of special rules for matching `initializer_list` parameters.
- One example should be enough to ruin your day:

```
template <typename T>
class Cont {
public:
    Cont(int n, int m);                 // #1
    Cont(initializer_list<T> init);     // #2
    ~~~
};
Cont<int> cont {1, 2};                  // call #2
```

148

Overload Resolution Confusion

- These rules are *supposed* apply to `initializer_list` & constructors as well…but:

```
template <typename T>
class Cont {
public:
    Cont(int n, int m);                  // #1
    Cont(initializer_list<T> &init);     // #2
    ~~~
};
Cont<int> cont {1, 2};                   // call #1, often
```

- It's easy to contrive even more complex examples.

✓ *Pass `initializer_lists` by value.*

149

std::initializer_list

- An `initializer_list` is a standard, container-like class template with customary member types:

```
template <typename E>
class initializer_list {
public:
    typedef E value_type;
    typedef E const &reference;
    typedef E const &const_reference;
    typedef size_t size_type;
    typedef E const *iterator;
    typedef E const *const_iterator;
    ~~~
};
```

150

std::initializer_list

- The typical implementation uses two pointers, or a pointer and a length:

```
template <typename E>
class initializer_list {
public:
    ~~~
    constexpr initializer_list() noexcept;
    constexpr size_t size() const noexcept;
    constexpr E const *begin() const noexcept;
    constexpr E const *end() const noexcept;
private:
    E *begin_, *end_;
};
```

151

Initializer List Details

- An `initializer_list` provides access to an array of constant elements.
- Initializer lists are small.
- Passing them by value is cheap.
  - Copying an initializer list doesn't copy the array elements.
- The return values of `begin` and `end` for an empty initializer list are unspecified, but compare equal.
- An implementation is allowed to optimize initializer lists.
- You're not allowed to explicitly specialize or partially specialize an `initializer_list`.

152

Argument Evaluation Order is Undefined

- Consider a function that returns a unique value each time it's called:

```
size_t value() {
    static size_t a = 0;
    return a++;
}
```

- What arguments will be passed in the call to `func`?

```
func(value(), value(), value());
```

- We don't know, since the order of calls to `value` is unspecified.

153

Initializer Lists Fix Evaluation Order

- Here's a situation that looks similar:

```
vector<size_t> v { value(), value(), value() };
```

- What is the initial sequence contained by `v`?
- It's well-defined, because the order of evaluation of elements of an initializer list is also well-defined: left to right.
- This property of initializer lists is sometimes used to fix evaluation order in contexts that otherwise would not.
- Note that C++17 fixes evaluation order to a much greater degree than do C++98/03/11/14.
  - However, order of argument evaluation is still unspecified in C++17.

154

Non-Member begin and end

- The canonical form of a loop that traverses a standard container, s, looks like:

```
for (auto i = s.begin(); i != s.end(); ++i) {
    // do something with *i
}
```

- A loop that traverses an array, x, looks similar but nonetheless different:

```
for (auto i = x; i != x + N; ++i) {
    // do something with *i
}
```

155

Reverse Iteration

- By the way, here's the canonical reverse iteration:

```
for (auto i = s.end(); i != s.begin(); ) {
    --i;
    // do something with *i
}
```

- Or, for an array:

```
for (auto i = x + N; i != x; ) {
    --i;
    // do something with *i
}
```

- Any complaints?

156

Even For Indexes

- What's wrong with:

```
T x[] = { ~~~ };
auto const N = sizeof(x)/sizeof(x[0]);
~~~
for (auto i = N-1; i >= 0; --i) { ~~~ }      // no.
```

- Infinite loop.
- The recommended style keeps the index/pointer/iterator between begin and end, and doesn't rely on random-access operations.

```
for (auto i = N; i != 0; ) { --i; ~~~ }     // yes.
```

157

Non-Member begin and end

- C++11 now provides non-member forms for begin and end that work equally well with standard containers and arrays.
- The canonical form of a loop that traverses a sequence, s, looks like:

```
for (auto i = begin(s); i != end(s); ++i) {
    // do something with *i
}
```

- This loop works whether s is a container or an array.
- The same goes for:

```
sort(begin(s), end(s));
```

158

Non-Member begin and end

- The C++11 standard header `<iterator>` defines begin and end as a small collection of function templates.
- For example, the end function templates for non-constant and constant containers look something like:

```
template <class C>
auto end(C &c) -> decltype(c.end()) {
    return c.end();
}

template <class C>
auto end(C const &c) -> decltype(c.end()) {
    return c.end();
}
```

159

Non-Member begin and end

- A third end function template applies to built-in arrays:

```
template <typename T, size_t N>
constexpr T *end(T (&x)[N]) noexcept {
    return x + N;
}
```

- The template deduces the array's dimension.

160

Non-Member begin and end

- The standard header <initializer_list> defines non-member begin and end for initializer_list<E>:

```
template <typename E>
constexpr E const *
begin(initializer_list<E> il) noexcept {
    return il.begin();
}

template <typename E>
constexpr E const *
end(initializer_list<E> il) noexcept {
    return il.end();
}
```

161

Range-Based For-Statements

- C++11 provides ***range-based for-statements*** as an even more concise notation for traversing sequences.
- For example,

```
vector<int> v;
~~~
for (auto e: v) {
    cout << e << '\n';
}
```

- Here, e is not an iterator.
- Rather, e takes on the value of each successive element in v.

162

Range-Based For-Statements

- A loop-control variable of non-reference type contains a ***copy*** of each successive value in the sequence.
- Thus, modifying the variable's value doesn't alter the sequence:

```
vector<int> v;
~~~
for (auto e: v) {
    ++e;            // has no effect on v
}
```

163

Range-Based For-Statements

- A loop-control variable of reference type provides a reference to each successive value in the sequence.
- This provides a means to modify elements in the sequence:

```
vector<int> v;
~~~
for (auto &e: v) {
    if (e < 0) {
        e = -e;     // modify element in sequence
    }
}
```

164

Range-Based For-Statements

- A range-based for-statement of the form:

```
for (declaration: expr)
    statement
```

is more-or-less equivalent to:

```
for (auto b = begin(expr), e = end(expr); b != e; ++b) {
    declaration = *b;
    statement
}
```

- Note that the end value is “hoisted” out of the loop.
- It’s not recalculated each time.

165

“For” Example

```
for (auto val: {1, 3, 5, 7, 9})
    thats_odd(val);
~~~
map<string, string> farm;
farm["cow"] = "moo";
farm["sheep"] = "baa";
farm["swan"] = "";
farm["dog"] = "woof";
~~~
for (auto &e: farm) {
    cout << "The " << e.first
         << " says " << e.second << '!' << endl;
}
```

166

Non-Member begin and end

- Most algorithms that operate on iterator ranges work not only with standard container iterators, but also with built-in pointers.
- For example,

```
vector<int> v;
int x[N];
~~~
sort(v.begin(), v.end());    // sort vector v
sort(x, x + N);              // sort array x
```

- Passing pointers as arguments is similar, but not quite the same, as passing container iterators.
- Along the same lines…

167

Range-Based For-Statements

- A range-based for-statement can use type-specifiers other than `auto`:

```
vector<float> v;
~~~
for (double d: v)
    // use d
```

- However, it won't compile unless there's a conversion from the sequence's element type to the loop-control variable's type.

168

Convergence?

- There's clearly a trend in C++ to give containers (including arrays and initializer_lists) a common user interface.
- In addition to non-member begin and end, C++11 offers non-members:
  - rbegin and rend for reverse_iterators,
  - cbegin and cend for const_iterators, and
  - crbegin and crend for const_reverse_iterators.
- C++14 also provides non-member size, empty, and data operations.

169

Non-Member Container Operations

```
template <typename Cont>
void rp(Cont const &c) {
    if (!empty(c)) {
        for (auto i = crbegin(c); i != crend(c); ++i) {
            cout << *i << endl;
        }
    }
}
~~~
int a[] = { 7, 2, 7 };
rp(a);
auto b = { 7, 5, 7 };
rp(b);
vector<int> v { 7, 8, 7 };
rp(v);
```

170

Range-Based For vs. Algorithm

- It’s simple to transform a sequence with a range-based for:

```
vector<int> v;
~~~
for (auto &e: v) {
    e = -e;
}
```

- We could also use a generic algorithm:

```
transform(begin(v), end(v), begin(v),
          [](auto e){ return -e; });
```

- There is as yet no consensus as to which is the more idiomatic rendering.

171

172

Bibliography

- Abrahams [2010]. David Abrahams, Rani Sharoni, Doug Gregor, N3050=10-0040
- ISO [1998]. *ISO/IEC Standard 14882:1998, Programming languages—C++*.
- ISO [2003]. *ISO/IEC 14882:2003: Programming languages — C++*.
- ISO [2005]. *ISO/IEC TR 19768, C++ Library Extensions*.
- ISO [2011a]. *ISO/IEC 14882:2011: Programming languages — C++*.
- ISO [2011b]. *ISO/IEC 9899:2011: Programming languages — C*.
- ISO [2014]. *ISO/IEC Standard 14882:2014, Programming languages—C++*.
- Karlsson [2004]. Bjorn Karlsson, "The Safe Bool Idiom". *The C++ Source*. www.artima.com/cppsource/safebool.html

173

Bibliography

- Meyers [2015]. Scott Meyers, Effective Modern C++. O'Reilly.
- Stroustrup [2013]. Bjarne Stroustrup, *The C++ Programming Language, 4th ed*. Addison-Wesley.
- Sutter [2013]. Herb Sutter, "GotW #94 Solution: AAA Style (Almost Always Auto)", *Sutter's Mill*. herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
- Sutter [2013a]. Herb Sutter, "GotW #91: herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/
- Sutter [2014]. Herb Sutter, "Back to the Basics! Essentials of Modern C++ Style", *CppCon*. www.youtube.com/watch?v=xnqTKD8uD64
- Vandevoorde [2003]. David Vandevoorde and Nicolai Josuttis, *C++ Templates*. Addison-Wesley.

174

